

Database Management Systems Lab

LAB MANUAL



Academic Year: 2023-24



**GOKARAJU RANGARAJU INSTITUTE OF
ENGINEERING AND TECHNOLOGY (Autonomous)**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE: DATABASE MANAGEMENT SYSTEMS LAB

COURSE CODE: GR22A2072

ACADEMIC YEAR: 2023-24

SEMESTER: I

Marks: Int: 30 Extn: 70 Total: 100

Course Objectives:

1. Develop the logical design of the database using data modeling concepts such as Relational model
2. Infer the data models and use of queries in retrieving the data.
3. Create a relational database using a relational database package.
4. Manipulate a database using SQL.
5. Render the concepts of database system structure.

Course Outcomes:

1. Construct the schema of the database and modify it.
2. Compile a query to obtain the aggregated result from the database.
3. Speculate the concepts of various database objects.
4. Compare the use of procedure and function in database.
5. Use triggers and packages to create applications in the database.

COURSE SYLLABUS

Task- 1:

DDL commands (Create, Alter, Drop, Truncate)

1. Create a table EMP with the following structure.

Name	Type

EMPNO	NUMBER(6)
ENAME	VARCHAR2(20)
JOB	VARCHAR2(10)
MGR	NUMBER(4)
DEPTNO	NUMBER(3)
SAL	NUMBER(7,2)

2. Add a column commission to the emp table. Commission should be numeric with null values allowed.

3. Modify the column width of the job field of emp table.
4. Create dept table with the following structure.

NameType

```

-----
DEPTNO          NUMBER(2)
DNAME           VARCHAR2(10)
LOC             VARCHAR2(10)
DEPTNO as the primary key

```

5. Add constraints to the emp table that is empno as the primary key and deptno as the foreign key.
6. Add constraints to the emp table to check the empno value while entering (i.e) empno > 100. Salary value by default is 5000, otherwise it should accept the values from the user.
7. Add columns DOB to the emp table. Add and drop a column DOJ to the emp table.

Task- 2: DML COMMANDS (Insert, Update, Delete)

1. Insert 5 records into dept Insert few rows and truncate those from the emp1 table and also drop it.
2. Insert 11 records into emp table.
3. Update the emp table to set the value of commission of all employees to Rs1000/- who are working as managers.
4. Delete only those who are working as supervisors.
5. Delete the rows whose empno is 7599.

Task- 3: DQL COMMAND (Select)- SQL Operators and Order by Clause

List the records in the emp table order by salary in descending order.

1. Display only those employees whose deptno is 30.
2. Display deptno from the table employee avoiding the duplicated values.
3. List all employee names, salary and 15% rise in salary. Label the column as pay hike.
4. Display the rows whose salary ranges from 15000 to 30000.
5. Display all the employees in dept 10 and 20 in alphabetical order of names.
6. List the employee names who do not earn commission.
7. Display all the details of the records with 5 character names with 'S' as starting character.
8. Display joining date of all employees in the year of 1998.
9. List out the employee names whose salary is greater than 5000 and less than 6000

Task- 4: SQL Aggregate Functions, Group By clause, Having clause

1. Count the total records in the emp table.
2. Calculate the total and average salary of the employee.
3. Determine the max and min salary and rename the column as max-salary and min_salary.
4. Find number of departments in employee table.

5. Display job wise sum, average, max, min salaries.
6. Display maximum salaries of all the departments having maximum salary > 2000
7. Display job wise sum, avg, max, min salaries in department 10 having average salary is greater than 1000 and the result is ordered by sum of salary in descending order.

Task- 5: SQL Functions

1. Display the employee name in upper case and employee number in lower case.
3. Display the month name of date "14-jul-09" in full.
4. Display the Date of joining of all employees in the format "dd-mm-yy".
5. Display the date two months after the Date of joining of employees.
6. Display the last date of that month in "05-Oct-09".
7. Display the rounded date in the year format, month format, day format in the employee
8. Display the commissions earned by employees. If they do not earn commission, display it as "No Commission".

Task- 6: Nested Queries

1. Find the third highest salary of an employee.
2. Display all employee names and salary whose salary is greater than minimum salary of the company and job title starts with 'M'.
4. Write a query to display information about employees who earn more than any employee in dept 30.
5. Display the employees who have the same job as Jones and whose salary is greater than or equal to the salary of Ford.
6. List out the employee names who get the salary greater than the maximum salaries of dept with dept no 20, 30.
7. Display the maximum salaries of the departments whose maximum salary is greater than 9000.
8. Create a table employee with the same structure as the table emp and insert rows into the table using select clauses.
9. Create a manager table from the emp table which should hold details only about the managers.

Task- 7:

Joins, Set Operators.

1. Display all the employees and the departments implementing a left outer join.
2. Display the employee name and department name in which they are working implementing a full outer join.
3. Write a query to display their employee names and their managers' name and salary for every employee.
4. Write a query to output the name, job, empno, deptname and location for each dept, even if there are no employees.

5. Display the details of those who draw the same salary.

Task- 8:Views

1. Create a view that displays the employee id, name and salary of employees who belong to 10th department.
2. Create a view with read only option that displays the employee name and their department name.
3. Display all the views generated.
4. Execute the DML commands on views created and drop them.

Task- 9: Practice on DCL commands,sequence and indexes.

Task- 10:

1. Write a PL/SQL code to retrieve the employee name, join date and designation of an employee whose number is given as input by the user.
2. Write a PL/SQL code to calculate tax of employee.
3. Write a PL/SQL program to display top ten employee details based on salary using cursors.
4. Write a PL/SQL program to update the commission values for all the employees' with salary less than 2000, by adding 1000 to the existing values.

Task- 11:

1. Write a trigger on employee table that shows the old and new values of employee name after updating on employee name.
2. Write a PL/SQL procedure for inserting, deleting and updating the employee table.
3. Write a PL/SQL function that accepts the department number and returns the total salary of that department.

Task- 12:

1. Write PL/SQL program to handle predefined exceptions.
2. Write PL/SQL program to handle user defined exception.
3. Write a PL/SQL code to create
 - a. Package specification
 - b. Package body to insert ,update, delete and retrieve data on emp table.

Text/Reference Books

1. The Complete Reference,3rd edition by James R.Groff, Paul N.Weinberg, Andrew J. Oppel
2. SQL & PL/SQL for Oracle10g, Black Book, Dr.P.S.Deshpande

How to write and execute sql statements:

- 1) Open Oracle application by the following navigation

Start->All Programs->Oracle 10g Express Edition->Run SQL Command Line.

2) You will be asked for user name, password.

Enter user name, password as given by the administrator.

3) Upon successful login, SQL prompt will be displayed. (SQL>).

```
SQL> SELECT ename,empno,
```

```
2      sal from
```

```
3      emp;
```

where 2 and 3 are the line numbers and rest are the statements.

4) After the login session, use the 'exit' command at SQL prompt to safely log out of the terminal.

How to Write and execute pl/sql programs:

1) Open your oracle application by the following navigation

Start->all programs->oracle 12c enterprise edition->Run SQL command line.

2) You will be asked for user name, pass word and host string

You have to enter user name, password and host string as given by the administrator. It will be different from one user to another user.

3) Upon successful login you will get SQL prompt (SQL>).

In two ways you can write your programs:

a) directly at SQL prompt

b) or in sql editor.

If you type your programs at sql prompt then screen will look like follow:

```
SQL> SELECT ename,empno,
```

```
sal from
```

```
emp;
```

where 2 and 3 are the line numbers and rest is the command /program.....

To execute above program/command you have to press '/' then enter.

Here editing the program is somewhat difficult; if you want to edit the previous command then you have to open sql editor (by default it displays the sql buffer contents). By giving 'ed' at sql prompt.(this is what I mentioned as a second method to type/enter the program).

in the sql editor you can do all the formatting/editing/file operations directly by selecting menu options provided by it.

To execute the program which was saved; do the following

```
SQL> @ programname.sql
```

Or

SQL> Run programname.sql

Then press '\ ' key and enter.

This is how we can write, edit and execute the sql command and programs. Always you have to save your programs in your own logins.

INTRODUCTION TO SQL

SQL has clearly established itself as the standard relational database language. The original version was developed by IBM at its SAN JOSE RESEARCH LABORATORY (now THE ALMADEN RESEARCH CENTER). This language originally called SEQUEL, was implemented as a part of the system R project in the early 1970's. The SEQUEL language has evolved since then and its name has changed to SQL (STRUCTURED QUERY LANGUAGE).

Numerous products now support the SQL language. Although the product versions of SQL differ in several language details, the differences are the most part minor.

In 1986 ANSI (AMERICAN NATIONAL STANDARDS INSTITUTE) published an SQL standard. In 1979, ORACLE corporation introduced the first commercial available implementation of SQL.

FEATURES OF SQL

1. SQL is an English-like language. It uses the words such as SELECT, INSERT in its commands
2. SQL is a non-procedural language, i.e., you specify what information you require, not how to get it.
3. SQL is an interactive query language that allows users to use SQL statements to retrieve data and display it on the screen, providing a convenient easy to use tool for ad-hoc database queries.
4. SQL is a data administration language that defines the structure of the database, controls the user access to data.
5. SQL is a client/server language that allows application programs on PCs connected via a LAN to communicate with database servers that store shared data. Applications using client/server architecture make optimum use of the PCs and servers and minimize network traffic.
6. SQL is a database gateway language: In a computer network with a mix of different DBMS products, SQL is often used in a gateway that allows one brand of DBMS to communicate with another brand.
7. Using SQL the user has to specify what operation is to be performed, how to perform that operation will be handled by the RDBMS.
8. Any complex query can be performed very easily using SQL.

9. Unified language to perform different operations over the database.

PSEUDO COLUMN

A pseudo column behaves like a table column but is not actually stored in the table. Data can be selected from pseudo columns, but cannot insert, update or delete the values.

Example:

- CURRVAL
- NEXTVAL
- SYSDATE
- LEVEL
- ROWID
- ROWNUM

Some System tables

USER_USERS	-	contains information about all the users.
TAB	-	contains information about all the objects created by the user.
DUAL	-	dummy table containing one column and one row.

ENDING A SQL COMMAND

User can end a SQL command in three ways:

1. With a semi column(;) - To execute the SQL command
2. With a slash on a new line(/) - To execute the SQL command
3. With a blank line - To ignore the SQL command.

FEATURES OF ORACLE 8i

The database development and management environment from Oracle provides for client/server DBMS, including a number of utilities useful in the development of database systems:

- Oracle Navigator – useful in the creation of new tables within a visual environment
- Oracle Browser – useful in the creation of SQL queries as well as QBE (query by example queries).
- Oracle Forms – these create/develop menus and/or graphical forms applicable to user applications.
- Oracle Reports – generation of reports in printing and display of data
- SQL Plus – creation testing and debugging command-prompt SQL queries
- Oracle Graphics – generation of graphical representations and visual aids based on information stored in the database.
- Enterprise Manager – management and tuning of the database
- Oracle Webserver – used to create a WWW site to enable users develop dynamic web pages using Oracle

databases.

FEATURES OF ORACLE 9i

The improvement in Oracle 9i aims at improved optimization of conventional business applications, stimulations of the emergent hosted applications market and facilitation of vital advancement capabilities required for internet-based business.

As regards availability, the Oracle 9i DB provides an extension of Internet Database Availability in principal areas, including online data evolution, enhanced disaster recovery environments and precision database repair. Enhanced data recovery environments have seen implementation through the introduction of many features and enhancements, which can condense into the four categories below:

It also implements new online architecture featuring the following:

- Changes on any physical table attributes
- Changes on many of the logical attributes as well
- Secondary indexes on IOTs may be created and/or rebuilt online
- Online creation and analysis of indexes
- Online repair for physical guesses proven invalid

FEATURES OF ORACLE 10g

There are several hundred feature differences between the 9i and 10g versions, but the following offers and worth summary of the new features implemented on the Oracle 10g database version:

- SQL optimizer internals have been majorly altered
- Oracle grid computing
- RAC in Oracle 10g enhanced to enable dynamic scalability on server blades
- Oracle Enterprise Manager (OEM) in 10g completely redesigned
- The OEM performance pack now includes ASH and AWR tables, same as the Diagnostic Pack
- The imp utility on the Data Pump has been replaced with impdp
- SQLTuning Advisor
- Automatic Database Diagnostic Monitor (ADDM)
- SQLAccess Advisor

FEATURES OF ORACLE 11g

At the time of launch, Oracle 11g was a promise to consumers as an implementation of 482 new features. These changes aimed at satisfying a then hungry commercial database market that had since developed rather mature and high expectations of an RDBMS solution.

A list of enhancements in Oracle 11g is as outlined below:

to ILM

s much of the
and upgrades

that customers
d features. An

s generations,

ses as well as

base allowing

Data Type Syntax	Oracle 9i	Oracle 10g	Oracle 11g	Explanation
char(size)	Maximum size of 2000 bytes.	Maximum size of 2000 bytes.	Maximum size of 2000 bytes.	Where <i>size</i> is the number of characters to store. Fixed-length strings. Space padded.

Where p is the precision

timestamp (<i>fractional seconds precision</i>)	<i>fractional seconds precision</i> must be a number between 0 and 9. (default is 6)	<i>fractional seconds precision</i> must be a number between 0 and 9. (default is 6)	<i>fractional seconds precision</i> must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds. For example: timestamp(6) Includes year, month, day,
--	--	--	--	---

Data Type Syntax	Rowid			Explanation
	Oracle 9i	Oracle 10g	Oracle 11g	
	The format of the rowid is: BBBBBBB.RRRR.FFFF	The format of the rowid is: BBBBBBB.RRRR.FFFF	The format of the rowid is: BBBBBBB.RRRR.FFFF	Fixed-length binary data. Every record in the database has a
Rowid	Where BBBBBBB is the block in the database file; RRRR is the row in	Where BBBBBBB is the block in the database file; RRRR is the row in	Where BBBBBBB is the block in the database file; RRRR is the row in	

the block;
FFFFFF is the
database file.

the block;
FFFFFF is the
database file.

the block;
FFFFFF is the
database file.

physical
address
or **rowid**

CLASSIFICATION OF SQL COMMAND SET

- DDL (Data definition language)
- DRL or DQL (Data Retrieval Language or Data Query Language)
- DML (Data Manipulation language)
- TCL (Transaction Control language)

- DCL (Data Control Language)

Types of commands used in SQL

- DDL
 - CREATE
 - ALTER
 - TRUNCATE
 - DROP
 - RENAME
 - COMMENT
- DRL
 - SELECT
- DML
 - INSERT
 - UPDATE
 - DELETE
- TCL
 - COMMIT
 - ROLLBACK
 - SAVEPOINT
- DCL
 - GRANT
 - REVOKE
 - ROLES

DATA DEFINITION LANGUAGE

Data definition language is the set of SQL commands that create and define objects in the database, storing their definitions in the data dictionary. It is a set of commands that define database objects.

Database objects are data structures that are used to store data.

- 1) CREATE (creates new database objects)

- 2) ALTER (alters existing database objects)
- 3) DROP (drops existing database objects permanently from the database)
- 4) TRUNCATE (deletes the entire data permanently from the database but creates schema)
- 5) RENAME (Renames the table name)
- 6) COMMENT (adds text to describe either table or column)

1) CREATE:

The CREATE TABLE statement is used to create a table in a database.

Syntax:

```
CREATE TABLE table_name
(
  Column_name1 data_type,
  Column_name2 data_type,
  Column_name3 data_type,
  ....
)
```

The data type specifies what type of data the column can hold.

Rules for naming a table:

- ✓ The first step in creating a table is to name it. Enter the table's name after the words CREATE TABLE.
- ✓ The name you choose for a table must follow the standard rules for naming an ORACLE database object.
- ✓ It must start with any letter (a-z) and it may contain letters, numerals and special character (_).
- ✓ It is the same whether upper or lower-case letters are used, and it may be up to 30 characters.
- ✓ May not duplicate the name of another table or view or a ORACLE reserve word.
- ✓ If a table name enclosed quotes wherever it is used, the first three rules do not apply, instead the following rules apply.
- ✓ The name may contain any combination of characters except that it may not contain a double code.
- ✓ Upper and lower-case letters are not equivalent.

Creating a table from a table:

Tables are also created using the CREATE TABLE command from the other table. The column names must be specified and the tables that are owned by the user must be different from one another. The column names in the table must be unique and can be duplicated across the table.

Syntax:

CREATE TABLE <table-name> (<column-name><data type(size)> ...);

Or

CREATE TABLE <new-table-name> AS SELECT (col1, col2 ...) FROM <old-table-name>

Or

CREATE TABLE <new-table-name> AS SELECT * FROM <old-table-name>;

Example:

CREATE TABLE Emp(Empno number, Empname varchar2, ...);

Creating Tables

i) CREATE TABLE EMP(EMPNO Number(4) CONSTRAINT Pk EMP PRIMARY KEY,
ENAME varchar2(25),
JOB varchar2(20),
MGR NUMBER(10),
HIRE_DATE Date,
SAL Number(10),
COMM Number(5),
DEPTNO Number(2) CONSTRAINT FkDeptno
REFERENCES DEPT(DEPTNO));

ii) CREATE TABLE DEPT
(DEPTNO Number(2) constraint dept_deptno_pk primary key,
DNAME VARCHAR2(14),
LOC VARCHAR2(13));

- For listing all the tables in the database:

SQL> SELECT * FROM Tab;

- Listing a table definition (Description of a table)

SQL>DESCRIBE EMP; (or) **DESC EMP;**

2) ALTER:

Used to alter or in other words, change the structure of a table, view or index. This is particularly used when there is a scenario wherein the properties of fields inside a table, view, and index are supposed to be updated.

*to add a column

- *to add an integrity constraint
- *to redefine a column (data type, size, default value)
- *to modify storage characteristics or other parameters
- *to enable, disable, or drop an integrity constraint or trigger
- *to explicitly allocate an extent

Syntax:

ALTER TABLE <table-name> ADD (col1 datatype, col2 datatype, ...)

Or

ALTER TABLE <table-name> ADD CONSTRAINT <constraint-name><constraint-type>

Or

ALTER TABLE <table-name> MODIFY (col1 datatype, col2 datatype ...)

Or

ALTER TABLE <table-name> ENABLE CONSTRAINT <constraint-name>

Or

ALTER TABLE <table-name> DISABLE CONSTRAINT <constraint-name>

Or

ALTER TABLE <table-name> DROP CONSTRAINT <constraint-name>

Or

ALTER TABLE<table-name> DROP COLUMN<column name>

Or

ALTER TABLE<table-name> RENAME COLUMN col1 TO col2;

Or

ALTER TABLE<table-name > RENAME CONSTRAINT dname_ukey
TO dname_unikey;

3) DROP:

The DROP command removes a table from the database. All the table's rows, indexes and privileges will also be removed. No DML triggers will be fired. The operation cannot be rolled back.

Syntax:

DROP TABLE table-name;

Example:

DROP TABLE student;

4) TRUNCATE:

Removes all rows from a table. The operation cannot be rolled back and no triggers will be fired. As such, TRUNCATE is faster and doesn't use as much undo space as a DELETE.

Syntax:

```
TRUNCATE TABLE <table-name>;
```

Example:

```
TRUNCATE TABLE EMPLOYEE;
```

5) RENAME:

We can rename any table by using RENAME command. The data will not be lost. Only the table name will be changed to new name.

Syntax:

```
RENAME <old table name> TO <new table name>
```

Example:

```
RENAME employee TO emp;
```

6) COMMENT:

In Oracle, comments may be introduced in two ways:

1. With `/*...*/`, as in C.
2. With a line that begins with two dashes--.

Thus, --This is a comment.

Example: `SELECT */*` and so is this `*/ FROM R;`

Adding a comment

You can add text to describe your tables and columns by COMMENT command.

Ex 1: To add a comment to the table EMP.

```
COMMENT ON TABLE EMP IS "Employee Details";
```

Ex2: To add comment on a column

```
COMMENT ON COLUMN EMP.EMPNO IS "UNIQUE EMPLOYEE NUMBER";
```

To see the comments, select the comments column from one of the data dictionary views

ALL_COL_COMMENTS or USER_COL_COMMENTS.

TYPES OF CONSTRAINTS:

Constraints are classified into two types:

- Table Constraints
- Column Constraints

Table Constraint:

A constraint given at the table level is called as table constraint. It may refer to more than one column of the table.

A typical example is PRIMARY KEY constraint that is used to define composite primary key.

Column Constraint:

A constraint given at the column level is called as column constraint. It defines a rule for a single column. It cannot refer to column other than the column, at which it is defined.

A typical example is PRIMARY KEY constraint when a single column is the primary key of the table.

VARIOUS TYPES OF INTEGRITY CONSTRAINTS:

- | | | |
|----------------|---|-----------------------------------|
| 1. Primary Key | } | Key Integrity Constraints |
| 2. Unique Key | | |
| 3. Not NULL | } | Domain Integrity Constraints |
| 4. CHECK | | |
| 5. Foreign Key | } | Referential Integrity Constraints |

1. PRIMARY KEY:

The purpose of primary key is to ensure that information in the column is unique and MUST be entered some data.

UNIQUE+ NOT NULL = PRIMARY KEY

It is used to uniquely identify the rows in a table. There can be only one primary key in a table. It may consist of more than one column, if so, it is called as composite primary key. It maintains uniqueness in the data and null values are not acceptable.

Features of Primary Key:

- I. Primary key will not allow duplicate values.
- II. It will not allow null values.
- III. Only one primary key is allowed per table.

Syntax:

Column level:

[CONSTRAINT const-name] PRIMARY KEY

Example:

```
CREATE TABLE emp (empnoNUMBER(5) CONSTRAINT pemp PRIMARY KEY, ename  
VARCHAR2(20));
```

2. UNIQUE KEY:

The purpose of a unique key is to ensure that information in the column is UNIQUE.

Features of Unique Key:

- I. It does not allow duplicate values.
- II. It allows NULL values.

Syntax:

Column level:

[CONSTRAINT const-name] UNIQUE

Table level:

[CONSTRAINT const-name] UNIQUE (column1, column2,)

Example:

```
CREATE TABLE dept (deptnoNUMBER(5), loc VARCHAR2(20), CONSTRAINT udept
UNIQUE(deptno,loc));
```

3. NOT NULL:

Uniqueness not maintained and null values are not acceptable.

- To satisfy a NOT NULL constraint, every row in the table must contain a value for the column.
- The default is not specified as NULL.

Example:

```
CREATE TABLE griet (snoNUMBER(8) CONSTRAINT psno_nn NOT NULL, sname
VARCHAR2(20) CONSTRAINT sname_nn NOT NULL, branch VARCHAR2(20) branch_nn NOT NULL);
```

4. CHECK:

Defines the condition that should be satisfied before inserting and updating are done.

Syntax:

[CONSTRAINT const-name] CHECK(condition)

Example:

```
CONSTRAINTSal_chk CHECK (sal<=2500)
```

5. REFERENTIAL INTEGRITY CONSTRAINTS:

This constraint is useful for maintaining relation with other table.

Various referential integrity constraints we can use in Oracle are:

- References
- On Delete Cascade

REFERENCES:

It is mainly useful for creating reference key or a foreign key. Reference is always given only in the key fields of other tables. A table can have any number of references. Reference key accepts NULL and duplicate values.

Features of foreign key:

- i. Records cannot be inserted into a detail table if corresponding records in the master table do not exist.
- ii. Records of the master table cannot be deleted or updated if corresponding records in the detail table actually exist.

ON DELETE CASCADE:

It is used along with references. When we delete master table record, it allows us to delete child table records that refer to the deleted master table record.

It deletes rows from child table as and when row from parent table is deleted.

Syntax for foreign key constraint:

Column level:

[CONSTRAINT const-name] REFERENCES TABLE (column)

Table level:

[CONSTRAINT const-name] FOREIGN KEY (col1,col2,...) REFERENCES
(TABLE1(col1),TABLE2(col2),....)

Example:

CONSTRAINT fk_dept FOREIGN KEY(deptno) REFERENCES dept(deptno) ON DELETE
CASCADE

Example:

CREATE TABLE T1 (empnoNUMBER(4) PRIMARY KEY, ename VARCHAR2(20));
CREATE TABLE T2(empnoNUMBER(4),sal NUMBER(7,2), CONSTRAINT odc FOREIGN
KEY(empno) REFERENCES T1(empno) ON DELETE CASCADE);

TASK – 1

DDL COMMANDS (Create, Alter, Drop, Truncate)

1. Create a table EMP with the following structure.

<u>Name</u>	<u>Type</u>
EMPNO	NUMBER(6)
ENAME	VARCHAR2(20)
JOB	VARCHAR2(10)
MGR	NUMBER(4)
DEPTNO	NUMBER(3)
SAL	NUMBER(7,2)

Query:

```
SQL>create table emp(empno number(6), ename varchar2(20), jobvarchar2(10), mgr  
number(4), deptno number(3), sal number(7,2));
```

Table created.

Output:

```
SQL> desc emp;
```

Name	Null?	Type
EMPNO		NUMBER(6)
ENAME		VARCHAR2(20)
JOB		VARCHAR2(10)
MGR		NUMBER(4)
DEPTNO		NUMBER(3)
SAL		NUMBER(7,2)

2. Add a column commission to the EMP table. Commission should be numeric with null values allowed.

Query:

```
SQL>Alter table empadd(commission number(4));
```

Output:

Table altered.

```
SQL> desc emp
```

Name	Null?	Type
EMPNO		NUMBER(6)
ENAME		VARCHAR2(20)
JOB		VARCHAR2(10)
MGR		NUMBER(4)
DEPTNO		NUMBER(3)
SAL		NUMBER(7,2)
COMMISSION		NUMBER(6)

3. Modify the column width of the job field of emp table.

Query:

SQL>Alter table emp modify(job varchar2(15));

Output:

Table altered.

```
SQL> desc emp
Name                          Null?    Type
-----
EMPNO                          NUMBER(6)
ENAME                          VARCHAR2(20)
JOB                             VARCHAR2(15)
MGR                             NUMBER(4)
DEPTNO                          NUMBER(3)
SAL                             NUMBER(7,2)
COMMISSION                      NUMBER(6)
```

4. Create dept table with the following structure.

<u>Name</u>	<u>Type</u>
DEPTNO	NUMBER(2)
DNAME	VARCHAR2(10)
LOC	VARCHAR2(10)

Query:

SQL>create table dept(deptno number(2), dname varchar2(10), loc varchar2(10));

Output:

Table created.

```
SQL> desc dept
Name                          Null?    Type
-----
DEPTNO                          NUMBER(2)
DNAME                          VARCHAR2(10)
LOC                             VARCHAR2(10)
```

5. Add constraint to the emp table that is empno as primary key and deptno as foreign key.

Query:

SQL>alter table emp add constraint emp_id_pk primary key(empno);

SQL>alter table dept add constraint pk primary key(deptno);

Output:

Table altered

```
SQL> alter table dept add constraint pk primary key(deptno);
Table altered.
```

SQL>Alter table emp add constraint emp_deptno_fk foreign key(deptno) references dept(deptno);

```
SQL> alter table emp add constraint emp_id_pk primary key(empno);
Table altered.
SQL> alter table emp add constraint emp_deptno foreign key(deptno) references de
pt(deptno);
Table altered.
```

6. Add constraints to the emp table to check the empno value while entering i.e empno>100.
Salary value by default is 5000, otherwise it should accept the values from the user.

Query:

```
SQL> alter table emp add check (empno>100);
```

```
SQL> alter table emp modify sal default 5000;
```

Output:

```
SQL> alter table emp add check(empno>100);
Table altered.
```

```
SQL> alter table emp modify sal default 5000;
Table altered.
```

7. Add column DOB to the emp table Add and drop a column DOJ to the emp table.

Query:

```
SQL> alter table emp add(dob date);
```

```
SQL> alter table emp add(doj date);
```

```
SQL> alter table emp drop(doj);
```

Output:

```
SQL> alter table emp add(dob date);
Table altered.
```

```
SQL> alter table emp add(doj date);
Table altered.
```

```
SQL> alter table emp drop(doj);
Table altered.
```

EXERCISE 1

1. Create a dept table with the following details and confirm that the table is created.

Column Name	ID	NAME
Key Type	Primary Key	
Null/Unique		
FK Table		
FK Column		
Data type	NUMBER	VARCHAR2

Length	7	25
--------	---	----

- Populate the dept table from the departments table. Include only those columns that you need.
- Create a emp table with the following details and confirm that the table is created.

Column Name	ID	LAST_NAME	FIRST_NAME	DEPT_ID
Key Type				
Null/Unique				
FK Table				DEPT
FK Column				ID
Data type	NUMBER	VARCHAR2	VARCHAR2	NUMBER
Length	7	25	25	7

- Create Employees2 table based on the structure of employees table. Include only employee_id, first_name, last_name, salary and department_id columns. Name the new columns in the table as id, fname, lname, sal, deptid.
- Alter the employees2 table status to Read Only and try to insert a record.
- Alter the employees2 table status to Read Write status and try to insert a record.
- Drop the employees2 table.

DATA MANIPULATION LANGUAGE

Data Manipulation Language is a core part of SQL. DML statement is executed when data is added, updated or deleted to/from the database. A collection of DML statements that form a logical unit of work is called a transaction.

It comprises of the following statements:

- Insert
- Update
- Delete

4. Merge

1) INSERT

INSERT command is used with a query to select rows from one table and insert them into another.

Inserting through parameter substitution

- Inserting null values
- Inserting data values
- Inserting with a query

While inserting data into tables the following should be taken care of:

- Values must be separated by commas.
- Character data must be enclosed within quotes.
- Column values for data type of column are provided with in single quotes.
ORACLE internally converts the character field to the data type field.
- Null values are given as NULL without any quotes.
- If no data is available for all the columns, then the column list must be included.

For inserting a single record:

Syntax:

```
INSERT INTO <table-name> (col1, col2, ...) VALUES (value1, value2, ...)
```

Example:

```
INSERT INTO EMP VALUES(7369,'SMITH','CLERK',7902, '17-DEC-80',9500,'20');
```

Using the INSERT statement values can be entered interactively. Any number of rows can be inserted one after the other by using the '/' (slash), which executes the last statement.

For inserting number of records:

Syntax:

```
INSERT INTO <table-name> VALUES (&col1, &col2, ...)
```

Or

```
INSERT INTO <table-name> ('field1', 'field2') VALUES ('value1', 'value2');
```

Or

```
INSERT INTO <table-name> ('field1', 'data-field') VALUES('value1',TO_DATE('date', 'DD-MM-YY'));
```

Example:

INSERT INTO EMP

VALUES (&EMPNO, '&ENAME', '&JOB', &MGR, '&HIREDATE', &COMM, &DEPTNO);

You can use an INSERT command to a query to select rows from one table and insert them into another. The query replaces the VALUES clause. Only those columns and rows selected by the query will be inserted.

Syntax:

INSERT INTO <table-name1> (field1, field2, ...) SELECT field1, field2, ... FROM
<table-name2> WHERE <condition>;

2) DELETE:

Rows can be deleted from a table using the DELETE statement. A set of rows can be deleted from the table by specifying a condition in a “WHERE” clause. Specific columns cannot be deleted from a table.

Syntax:

DELETE FROM <table-name> [WHERE <condition>]

Example:

DELETE FROM emp where eno=7368;

3) UPDATE:

Columns in a table are updated using the UPDATE command. Values of a single column or a group of columns can be updated. Updation can be carried out for all the rows in a table or selected rows. Expressions and subqueries can also be used in the UPDATE command.

Syntax:

UPDATE <table-name> SET [<col-name>=<value>, <col-name>=<value>,] [WHERE
<condition>]

Example:

UPDATE emp SET ename='John', mgr=7902 where eno=7362;

4) MERGE: (widely used in data warehousing)

It performs two tasks.

- i) INSERT (When rows not matched)
- ii) UPDATE (when rows matched)

-It follows IF-ELSE-ENDIF Logic.

Syntax:

Merge into table name using table name on (join condition)
when matched then
when unmatched then
Insert table name Values (.....);

Example:

Merge into emp_all a using emp-company b on (a.ename=b.ename)
when matched then update set a.salary=b.salary+100
when not matched then insert (ename, salary, jdate)

TASK – 2**DML COMMANDS (Insert, Select, Update, Delete)**

1. Insert 5 records into dept table. Insert few rows and truncate those from emp1 table and also drop it. .

Query:

SQL> Insert into dept values(&deptno, '&dname', '&loc');
SQL> create table emp1 as select * from emp;

SQL>insert into emp1 values(7000, 'King', 'Pres', 10, 20,10000,500, '12-Jan-92');

SQL>insert into emp1 values(7010, 'Jack', 'VP', 10, 30, 9000, 300, '19-Jul-92');

SQL>Truncate table emp1;

SQL>Drop table emp1;

Output:

```
SQL> insert into dept values(&deptno, '&dname', '&loc');
Enter value for deptno: 10
Enter value for dname: Executive
Enter value for loc: USA
1 row created.

SQL> /
Enter value for deptno: 20
Enter value for dname: Marketing
Enter value for loc: UK
1 row created.

SQL> /
SQL> /
Enter value for deptno: 30
Enter value for dname: Production
Enter value for loc: INDIA
1 row created.

SQL> /
Enter value for deptno: 33
Enter value for dname: Despatch
Enter value for loc: SL
1 row created.

SQL> /
Enter value for deptno: 40
Enter value for dname: Packaging
Enter value for loc: JAPAN
1 row created.
```

```
SQL> create table emp1 as select * from emp;
Table created.

SQL> insert into emp1 values(7000, 'King', 'Pres', 10, 20, 10000, 500, '12-Jan-92');
1 row created.

SQL> insert into emp1 values(7010, 'Jack', 'VP', 10, 30, 9000, 300, '19-Jul-92');
1 row created.

SQL> truncate table emp1;
Table truncated.

SQL> drop table emp1;
Table dropped.

SQL>
```

2. Insert 11 records into the emp table.

Query:

SQL>insert into emp values(&no, '&name', '&job', &mgr, &deptno, &sal, &comm, '&dob');

Note: Repeat execution of this statement for 11 times for 11 record insertions

Output:

```
SQL> select * from emp;
```

EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7000 500	King 12-JAN-88	President	7500	20	10000
7200 200	Whalen 04-FEB-91	Supervisor	7580	10	8000
7500 1000	OConnell 07-JUL-89	Manager		30	9000
EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7580 1000	Jane 09-DEC-91	SWManager		10	8000
7599 300	Mary 13-FEB-89	Advisor	7500	33	9000
7600	Birch 26-JAN-94	Clerk	7800	20	6000
EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7650 300	SPaul 19-SEP-89	GM	7580	10	10000
7680	Kochhar 15-AUG-92	AsstHead	7850	10	10000
7850 1000	Hartstein 13-AUG-90	Manager		20	5000
EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7700 300	Russell 29-JAN-93	Clerk	7800	10	9000
7800 1000	Grant 18-NOV-91	ExeManager		33	9000

11 rows selected.

3. Update the emp table to set the default commission of all employees to Rs.1000 /- who are working as managers.

Query:

SQL>update emp set commission=1000 where job like '%Manager%';

Output:

```
SQL> update emp set commission=1000 where job like '%Manager%';  
4 rows updated.
```

4. Delete only those who are working as Supervisors.

Query:

SQL>delete from employee where job like '%Supervisor';

Output:

```
SQL> delete from employee where job like '%Supervisor';  
1 row deleted.
```

5. Delete the rows whose empno is 7599.

Query:

SQL>delete from employee where empno=7599;

Output:

```
SQL> delete from employee where empno=7599;  
1 row deleted.
```

EXERCISE 2

1. Create an insert statement to add the first row of the following data to emp table created in exercise 1.

ID	LAST_NAME	FIRST_NAME	DEPT_ID
1	Patel	Ralph	20
2	Dancs	Betty	20
3	Biri	Ben	30
4	Newman	Chad	10
5	Ropeburn	Audrey	40

2. Populate the emp table with the second row from the above table. List the columns explicitly in the Insert clause. Confirm the additions.
3. Use dynamic reusable Insert statement to add the remaining rows to emp table.
4. Change the last name of employee 3 to 'Drexler' in emp table.
5. Change the department to 10 for the employees working in dept 20.
6. Delete BettyDancs from the emp table.
7. Confirm the changes made.

DATA RETRIEVAL LANGUAGE

SELECT:

This command is used to select data from the tables located in a data base.

Syntax:

```
SELECT attribute1, attribute2..... attributen FROM relation1,  
relation2.....relationn;
```

Syntax: SELECT [DISTINCT] {*,column {alias}.....}
FROM table
WHERE condition(s)
ORDER BY {column, expr} [ASC/DESC];

Example:

```
SELECT ename.job.sal from emp where sal<1000 orders by saldesc;
```

IMPORTANT CLAUSES USED IN SQL (Basic Clauses):

Syntax:

```
SELECT column(s) FROM table(s) WHERE condition(s) GROUP BY column(s)  
HAVING condition(s) using group functions ORDER BY column(s);
```

SELECT:

To retrieve data from one or more tables, views or snap shots.

FROM:

To retrieve the columns which are specified in SELECT clause from one or more tables, views or snap shots.

OPTIONAL CLAUSES:

WHERE:

Restricts the rows selected to those for which the condition is TRUE. If you omit this clause, ORACLE returns all rows from the tables, views or snap shots in the FROM clause.

DISTINCT:

Returns only one copy of each set of duplicate rows selected. Duplicate rows are those with matching values for each expression in the select list.

Example:

```
SQL> Select DISTINCT (job) from emp;
```

GROUP BY:

Groups the selected rows based on the value of expression for each row and return a single row of summary information for each group.

HAVING:

Restricts the groups of rows returned to those groups for which the specified condition is TRUE. If you omit this clause, ORACLE returns summary rows for all groups.

Example:

Select deptno,avg(sal) from emp2 group by deptno having count(*)>5;

Select job,avg(sal) from emp2 group by job having avg(sal)>(select avg(sal) from emp2 where deptno=20);

Note:

The 'WHERE' clause cannot be used to restrict the groups that are returned. It can be used only to restrict the individual rows. To exclude/ include groups, we use 'HAVING' clause.

ORDER BY:

Keeps the rows in order that are returned by the statement.

Syntax:

Order by <ASC> or <DESC>

(Specifies either ascending or descending order. ASC is the default).

The where clause of the select command allows to define a predicate, a condition that can be either true or false for any row of the table.

ARITHMETIC OPERATORS

+ - * /

SQL> select ename, sal * 12 from emp;

SQL> select ename, sal + 100 * 12 from emp;

SQL> select ename, (sal + 100) * 12 from emp;

SQL> select ename, hiredate, (sysdate - hiredate) from emp where (sysdate - hiredate) > (15 * 365);

COLUMN ALIASES

When displaying the result of a query, SQL*Plus uses the selected column's name as the heading. A Column Alias gives a column an alternative heading in output. By default, alias heading will be forced to uppercase and cannot contain blank spaces, unless the alias is enclosed in double quotes (" ").

Example:

```
SQL> select ename Name, sal*12 AnnualSalary from emp;
SQL> select ename "E_Name", sal*12 "Annual Salary" from emp;
SQL> select ename, sal, sal * 0.4 "DA", sal * 0.2 "HRA", sal * 0.1 "PF",
sal + (sal * 0.4) + (sal * 0.2) - (sal * 0.1) "NET" from emp
```

CONCATENATION OPERATOR (||) & LITERALS

It allows columns to be linked to other columns to create a character expression. And a literal is any value included in SELECT list which is not a column name or a column alias.

```
SQL> select empno || ename Employee from emp;
SQL> select empno || '- ' || ename Employee, 'Works in Department', deptno from emp;
```

SINGLE AMPERSAND SUBSTITUTION VARIABLE

A substitution variable is prefixed by a single ampersand (&).

```
SQL> select empno, ename, sal from emp where deptno = &department_no;
Enter value for department_no : 10
```

Displays the information of employees in deptno 10

Single ampersand substitution variable prompts every time the statement is executed to enter a value.

DOUBLE AMPERSAND SUBSTITUTION VARIABLE

If a variable is prefixed with a (&&), SQL*Plus will only prompt for the value once and uses it whenever SQL command is executed.

```
SQL> select empno, ename, sal from emp where job = '&&designation';
```

DEFINE COMMAND

Even using define a value can be assigned to a variable.

```
SQL> DEFINE NAME='SMITH'
SQL> select * from emp where ename = '&name';
SQL> UNDEFINE NAME
```

For undefining the variable.

LOGICAL OPERATORS

AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND is TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

AND Syntax

```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition1 AND condition2 AND condition3 ...;
```

OR Syntax

```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition1 OR condition2 OR condition3 ...;
```

NOT Syntax

```
SELECT column1, column2, ...  
FROM table_name  
WHERE NOT condition;
```

Examples:

```
SQL> select * from employees where deptno=10 AND deptno=20;
```

```
SQL> select empid from employees where deptno=10 OR deptno=20;
```

```
SQL> select * from employees where not deptno=10;
```

LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

LIKE Operator	Description
Syntax WHERE CustomerName LIKE SELECT column1, column2, ...FROM table_nameWHERE columnN LIKE pattern; 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that start with "a" and are at least 3 characters in length

WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"
---------------------------------------	--

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

Example:

SQL> select empname from employees where empname like 'A%';

IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

Syntax

SELECT *column_name(s)*

FROM *table_name*

WHERE *column_name* IN (*value1, value2, ...*);

The following SQL statement selects all customers that are located in "Germany", "France" and "UK":

Example:

SELECT * FROM Customers

WHERE Country IN ('Germany', 'France', 'UK');

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

Example:

SELECT * FROM Customers

WHERE Country NOT IN ('Germany', 'France', 'UK');

BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive: begin and end values are included.

Syntax

SELECT *column_name(s)*

FROM *table_name*

WHERE *column_name* BETWEEN *value1* AND *value2*;

The following SQL statement selects all products with a price BETWEEN 10 and 20:

Example:

SELECT * FROM Products

WHERE Price BETWEEN 10 AND 20;

TASK – 3

SQL Operators

1. List the records in the emp table order by salary in descending order.

Query:

SQL>select * from emp order by saldesc;

Output:

```
SQL> select * from emp order by sal desc;
```

EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7000 500	King 12-JAN-88	President	7500	20	10000
7680	Kochhar 15-AUG-92	AsstHead	7850	10	10000
7650 300	SPaul 19-SEP-89	GM	7580	10	10000
EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7800 1000	Grant 18-NOV-91	ExeManager		33	9000
7700 300	Russell 29-JAN-93	Clerk	7800	10	9000
7500 1000	OConnell 07-JUL-89	Manager		30	9000
EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7599 300	Mary 13-FEB-89	Advisor	7500	33	9000
7200 200	Whalen 04-FEB-91	Supervisor	7580	10	8000
7580 1000	Jane 09-DEC-91	SWManager		10	8000
EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7600	Birch 26-JAN-94	Clerk	7800	20	6000
7850 1000	Hartstein 13-AUG-90	Manager		20	5000

11 rows selected.

2. Display only those employees whose deptno is 30.

Query:

```
SQL>select * from emp where deptno=30;
```

Output:

```
SQL> select * from emp where deptno=30;
```

EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
COMMISSION	DOB				
7500 1000	OConnell 07-JUL-89	Manager		30	9000

3. Display deptno from the table employee avoiding the duplicate values.

Query:

SQL>select distinct deptno from emp;

Output:

```
SQL> select distinct deptno from emp;

DEPTNO
-----
30
20
33
10
```

4. List all employee names, salary and 15% rise in salary. Label the column as New Sal.

Query:

SQL>select ename, sal, (sal*1.15) "New Sal" from emp;

Output:

```
SQL> select ename, sal, sal*1.15 "New Sal" from emp;

ENAME                SAL      New Sal
-----
King                 10000      11500
Whalen                8000       9200
OConnell             9000      10350
Jane                 8000       9200
Mary                 9000      10350
Birch                6000       6900
SPaul               10000      11500
Kochhar              10000      11500
Hartstein            5000       5750
Russell              9000      10350
Grant                9000      10350
```

5. Display the rows whose empno ranges from 7500 to 7600.

Query:

SQL>select empno, ename, sal from emp where empno between 7500 and 7600;

Output:

```
SQL> select empno, ename, sal from emp where empno between 7500 and 7600;

EMPNO ENAME                SAL
-----
7500 OConnell             9000
7580 Jane                 8000
7599 Mary                 9000
7600 Birch                6000
```

6. Display all the employees in dept 10 and 20 in alphabetical order of names.

Query:

SQL>select empno, ename, deptno from emp where deptno in(10,20) group by ename;

Output:

```
SQL> select empno, ename, deptno from emp where deptno in (10,20) order by ename;
```

EMPNO	ENAME	DEPTNO
7600	Birch	20
7850	Hartstein	20
7580	Jane	10
7000	King	20
7680	Kochhar	10
7700	Russell	10
7650	SPaul	10
7200	Whalen	10

7. List the employee names who do not earn commission.

Query:

```
SQL>select empno, ename, sal from emp where commission is null;
```

Output:

```
SQL> select ename from emp where commission is null;
```

ENAME
Birch
Kochhar

8. Display all the details of the records with 5 character names with 'S' as starting character.

Query:

```
SQL>select * from employees where length(last_name)=5 and last_name like 's%';
```

```
SQL> select * from employees where length(last_name)=5 and last_name like 'S%';
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000
171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400
157	Patrick	Sully	PSULLY	011.44.1345.929268	04-MAR-96	SA_REP	9500

9. Display joining date of all employees in the year of 1998.

Query:

SQL>select employee_id ,hire_date from employees where hire_date between '1-jan-1998' and '31-dec-1998';

```
SQL> select sysdate from dual
2 ;

SYSDATE
-----
10-FEB-19

SQL> select employee_id , hire_date from employees where hire_date between '1-jan-1998' and '31-dec-1998';

EMPLOYEE_ID HIRE_DATE
-----
106 05-FEB-98
112 07-MAR-98
118 15-NOV-98
126 28-SEP-98
134 26-AUG-98
139 12-FEB-98
140 06-APR-98
143 15-MAR-98
144 09-JUL-98
153 30-MAR-98
154 09-DEC-98

EMPLOYEE_ID HIRE_DATE
-----
161 03-NOV-98
169 23-MAR-98
170 24-JAN-98
176 24-MAR-98
177 23-APR-98
180 24-JAN-98
181 23-FEB-98
186 24-JUN-98
190 11-JUL-98
194 01-JUL-98
196 24-APR-98

EMPLOYEE_ID HIRE_DATE
-----
197 23-MAY-98

23 rows selected.
```

10. List out the employee names whose salary is greater than 5000 and greater than 6000.

Query:

SQL>select ename from emp where sal>5000 and sal>6000;

Output:

```
SQL> select ename from emp where sal>5000 and sal>6000;

ENAME
-----
King
Whalen
OConnell
Jane
Mary
SPaul
Kochhar
Russell
Grant
```

EXERCISE 3

1. Select all the information from the emp table.
2. Display ename, job and hiredate from employee table
3. Select all employees who have a salary between 1000 and 2000
4. List department numbers and names in department name order.
5. List the details of the employees in department 10 and 20 in alphabetical order of names.
6. List names and jobs of all clerks in department no. 20.
7. Display all employees whose names start with letter 'S'.
8. Display all employees whose names end with letter 'T'.
9. Display all employees names which have TH or LL in them.
10. List the details of all employees who have a manager.
11. Display all employees who were hired during 1983.
12. Display name and remuneration of all employees.
13. Display annual salary and commission of all sales people whose monthly salary is greater than their commission. The output should be ordered by salary, highest first. If two or more employees have the same salary sort by employees name within the highest salary order.
14. List all employees who have name exactly 4 characters in length.
15. List all employees who have no manager.
16. Find employees whose salary is not between a range 1000 and 2000.
17. Find those employees whose job does not start with M.
18. To find clerks who earn between 2000 and 4000.
19. Write a SQL statement to return all managers with salaries over 1500, and all salesmen.
20. Display all different job types.

SQL- AGGREGATE OPERATORS

An aggregate function allows you to perform a calculation on a set of values to return a single scalar value. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

The following are the most commonly used SQL aggregate functions:

- AVG – calculates the average of a set of values.
- COUNT – counts rows in a specified table or view.
- MIN – gets the minimum value in a set of values.
- MAX – gets the maximum value in a set of values.
- SUM – calculates the sum of values.

Notice that all aggregate functions above ignore NULL values except for the COUNT function.

SYNTAX

SELECT MIN(*column_name*) FROM *table_name* WHERE *condition*;

SYNTAX

SELECT MAX(*column_name*) FROM *table_name* WHERE *condition*;

SYNTAX

SELECT COUNT(*column_name*) FROM *table_name* WHERE *condition*;

SYNTAX

SELECT AVG(*column_name*) FROM *table_name* WHERE *condition*;

SYNTAX

SELECT SUM(*column_name*) FROM *table_name* WHERE *condition*;

Example:

1. Display the lowest paid salary from employees

SQL>Select min(salary) from emp;

2. Display the recently joined employee's hiredate

SQL>select max(hire_date) from emp;

3. Display the total number of distinct departments from employees table.

SQL> select count(distinct deptno) from emp;

4. Display the total amount paid to employees.

```
SQL> select sum(salary) from emp;
```

5. Display the average salary paid.

```
SQL> select avg(salary) from emp;
```

GROUP BY:

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example:

1. Display the number of employees in each department.

```
SQL> select count(empid) from employees group by deptno;
```

2. Display the number of employees in each department, sorted high to low.

```
SQL> select count(empid) from employees group by deptno order by count(empid) desc;
```

HAVING CLAUSE:

The having clause is used to restrict the grouped records. The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Example:

Write an SQL statement lists the number of employees in each department. Only include departments with more than 5 employees:

```
SELECT COUNT(empid), deptid FROM emp GROUP BY deptid
HAVING COUNT(empid) > 5;
```

TASK – 4

SQL Aggregate Functions, Group By clause, Having clause

1. Count the total records in the emp table.

Query: select count(*) from emp;

Output:

```
SQL> select count(*) from emp;
COUNT(*)
-----
      11
```

2. Calculate the total and average salary of the employees.

Query: select sum(sal) "Total", avg(sal) "Average" from emp;

Output:

```
SQL> select sum(sal) "Total", avg(sal) "Average" from emp;
      Total      Average
-----
    93000 8454.54545
```

3. Determine the maximum and minimum salary of the employees and rename the columns max_salary and min_salary.

Query: select max(sal) "max_salary", min(sal) "min_salary" from emp;

Output:

```
SQL> select max(sal) "max_salary", min(sal) "min_salary" from emp;
max_salary min_salary
-----
    10000      5000
```

4. Find the no.of departments in employee table.

Query: select deptno, count(deptno) from emp group by deptno;

Output:

```
SQL> select deptno, count(deptno) from emp group by deptno;
DEPTNO COUNT(DEPTNO)
-----
      30              1
      20              3
      33              2
      10              5
```

5. Display job wise sum, avg, max, min salaries.

Query: select job, sum(sal), avg(sal), max(sal), min(sal) from emp group by job;

Output:

```
SQL> select job, sum(sal), avg(sal), max(sal), min(sal) from emp group by job;
```

JOB	SUM(SAL)	AUG(SAL)	MAX(SAL)	MIN(SAL)
Manager	14000	7000	9000	5000
Advisor	9000	9000	9000	9000
Clerk	15000	7500	9000	6000
Supervisor	8000	8000	8000	8000
President	10000	10000	10000	10000
ExeManager	9000	9000	9000	9000
AsstHead	10000	10000	10000	10000
SWManager	8000	8000	8000	8000
GM	10000	10000	10000	10000

6. Display maximum salaries of all departments having maximum salary>2000.

Query: select deptno, max(sal) from emp group by deptno having max(sal)>2000;

Output:

```
SQL> select deptno, max(sal) from emp group by deptno having max(sal)>2000;
```

DEPTNO	MAX(SAL)
30	9000
20	10000
33	9000
10	10000

7. Display job wise sum, avg, max and min salaries in department 10 having average salary>1000 and result is ordered by sum of salary in desc order.

Query: select job, sum(sal), avg(sal), max(sal), min(sal) from emp where deptno=10 group by job having avg(sal)>1000 order by sum(sal) desc;

Output:

```
SQL> select job, sum(sal), avg(sal), max(sal), min(sal) from emp where deptno=10 group by job having avg(sal)>1000 order by sum(sal) desc;
```

JOB	SUM(SAL)	AUG(SAL)	MAX(SAL)	MIN(SAL)
AsstHead	10000	10000	10000	10000
GM	10000	10000	10000	10000
Clerk	9000	9000	9000	9000
SWManager	8000	8000	8000	8000
Supervisor	8000	8000	8000	8000

EXERCISE 4

1. Find the minimum salary of all employees.
2. List the minimum and maximum salary for each job type.
3. Find the minimum, maximum and Average salaries of all employees.
4. Find the average salary and average total remuneration for each job type.
5. Find out the difference of highest and lowest salaries.
6. Select data as displayed who, what and when smith has held the position of clerk in dept no. 20 since 17-dec-80. Allen has held the position of clerk in dept no. 20 since 15-jan-80.
7. Show only employees on Grade 3.
8. Display all employee names, ename of first four characters of Employee table (only on ename field).
9. List the employees who joined in the month of December.
10. Find the sum of ASCII values of 'A' and 'a'.
11. Change all employee names and let their names to lower case whose name start with letter 'A'.
12. Find the time of joining of all the employees from employee table.
13. Display the employees monthly, daily, hourly, salaries from emp table.
14. Find the service of years of all the employees.
15. Select all employee names and their commissions whose dept is '30'. It should display 'Not applicable' whose commission is Null.
16. List lower paid employees working for each manager excluding any group where the Min sal is < 1000 sort the output by salary.
17. Count the no. of times a letter 'S' occurs in table (ename field).
18. Display employee names in ascending order according to their salaries in descending order.
19. Find all departments which have more than 3 employees.
20. Check whether all employee numbers are indeed unique.

SQL FUNCTIONS

Single row functions:

They can be categorized as:

- Character functions
- Number functions
- Date functions
- Conversion functions
- Miscellaneous functions

CHARACTER FUNCTIONS

Character functions take character arguments. Some functions return character values, others return numeric values.

CONCAT

It appends two strings and returns the combined string. If either argument is null, CONCAT returns the other argument. If both arguments are null, CONCAT returns a null.

Syntax: Function CONCAT(str1 varchar2, str2 varchar2) return varchar2

```
SQL> select CONCAT( 'Name ',ename ) from emp;
```

```
SQL> select CONCAT( ename, CONCAT( 'is ', job ) ) from emp;
```

INITCAP

Returns the string with the first letter of each word in upper case and all other letters in the lower case. Words are delimited by spaces or non-alphanumeric characters.

Syntax: Function INITCAP(str varchar2) return varchar2

```
SQL> select INITCAP(ename) from emp;
```

INSTR

Syntax:

Function INSTR(str1 varchar2, str2 varchar2 [, pos number [, n number]])

return number

Searches string str1 starting at character position pos for the nth occurrence of the string str2 and returns the position of the first character of the occurrence. If pos is negative, INSTR searches backwards from the end of str1. The default value of pos & n is 1. If search fails INSTR returns 0.

```
SQL> select ename, INSTR(ename, 'T') from emp;
```

SQL> select ename, INSTR(ename, 'T', 2) from emp;

SQL> select ename, INSTR(ename, 'T', 2, 2) from emp;

LENGTH

It returns the number of characters in string str.

Syntax: Function LENGTH(str char) return char

Function LENGTH(str varchar2) return varchar2

SQL> select ename, LENGTH(ename) from emp;

SQL> select CONCAT(ename, CONCAT(' is ', CONCAT(LENGTH(ename),
' characters long'))) from emp;

LOWER It returns the string str with all letters in the lower case.

Syntax: Function LOWER(str char) return char

Function LOWER(str varchar2) return varchar2

SQL> select LOWER(ename), LOWER(job) from emp;

LTRIM

It returns the string str, with the initial characters removed up to the first character not in the set. If the set is not specified, it defaults to a single blank.

Syntax: Function LTRIM(str varchar2 [, set varchar2]) return varchar2

SQL> select LTRIM(ename, 'S') from emp;

REPLACE It returns the string str1, with every occurrence of string str2 replaced by string str3. If string str3 is not specified all occurrences of str2 are removed.

Syntax:

Function REPLACE(str1 varchar2, str2 varchar2 [, str3 varchar2]) return varchar2

SQL> select REPLACE(ename, 'J', 'BC') from emp;

SQL> select REPLACE(ename, 'J') from emp;

LPAD

It returns the string left padded to length len with the sequence of characters in the pad replicated as many times as necessary. If pad is not specified, it defaults to a single blank.

Syntax:

Function LPAD(str varchar2, len number [, pad varchar2]) return varchar2

SQL > select LPAD(ename, 15, '.') from emp;

RPAD

It returns the string right padded to length len with the sequence of characters in the pad replicated as many times as necessary. If pad is not specified, it defaults to a single blank.

Syntax:

Function RPAD(str varchar2, len number [, pad varchar2]) return varchar2

SQL > select RPAD(ename, 15, '.') from emp;

RTRIM

It returns the string **str**, with the final characters removed after the last character not in the set.

Syntax: Function RTRIM(str varchar2 [, set varchar2]) return varchar2

SQL> select RTRIM(ename, 'S') from emp;

SUBSTR**Syntax:**

Function SUBSTR(str varchar2, pos number [, len number]) return varchar2

It returns a substring of string **str**, starting at character position **pos** and including **len** characters. If len is not specified, all characters till the end of the string are returned.

SQL> select ename, SUBSTR(ename, 2, 4) from emp;

SQL> select ename, SUBSTR(ename, 2) from emp;

UPPER

It returns string **str**, with all letters in the upper case.

Syntax: Function UPPER(str char) return char

Function UPPER(str varchar2) return varchar2

SQL> select UPPER(ename) from emp;

TRANSLATE

It returns the string str1, with every occurrence of string **str2** translated to corresponding occurrence of string str3.

Syntax:

Function TRANSLATE(str1 varchar2, str2 varchar2, str3 varchar2) return varchar2

SQL> select TRANSLATE(ename, 'JL', 'BC') from emp;

ASCII

It returns the number representation of the first byte of the character string, in the database character set.

Syntax:

Function ASCII(char ch) return number

Function ASCII(varchar2 str) return number

SQL> select ASCII('a'), ASCII(ename), ename from emp;

CHR

It returns the character that has the value equivalent to number **n** in the database character set.

Syntax:

Function CHR(number n) return char

```
SQL> select CHR(65) from dual;
```

NUMBER FUNCTIONS

Number functions take numeric arguments and return numeric values.

ABS

It returns the absolute value of the number n.

Syntax:

ABS (n number) return NUMBER

```
SQL> SELECT ABS(-5) FROM dual ;
```

Output: 5

CEIL

It returns the smallest integer greater than or equal to the number n.

Syntax:

CEIL (n number) return NUMBER

```
SQL> SELECT CEIL(15.7), CEIL(15.3) FROM dual ;
```

Output: 16 16

FLOOR

It returns the largest integer equal to or less than the number n.

Syntax:

FLOOR (n number) return NUMBER

```
SQL> SELECT FLOOR(16.8), FLOOR(16.3) FROM dual ;
```

Output: 16 16

MOD

It returns the remainder of the number m when divided by n. If n is 0, m is returned.

Syntax:

MOD (m number, n number) return NUMBER

```
SQL> SELECT MOD(25,6) FROM dual ;
```

Output: 1

ROUND

It rounds the number m, to n decimal places. If n is not specified, m is rounded to zero decimal places. In case n is negative, m is rounded off to the left of the decimal point.

Syntax:

ROUND (m number [, n number]) return NUMBER

```
SQL> SELECT ROUND(14.68), ROUND(14.6878,2), ROUND(16.675,-1) FROM dual ;
```

Output: 15 14.69 20

SQRT

It returns the square root of the number n, which cannot be negative.

Syntax:

SQRT (n number) return NUMBER

```
SQL> SELECT SQRT(16), SQRT(17.8) FROM dual ;
```

Output: 4 4.2190046

TRUNC

It truncates the number m, to n decimal places. If n is not specified, m is truncated to zero decimal places. In case n is negative, m is truncated off to the left of the decimal point.

Syntax:

TRUNC (m number [, n number]) return NUMBER

```
SQL> SELECT TRUNC(14.68), TRUNC (14.6878,2), TRUNC (16.675,-1) FROM dual ;
```

Output: 14 14.68 10

SIGN

It returns the sign of the value passed for positive 1, negative -1, and for zero 0.

```
SQL> SELECT SIGN(10), SIGN (-10), SIGN (0) FROM dual ;
```

Output: 1 -1 0

COS - It returns Cosine of an angle expressed in radians.

```
SQL> SELECT COS(45) FROM dual ;
```

Output: .52532199

SIN - It returns Sine of an angle expressed in radians.

```
SQL> SELECT SIN(45) FROM dual ;
```

Output: .85090352

TAN - It returns tangent of an angle expressed in radians.

```
SQL> SELECT TAN(180) FROM dual ;
```

Output: 1.3386902

SINH - It returns Hyperbolic Sine of an angle expressed in radians.

```
SQL> SELECT SINH(135) FROM dual ;
```

Output: 2.132E+58

COSH - It returns Hyperbolic Cosine of an angle expressed in radians.

```
SQL> SELECT COSH(30) FROM dual ;
```

Output: 5.343E+12

TANH - It returns Hyperbolic Tangent of an angle expressed in radians.

```
SQL> SELECT TANH(30) FROM dual ;
```

Output: 1

LN - It returns natural logarithm of the number.

```
SQL> SELECT LN(5) FROM dual ;
```

Output: 1.6094379

LOG - It returns logarithm of any number for which base is also specified within the function.

```
SQL> SELECT LOG(15,10) FROM dual ;
```

Output: .85027415

POWER - It returns the value after first number raised to the power of the second number.

```
SQL> SELECT POWER(2,10) FROM dual ;
```

Output: 1024

EXP - This function returns e raised to the nth power.

```
SQL> SELECT EXP(4) FROM dual ;
```

Output: 54.59815

DATE FUNCTIONS

Date functions take arguments of date type.

YYYY - year in four digits

YYY - year in three digits

YY - year in last 2 digits

YEAR - year spelled out

MM - month in number format

MON - month in 3 letter abbreviation

MONTH - month spelled out

DD - date

DAY	-	name of the day spelled out
DY	-	name of the day in 3 letter abbreviation
HH12	-	hour of day (0-12)
HH24	-	hour of day (0-24)
MI	-	minute
SS	-	seconds

ADD_MONTHS

It returns date **d** plus or minus **num** months.

Syntax:

ADD_MONTHS(d DATE, num NUMBER) return DATE

SQL> select add_months('12-OCT-95',2) from dual;

LAST_DAY

It returns the date of the last day of the month containing date.

Syntax: LAST_DAY(d DATE) return DATE

SQL> select LAST_DAY(sysdate) from dual;

SQL> select sysdate Today, LAST_DAY(sysdate) "Last Day", LAST_DAY(sysdate) – sysdate "Days Left" from dual;

MONTHS_BETWEEN

It returns the no. of months between two dates.

Syntax: MONTHS_BETWEEN(d1 DATE, d2 DATE) return NUMBER

SQL> select sysdate, MONTHS_BETWEEN('20-APR-99', '20-APR-98') from dual;

NEXT_DAY

It returns the date of the first weekday named by char **c** that is later than the date **d**.

Syntax: NEXT_DAY(d DATE, c CHAR) return DATE

SQL> select NEXT_DAY('15-MAY-99', 'FRIDAY') from dual;

ROUND

It rounds the date **d** to the unit specified by the format.

Syntax: ROUND(d DATE [, format]) return DATE

SQL> select ROUND(to_date('25-MAY-99'), 'MM') from dual;

TRUNC

It truncates the date **d** to the unit specified by the format.

Syntax: TRUNC(d DATE [, format]) return DATE

```
SQL> select TRUNC(to_date('25-MAY-99'), 'MM') from dual;
```

SYSDATE

It is pseudo column that returns the current date. You can use SYSDATE just as you use another name

```
SQL> select SYSDATE from dual;
```

CONVERSION FUNCTIONS

TO_CHAR

Converts date **d** to a value of varchar2 datatype in the format specified. If format is not specified, it converts the date to varchar2 with same format.

Syntax: TO_CHAR(d DATE [, format])

```
SQL> select ename, TO_CHAR(hiredate, 'MONTH DD, YYYY') from emp;
```

TO_CHAR

Converts number **n** to a value of varchar2 datatype in the format specified. If format is not specified, it converts the number to varchar2 with same format.

Syntax: TO_CHAR(n NUMBER [, format])

```
SQL> select TO_CHAR(456789, '9,99,999.000') from dual;
```

TO_NUMBER

It converts string **str** from a value of type char to a value of type number in the format specified.

Syntax: TO_NUMBER(str CHAR [, format]) return NUMBER

```
SQL> select TO_NUMBER('67235.33', '99999.99') from dual;
```

TO_DATE

Converts the string **str** to a value of type date in the format specified by the format specified.

Syntax: TO_DATE(str VARCHAR2 [, format]) return DATE

```
SQL> select TO_DATE('25-MAY-99', 'DD-MON-YY') from dual;
```

MISCELLANEOUS FUNCTIONS

DECODE

Syntax: DECODE(expr, search1, result1 [, search2, result2], [default])

```
SQL> select ename, DECODE(deptno, 10, 'SALES', 20, 'FINANCE',
```

'COMPUTERS') from emp;

NVL

If **expr1** is NULL, it returns **expr2** otherwise it returns **expr1**.

Syntax: NVL(expr1, expr2)

SQL> select ename, NVL(comm, 0) from emp;

SQL> select ename, NVL(to_char(comm), 'NOT APPLICABLE') from emp;

TASK - 5

Exercise on SQL Functions

1. Display the employee name concatenated with empno.

Query:

select concat(empno, concat(' ', ename)) from emp;

Output: SQL> select concat(empno, concat(' ', ename)) from emp;

CONCAT(EMPNO, CONCAT(' ', ENAME))

7000 King
7200 Whalen
7500 OConnell
7580 Jane
7599 Mary
7600 Birch
7650 SPaul
7680 Kochhar
7850 Hartstein
7700 Russell
7800 Grant

2. Display half of employee name in upper case and half in lower case.

Query:

Select

upper(substr(ename,0,length(ename)/2))||lower(substr(ename,(length(ename)/2)+1,length(ename))) "Name" from emp;

Output:

```
SQL> select upper(substr(ename,0,length(ename)/2))||lower(substr(ename,(length(ename)/2)+1,length(ename))) "Name" from emp;

Name
-----
KIng
WHAlen
OCOnnell
JAnE
MAry
BIRch
SPaul
KOChhar
HARIsTein
RUSsell
GRant
```

3. Display the month name of date "14-jul-09" in full.

Query:

```
select to_char(to_date('14-jul-09'),'MONTH') "Month" from dual;
```

Output:

```
SQL> select to_char(to_date('14-jul-09'),'MONTH') "Month" from dual;

Month
-----
JULY
```

4. Display the DOB of all employees in the format 'dd-mm-yy'.

Query:

```
select to_char(dob,'dd-mm-yy') from emp;
```

Output:

```
SQL> select to_char(dob,'dd-mm-yy') from emp;

TO_CHAR(
-----
12-01-88
04-02-91
07-07-89
09-12-91
13-02-89
26-01-94
19-09-89
15-08-92
13-08-90
29-01-93
18-11-91
```

5. Display the date two months after the DOB of employees.

Query:

```
select add_months(dob,2) from emp;
```

Output:

```
SQL> select add_months(dob,2) from emp;

ADD_MONTH
-----
12-MAR-88
04-APR-91
07-SEP-89
09-FEB-92
13-APR-89
26-MAR-94
19-NOV-89
15-OCT-92
13-OCT-90
29-MAR-93
18-JAN-92
```

6. Display the last date of that month in “05-Oct-09”.

Query:

select last_day(to_date('05-oct-09')) “Last” from dual;

Output: SQL> select last_day(to_date('05-oct-09')) "Last" from dual;

```
Last
-----
31-OCT-09
```

7. Display the rounded date in the year format, month format, day format in the employee.

Query:

select round(dob, 'dd'), round(dob, 'month'), round(dob, 'year') from emp;

Output:

```
SQL> select round(dob, 'dd'), round(dob, 'month'), round(dob, 'year') from emp;

ROUND(DOB) ROUND(DOB) ROUND(DOB)
-----
12-JAN-88 01-JAN-88 01-JAN-88
04-FEB-91 01-FEB-91 01-JAN-91
07-JUL-89 01-JUL-89 01-JAN-90
09-DEC-91 01-DEC-91 01-JAN-92
13-FEB-89 01-FEB-89 01-JAN-89
26-JAN-94 01-FEB-94 01-JAN-94
19-SEP-89 01-OCT-89 01-JAN-90
15-AUG-92 01-AUG-92 01-JAN-93
13-AUG-90 01-AUG-90 01-JAN-91
29-JAN-93 01-FEB-93 01-JAN-93
18-NOV-91 01-DEC-91 01-JAN-92
```

8. Display the commissions earned by employees. If they do not earn commission, display it as “No Commission”.

Query:

```
select employee_id,last_name,nvl(to_char(commission_pct),'No Commission')"commission"
from employees;
```

```
SQL> select employee_id,last_name ,nvl(to_char(commission_pct),'No Commission')"commission" from employees;
```

EMPLOYEE_ID	LAST_NAME	commission
100	King	No Commission
101	Kochhar	No Commission
102	De Haan	No Commission
103	Hunold	No Commission
104	Ernst	No Commission
105	Austin	No Commission
106	Pataballa	No Commission
107	Lorentz	No Commission
108	Greenberg	No Commission
109	Faviet	No Commission
110	Chen	No Commission

EMPLOYEE_ID	LAST_NAME	commission
111	Sciarra	No Commission
112	Urman	No Commission
113	Popp	No Commission
114	Raphaely	No Commission
115	Khoo	No Commission
116	Baida	No Commission
117	Tobias	No Commission
118	Himuro	No Commission
119	Colmenares	No Commission
120	Weiss	No Commission
121	Fripp	No Commission

EXERCISE 5

1. Write a query to display the system date.
2. Display the employee number, last name, salary, and salary increased by 15.5% (expressed as a whole number) for each employee. Label the column "New Salary".
3. Write a query that displays the last name (with the first letter in upper case and all the other letters in lower case) and the length of the last name for all employees whose name starts with the letters "J", "A", or "M". Give each column an appropriate label. Sort the results by the employees' last name.
4. Display the last name and calculate the number of months between today and the date on which the employee was hired. Label the column MONTHS_WORKED. Order the results by the number of months employed. Round the number of months up to the closest whole number.
5. Create a query to display the last name and salary for all employees. Format the salary to be 15 characters long, left padded with the \$ symbol. Label the column SALARY.
6. Create that displays the first eight characters of the employees' last names and indicates the amounts of their salaries with asterisk. Each asterisk signifies a thousand dollars. Sort the data in descending order of salary. Label the column EMPLOYEES_AND_THEIR_SALARIES.
7. Create a report that displays each employee's last name, hire date and salary review date, which is the first Monday after 6 months of service. Label the column REVIEW. Format the dates to appear in the format similar to "Monday, the Thirty-First of July, 2000".
8. Display the last name, hire date and day of the week on which the employee started. Label the column DAY. Order the results by the day of the week, starting with Monday.
9. Write a query that displays the grade of all employees based on the value of the JOB_ID column, using the following data:

Job	Grade
AD_PRES	A
ST_MAN	B
IT_PROG	C
SA_REP	D
ST_CLERK	E

None of the above 0

10. Rewrite the statement in the preceding exercise by using the CASE syntax.

NESTED QUERIES

A Sub query or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. It is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. These can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

Rules that sub queries must follow –

- Sub queries must be enclosed within parentheses.
- A sub query can have only one column in the SELECT clause, unless multiple columns are in the main query for the sub query to compare its selected columns.
- An ORDER BY command cannot be used in a sub query, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a sub query.
- Sub queries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A sub query cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a sub query. However, the BETWEEN operator can be used within the sub query.

Sub Queries with the SELECT Statement

Sub queries are most frequently used with the SELECT statement. The basic syntax is as follows –

Syntax:

```
SELECT column_name [, column_name ] FROM table1 [, table2 ] WHERE column_name  
OPERATOR
```

```
(SELECT column_name [, column_name ] FROM table1 [, table2 ] [WHERE]);
```

Example:

Display the employee details whose salary is same as employee with ID 104.

```
SQL> SELECT * FROM EMPLOYEES WHERE SALARY = (SELECT SALARY FROM  
EMPLOYEES WHERE EMPID=104);
```

Sub queries with the INSERT Statement

Sub queries also can be used with INSERT statements. The INSERT statement uses the data returned from the sub query to insert into another table. The selected data in the sub query can be modified with any of the character, date or number functions.

Syntax:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]  
  SELECT [ *|column1 [, column2 ]  
  FROM table1 [, table2 ]  
  [ WHERE CONDITION ]
```

Example

Consider a table Emp_copy with similar structure as EMP table. Copy the complete EMP table into the Emp_copy table.

```
SQL> INSERT INTO Emp_copy SELECT * FROM EMP WHERE empid IN (SELECT  
empid FROM EMP) ;
```

Sub queries with the UPDATE Statement

The sub query can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a sub query with the UPDATE statement.

Syntax:

```
UPDATE table SET column_name = new_value [ WHERE OPERATOR [ VALUE ]  
(SELECT COLUMN_NAME FROM TABLE_NAME) [ WHERE) ]
```

Example:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the emp table for all the customers whose age is greater than or equal to 27.

```
SQL> UPDATE emp SET SALARY = SALARY * 0.25 WHERE AGE IN (SELECT AGE  
FROM Emp_copy WHERE AGE >= 27 );
```

Sub queries with the DELETE Statement

The sub query can be used in conjunction with the DELETE statement like with any other statements mentioned above.

Syntax:

```
DELETE FROM TABLE_NAME [ WHERE OPERATOR [ VALUE ]  
(SELECT COLUMN_NAME FROM TABLE_NAME) [(WHERE)]
```

Example:

Assuming, we have a Emp_copy table available which is a backup of the Emp table. The following example deletes the records from the Emp table for all the employees whose age is greater than or equal to 27.

```
SQL> DELETE FROM EMP WHERE AGE IN (SELECT AGE FROM Emp_copy WHERE  
AGE >= 27 );
```

TASK – 6

Nested Queries

1. Find the third highest salary of the employees.

Query:

```
select max(sal) from emp where sal< (select max(sal) from emp where sal< (select max(sal) from emp));
```

Output:

```
SQL> select max(sal) from emp where sal<(select max(sal) from emp where sal<(select max(sal) from emp));
```

MAX(SAL)
8000

2. Display all the employee names and salary whose salary is greater than the minimum salary and job title starts with 'M'.

Query:

```
select ename, sal from emp where sal>(select min(sal) from emp) and job like 'M%';
```

Output:

```
SQL> select ename, sal from emp where sal>(select min(sal) from emp) and job like 'M%';
```

ENAME	SAL
McConnell	9000

3. Write a Query to display information about employees who earn more than any employee in department 30.

Query:

```
select empno, ename, sal, deptno from emp where sal>any (select sal from emp where deptno=30);
```

Output:

```
SQL> select empno, ename, sal, deptno from emp where sal>any (select sal from emp where deptno=30);
```

EMPNO	ENAME	SAL	DEPTNO
7000	King	10000	20
7650	SPaul	10000	10
7680	Kochhar	10000	10

4. Display the employees who have the same job as Jones and whose salary>=Fords.

Query:

```
select empno, ename, sal, job from emp where job= (select job from emp where
ename='Jones') and sal>= (select sal from emp where ename='Fords');
```

Output:

```
SQL> select empno, ename, sal, job from emp where job= (select job from emp where ename='Jones') and sal>= (select sal from emp where ename='Fords');
no rows selected
```

5. List out the employee names who get the salary> maximum salary of dept with deptno 20,30.

Query:

```
select ename from emp where sal>(select max(sal) from emp where deptno in(20,30));
```

Output:

```
SQL> select ename from emp where sal>(select max(sal) from emp where deptno in(20,30));
no rows selected
```

6. Display the maximum salaries of the departments whose maximum salary>9000.

Query:

```
select max(sal) from emp group by deptno having max(sal)>9000;
```

Output:

```
SQL> select max(sal) from emp group by deptno having max(sal)>9000;

  MAX(SAL)
-----
    10000
    10000
```

7. Create a table employee with the same structure as the table emp and insert rows into the table using select clauses.

Query:

```
SQL>create table employee as (select * from emp);
```

Output:

```
SQL> create table employee as (select * from emp);
```

```
Table created.
```

```
SQL> select * from employee;
```

EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
7000	King	President	7500	20	10000
7200	Whalen	Supervisor	7580	10	8000
7500	OConnell	Manager		30	9000

```
11 rows selected.
```

8. Create a manager table from the emp table which should hold details only about managers.

Query:

```
SQL>create table manager as (select * from emp where job like '%Manager%');
```

```
SQL> create table manager as (select * from emp where job like '%Manager%');
```

```
Table created.
```

```
SQL> select * from manager
```

EMPNO	ENAME	JOB	MGR	DEPTNO	SAL
7500	OConnell	Manager		30	9000
7580	Jane	SWManager		10	8000
7850	Hartstein	Manager		20	5000
7800	Grant	ExeManager		33	9000

EXERCISE 6

1. Write a query that prompts the user for an employee last name. The query then displays the last name and hire date of any employee in the same department as the employees whose name they supply (exclude that employee).
2. Display the employee number, last name and salary of all employees who earn more than the average salary. Sort the results in order of ascending salary.
3. Display the employee number and last name of all employees who work in a department with any employee whose last name contains a “u”.
4. Display the last name, department number and job ID of all employees whose department location ID is 1700.
5. Display the last name and salary of every employee who reports to King.
6. Display the department number, last name and jobID for every employee in the Executive department.
7. Display the names of all employees whose salary is more than the salary of any employee from department 60.
8. Display the employee number, last name and salary of all employees who earn more than the average salary and who work in a department with any employee whose last name contains a “u”.

JOINS

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables –

Table 1 – CUSTOMERS Table

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Table 2 – ORDERS Table

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Now, join these two tables in the SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT FROM CUSTOMERS, ORDERS WHERE
CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

ID	NAME	AGE	AMOUNT
3	kaushik	23	3000
3	kaushik	23	1500
2	Khilan	25	1560
4	Chaitali	25	2060

The join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL –

- **INNER JOIN** – returns rows when there is a match in both tables.
- **LEFT JOIN** – returns all rows from the left table, even if there are no matches in the right table.
- **RIGHT JOIN** – returns all rows from the right table, even if there are no matches in the left table.
- **FULL JOIN** – returns rows when there is a match in one of the tables.
- **SELF JOIN** – is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- **CARTESIAN JOIN** – returns the Cartesian product of the sets of records from the two or more joined tables.

INNER JOIN:

The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**. The **INNER JOIN** creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax:

```
SELECT table1.column1, table2.column2... FROM table1 INNER JOIN table2 ON
table1.common_field = table2.common_field;
```

Example:

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS JOIN ORDERS ON
CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

LEFT OUTER JOIN:

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table. This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax:

```
SELECT table1.column1, table2.column2... FROM table1 LEFT OUTER JOIN table2
ON table1.common_field = table2.common_field;
```

Example:

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT OUTER JOIN  
ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

RIGHT OUTER JOIN:

The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax:

```
SELECT table1.column1, table2.column2... FROM table1 RIGHT OUTER JOIN table2 ON  
table1.common_field = table2.common_field;
```

Example:

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT OUTER  
JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

FULL OUTER JOIN:

The SQL **FULL JOIN** combines the results of both left and right outer joins.

The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

Syntax

```
SELECT table1.column1, table2.column2... FROM table1 FULL OUTER JOIN table2 ON  
table1.common_field = table2.common_field;
```

Example:

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS FULL OUTER JOIN  
ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

SELF JOIN:

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax

```
SELECT a.column_name, b.column_name... FROM table1 a, table1 b WHERE  
a.common_field = b.common_field;
```

Example:

```
SQL> SELECT a.ID, b.NAME, a.SALARY FROM CUSTOMERS a JOIN CUSTOMERS b  
ON a.SALARY < b.SALARY;
```

CROSS JOIN

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

Syntax:

```
SELECT table1.column1, table2.column2... FROM table1, table2 [, table3 ];
```

Example:

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS, ORDERS;
```

SET OPERATORS

Set operators are used to join the results of two (or more) SELECT statements. The SET operators available in Oracle 11g are UNION, UNION ALL, INTERSECT and MINUS.

The UNION set operator returns the combined results of the two SELECT statements. Essentially, it removes duplicates from the results i.e. only one row will be listed for each duplicated result. To counter this behavior, use the UNION ALL set operator which retains the duplicates in the final result. INTERSECT lists only records that are common to both the SELECT queries; the MINUS set operator removes the second query's results from the output if they are also found in the first query's results. INTERSECT and MINUS set operations produce unduplicated results.

All the SET operators share the same degree of precedence among them. Instead, during query execution, Oracle starts evaluation from left to right or from top to bottom. If explicitly parentheses are used, then the order may differ as parentheses would be given priority over dangling operators.

Points to remember:

- Same number of columns must be selected by all participating SELECT statements. Column names used in the display are taken from the first query.
- Data types of the column list must be compatible/implicitly convertible by Oracle. Oracle will not perform implicit type conversion if corresponding columns in the component queries belong to different data type groups. For example, if a column in the first component query is of data type DATE, and the corresponding column in the second component query is of data type CHAR, Oracle will not perform implicit conversion, but raise ORA-01790 error.
- Positional ordering must be used to sort the result set. Individual result set ordering is not allowed with Set operators. ORDER BY can appear once at the end of the query. For example,
- UNION and INTERSECT operators are commutative, i.e. the order of queries is not important; it doesn't change the final result.
- Performance wise, UNION ALL shows better performance as compared to UNION because resources are not wasted in filtering duplicates and sorting the result set.

- Set operators can be the part of sub queries.
- Set operators can't be used in SELECT statements containing TABLE collection expressions.
- The LONG, BLOB, CLOB, BFILE, VARRAY, or nested table are not permitted to use in Set operators. For update clause is not allowed with the set operators.

UNION

When multiple SELECT queries are joined using UNION operator, Oracle displays the combined result from all the compounded SELECT queries, after removing all duplicates and in sorted order (ascending by default), without ignoring the NULL values.

Example:

```
SELECT 1 NUM FROM DUAL
UNION
SELECT 5 FROM DUAL
UNION
SELECT 3 FROM DUAL
UNION
SELECT 6 FROM DUAL
UNION
SELECT 3 FROM DUAL;
```

Output:

```
NUM
-----
1
3
5
6
```

UNION ALL

UNION and UNION ALL are similar in their functioning with a slight difference. But UNION ALL gives the result set without removing duplication and sorting the data. For example, in above query UNION is replaced by UNION ALL to see the effect.

Consider the query demonstrated in UNION section. Note the difference in the output which is generated without sorting and de-duplication.

Example:

```
SELECT 1 NUM FROM DUAL
UNION ALL
SELECT 5 FROM DUAL
```

```

UNION ALL
SELECT 3 FROM DUAL
UNION ALL
SELECT 6 FROM DUAL
UNION ALL
SELECT 3 FROM DUAL;

```

Output:

```

NUM
-----
1
5
3
6
3

```

INTERSECT

Using INTERSECT operator, Oracle displays the common rows from both the SELECT statements, with no duplicates and data arranged in sorted order (ascending by default).

For example, the below SELECT query retrieves the salary which are common in department 10 and 20. As per ISO SQL Standards, INTERSECT is above others in precedence of evaluation of set operators but this is not still incorporated by Oracle.

Example:

```

SELECT SALARY
FROM employees
WHERE DEPARTMENT_ID = 10
INTRESECT
SELECT SALARY
FROM employees
WHERE DEPARTMENT_ID = 20

```

Output:

```

SALARY
-----
1500
1200
2000
MINUS

```

Minus operator displays the rows which are present in the first query but absent in the second query, with no duplicates and data arranged in ascending order by default.

Example:

```
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 10
MINUS
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 20;
```

Output:

```
JOB_ID
-----
HR
FIN
ADMIN
```

Matching the SELECT statement

There may be the scenarios where the compound SELECT statements may have different count and data type of selected columns. Therefore, to match the column list explicitly, NULL columns are inserted at the missing positions so as match the count and data type of selected columns in each SELECT statement. For number columns, zero can also be substituted to match the type of the columns selected in the query.

In the below query, the data type of employee name (varchar2) and location id (number) do not match. Therefore, execution of the below query would raise error due to compatibility issue.

Output:

```
SELECT DEPARTMENT_ID "Dept", first_name "Employee"
FROM employees
UNION
SELECT DEPARTMENT_ID, LOCATION_ID
FROM departments;
```

Output:

ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression

Explicitly, columns can be matched by substituting NULL for location id and Employee name.

Output:


```
SELECT DEPARTMENT_ID "Dept", first_name "Employee", NULL "Location"
FROM employees
UNION
SELECT DEPARTMENT_ID, NULL "Employee", LOCATION_ID
FROM departments;
```

Using ORDER BY clause in SET operations

The ORDER BY clause can appear only once at the end of the query containing compound SELECT statements. It implies that individual SELECT statements cannot have ORDER BY clause. Additionally, the sorting can be based on the columns which appear in the first SELECT query only. For this reason, it is recommended to sort the compound query using column positions.

The compound query below unifies the results from two departments and sorts by the SALARY column.

Output:

```
SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=10
UNION
SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=20
ORDER BY 3;
```

TASK – 7

Joins, Set Operators

1. Display all the employees and departments implementing left outer join.

Query: select e.empno, e.ename, d.deptno, d.dname from emp e left outer join dept d on(e.deptno=d.deptno);

Output:

```
SQL> select e.empno, e.ename, d.deptno, d.dname from emp e left outer join dept d on(e.deptno=d.deptno);
```

EMPNO	ENAME	DEPTNO	DNAME
7000	King	20	Marketing
7200	Whalen	10	Executive
7500	OConnell	30	Production
7580	Jane	10	Executive
7599	Mary	33	Despatch

2. Display the employee name and department name in which they are working implementing a full outer join.

Query: select e.ename,d.dname from emp e full outer join dept d on(e.deptno=d.deptno);

Output:

```
SQL> select e.ename,d.dname from emp e full outer join dept d on(e.deptno=d.deptno);
```

ENAME	DNAME
Russell	Executive
Kochhar	Executive
SPaul	Executive
Jane	Executive
Whalen	Executive
Hartstein	Marketing
Birch	Marketing
King	Marketing
OConnell	Production
Grant	Despatch
Mary	Despatch
ENAME	DNAME
	Packaging

3. Write a Query to display the employee name and manager's name and salary for all employees.

Query: select e.ename, m.ename "MGR", m.sal "MGRSAL" from emp e, emp m where e.mgr=m.empno;

Output:

```
SQL> select e.ename, m.ename "MGR", m.sal "MGRSAL" from emp e, emp m where e.mgr=m.empno;
```

ENAME	MGR	MGRSAL
King	OConnell	9000
Whalen	Jane	8000
Mary	OConnell	9000
Birch	Grant	9000
SPaul	Jane	8000
Kochhar	Hartstein	5000
Russell	Grant	9000

4. Write a Query to Output the name, job, employee number, department name, location for each department even if there are no employees.

Query: select e.empno, e.ename, e.job, d.dname, d.loc from emp e join dept d on(e.deptno=d.deptno);

Output:

```
SQL> select e.empno, e.ename, e.job, d.dname, d.loc from emp e join dept d on(e.deptno=d.deptno);
```

EMPNO	ENAME	JOB	DNAME	LOC
7000	King	President	Marketing	UK
7200	Whalen	Supervisor	Executive	USA
7500	OConnell	Manager	Production	INDIA

5. Display the details of those who draw the same salary.

Query:select empno, ename, sal from emp where sal=&sal;

Output:

```
SQL> select empno, ename, sal from emp where sal=&sal;
Enter value for sal: 8000
```

EMPNO	ENAME	SAL
7200	Whalen	8000
7580	Jane	8000

EXERCISE 7

1. Write a query to produce the addresses of all the departments. Use the LOCATIONS and COUNTRIES tables. Show the location id, street address, city, state or province and country in the output. Use a NATURAL JOIN to produce the results.
2. Write a query to display the last name, department number and department name for all the employees.
3. Display the last name, job, department number, department name for all employees who work in Toronto.
4. Display employees' last names and employee number along with their managers' last names and manager number. Label the columns Employee, Emp#, Manager, Mgr# respectively.
5. Display all employees including King who has no manager. Order the results by the employee number.
6. Display the employee last names, department numbers and all the employees who work in the same department as a given employee.
7. Display the names of all employees and hire date who were hired after Davies.
8. Find the names and hire dates for all employees who were hired before their managers along with their managers' names and hire dates.
9. List the department ids for departments that do not contain the job id ST_CLERK.
10. Produce a list of jobs for departments 10, 50 and 20 in that order. Display job ID and department ID using the set operators.
11. List the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired by the company.
12. Write a compound query with the following specifications:
 - a) Last name and department id of all the employees from the employees table, regardless of whether or not they belong to a department.
 - b) Department id and department name of all the departments from the departments table, regardless of whether or not they have employees working in them.

VIEWS

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query. A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables that allow users to do the following –

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation.

Syntax:

```
CREATE VIEW view_name AS SELECT column1, column2..... FROM table_name  
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example:

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM  
CUSTOMERS;
```

```
SQL> SELECT * FROM CUSTOMERS_VIEW;
```

The WITH CHECK OPTION:

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition. If they do not satisfy the condition(s), the UPDATE or INSERT returns an error. The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

Example:

```
SQL>CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM  
CUSTOMERS WHERE age IS NOT NULL WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below –

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Bhargavi.

Example:

```
SQL > UPDATE CUSTOMERS_VIEW  
SET AGE = 35  
WHERE name = 'Bhargavi';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself.

Inserting Rows into a View:

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command. Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows from a View:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command. Following is an example to delete a record having AGE = 22.

Example:

```
SQL > DELETE FROM CUSTOMERS_VIEW WHERE age = 22;
```

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself.

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed.

Syntax:

```
DROP VIEW view_name;
```

Example:

```
DROP VIEW CUSTOMERS_VIEW;
```

TASK – 8

Views

1.Create a view that displays the employee id, name and salary of employees who belong to 10th department.

Query:

Create view emp_view as select employee_id,last_name,salary from employees where department_id=10;

```
SQL> create view emp_view as select employee_id,last_name,salary from employees where department_id=10;
View created.
SQL> select * from emp_view;
EMPLOYEE_ID LAST_NAME          SALARY
-----
          200 Whalen              4400
SQL>
```

2.Create a view with read only option that displays the employee name and their department name

Query:

create view emp_dept as select employee_id,last_name,department_id from employees with read onlyh constraint emp_dept_readonly;


```
SQL> create view emp_dept as select employee_id,last_name,department_id from employees with read only constraint emp_dept_readonly;
View created.

SQL> select * from emp_dept
2 ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
105	Austin	60
106	Pataballa	60
107	Lorentz	60
108	Greenberg	100
109	Faviet	100
110	Chen	100
111	Scianna	100
112	Urman	100
113	Popp	100
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30
120	Weiss	50
121	Fripp	50

3. Display all the views generated.

Query:

```
select view_name from user_views;
```

Output:

```
SQL> select view_name from user_views;
VIEW_NAME
-----
DEPT50
EMPLOYEES_VU
EMP_DETAILS_VIEW
MANAGER_VIEW
MY_VIEW
MY_VIEW1
6 rows selected.
```

4. Execute the DML commands on the view created.and drop them.

Query:

- delete from my_view where empno=7900;
- insert into manager_view values(8000, 'Grant', 'ExeHead', null, 10, 19000, 200, '19-dec-90');
- update manager_view set sal=15000 where sal<11000;

Output:

```
SQL> delete from my_view where empno=7800;  
1 row deleted.  
SQL> insert into manager_view values(8000,'Grant','ExeHead',null,10,19000,200,'19-dec-90');  
1 row created.  
SQL> update manager_view set sal=15000 where sal<11000;  
3 rows updated.
```

Drop a view.

Query: drop view my_view;

Output:

```
SQL> drop view my_view;  
View dropped.
```

EXERCISE 8

1. Create a view called emp_vu based on employee numbers, last names and department numbers from the employees table.
2. Display the contents of the emp_vu view.
3. Using the emp_vu, write a query to display all employee names and department numbers.
4. Create a view named DEPT50 that contains the employee numbers, last names and department numbers for all employees in department 50. Do not allow an employee to be reassigned to another department through the view.
5. Display the structure and contents of the dept50 view.
6. Attempt to reassign Matos to department 80.

DATA CONTROL LANGUAGE

Data Control Language (DCL) is used to control privileges in Database. To perform any operation in the database, such as for creating tables, sequences or views, a user needs privileges.

Privileges are of two types,

System: This includes permissions for creating session, table, etc and all types of other system privileges.

Object: This includes permissions for any command or query to perform any operation on the database tables.

In DCL we have two commands,

GRANT: Used to provide any user access privileges or other privileges for the database.

REVOKE: Used to take back permissions from any user.

Allow a User to create session

When we create a user in SQL, it is not even allowed to login and create a session until and unless proper permissions/privileges are granted to the user.

Following command can be used to grant the session creating privileges.

Syntax:

```
GRANT CREATE SESSION TO username;
```

Allow a User to create table

To allow a user to create tables in the database, we can use the below command,

Syntax:

```
GRANT CREATE TABLE TO username;
```

Grant all privilege to a User

sysdba is a set of privileges which has all the permissions in it. So if we want to provide all the privileges to any user, we can simply grant them the sysdba permission.

Syntax:

```
GRANT sysdba TO username;
```

Grant permission to create any table

Sometimes user is restricted from creating some tables with names which are reserved for system tables. But we can grant privileges to a user to create any table using the below command,

Syntax:

```
GRANT CREATE ANY TABLE TO username;
```

Grant permission to drop any table:

As the title suggests, if you want to allow user to drop any table from the database, then grant this privilege to the user,

Syntax:

```
GRANT DROP ANY TABLE TO username;
```

Reverting Permissions:

If you want to take back the privileges from any user, use the REVOKE command.

Syntax:

```
REVOKE CREATE TABLE FROM username;
```

INDEXES AND SEQUENCES

CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables. Indexes are used to retrieve data from the database very fast. The users cannot see the indexes; they are just used to speed up searches/queries. Duplicate values are allowed:

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

Syntax:

```
SQL>CREATE INDEX index_name ON table_name (column1, column2, ...);
```

Example:

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table.

```
SQL>CREATE INDEX idx_lastname ON Persons (LastName);
```

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

```
SQL>CREATE INDEX idx_pname ON Persons (LastName, FirstName);
```

DROP INDEX Statement:

The DROP INDEX statement is used to delete an index in a table.

Syntax:

```
SQL>DROP INDEX index_name;
```

Example:

```
SQL>DROP INDEX idx_pname;
```

SEQUENCE:

Sequence is a feature supported by some database systems to produce unique values on demand. Some DBMS like MySQL supports AUTO_INCREMENT in place of Sequence.

AUTO_INCREMENT is applied on columns, it automatically increments the column value by 1 each time a new record is inserted into the table.

Sequence is also similar to AUTO_INCREMENT but it has some additional features too.

Syntax to create a sequence:

```
SQL>CREATE SEQUENCE sequence-name START WITH initial-value
      INCREMENT BY increment-value
      MAXVALUE maximum-value
      CYCLE / NOCYCLE;
```

The initial-value specifies the starting value for the Sequence.

The increment-value is the value by which sequence will be incremented.

The maximum-value specifies the upper limit or the maximum value upto which sequence will increment itself.

The keyword CYCLE specifies that if the maximum value exceeds the set limit, sequence will restart its cycle from the beginning.

And, NO CYCLE specifies that if sequence exceeds MAXVALUE value, an error will be thrown.

Using Sequence in SQL Query

Let's start by creating a sequence, which will start from 1, increment by 1 with a maximum value of 999.

Example:

```
SQL> CREATE SEQUENCE seq_1
      START WITH 1
      INCREMENT BY 1
      MAXVALUE 999
      CYCLE;
```

```
SQL>INSERT INTO class VALUE(seq_1.nextval, 'anu');
```

Resultset table will look like,

ID	NAME
1	abhi

2	adam
4	alex
1	anu

Once you use nextval the sequence will increment even if you don't insert any record into the table.

TASK – 9

Practices on DCL Commands

1. SQL>Create user test identified by pswd;

Output:

```
SQL> create user test identified by pswd;  
User created.
```

2.SQL> Grant create session, create table, create sequence, create view to test;

Output:

```
SQL> Grant create session, create table, create sequence, create view to test;  
Grant succeeded.
```

3. SQL>Create role manager;

SQL>Grant create table, create view to manager;

SQL>Grant manager to test;

Output:

```
SQL> create role manager;  
Role created.  
SQL> grant create table, create view to manager;  
Grant succeeded.  
SQL> grant manager to test;  
Grant succeeded.
```

4. SQL>Alter user test identified by qwerty;

Output:

```
SQL> alter user test identified by qwerty;  
User altered.
```

5. SQL>Grant select on employees to test;

Output:

```
SQL> grant select on hr.emp to test;  
Grant succeeded.
```

6. SQL>Grant update (department_name,location_id) on departments to test;

Output:

```
SQL> grant update <dname, loc> on hr.dept to test;
Grant succeeded.
```

7. SQL> Grant select, insert on hr.locations to test;

Output:

```
SQL> grant select, insert on hr.dept to test;
Grant succeeded.
```

8. SQL> Revoke select, insert on departments from test;

Output:

```
SQL> revoke select, insert on hr.dept from test;
Revoke succeeded.
```

INDEXES

1. Function based indexes:

SQL> create index emp_index on emp (upper(ename));

Output:

```
SQL> create index emp_index on emp <upper(ename)>;
Index created.
```

SQL> select employee_id, last_name, job_id from employees where last_name between 'N' and 'P';

Output:

```
SQL> select ename from emp where ename between 'N' and 'P';
ENAME
-----
OConnell
```

2. Creating Index while creating Table.

SQL> create table emp2 (empno number(6) PRIMARY KEY USING INDEX (CREATE INDEX emp_idx ON emp2(empno)), ename varchar2(20), job varchar2(20));

Output:

```
SQL> create table emp2 (empno number(6) PRIMARY KEY USING INDEX (CREATE INDEX emp_idx ON emp2
(empno)) ,ename varchar2(20),job varchar2(20));
Table created.
```

User-defined indexes:

```
SQL>select index_name,table_name from user_indexes where table_name='EMP2';
```

Output:

```
SQL> select index_name,table_name from user_indexes where table_name='EMP2';
INDEX_NAME          TABLE_NAME
-----
EMP_IDX             EMP2
```

3. create table emp3(empno number(6) primary key, ename varchar2(20), job varchar2(10));

Output:

```
SQL> create table emp3(empno number(6) primary key, ename varchar2(20), job varchar2(10));
Table created.
```

Default indexes.

```
SQL>select index_name,table_name from user_indexes where table_name='EMP3';
```

Output:

```
SQL> select index_name,table_name from user_indexes where table_name='EMP3';
INDEX_NAME          TABLE_NAME
-----
SYS_C007173        EMP3
```

4.Displaying all the indexes.

```
SQL>Select index_name, table_name from user_indexes;
```

Output:

```
SQL> select index_name, table_name from user_indexes;
INDEX_NAME          TABLE_NAME
-----
REG_ID_PK           REGIONS
LOCATIONS_PK_IDX    LOCATIONS_NAMED_INDEX
LOC_ID_PK           LOCATIONS
LOC_CITY_IX         LOCATIONS
LOC_STATE_PROVINCE_IX LOCATIONS
LOC_COUNTRY_IX      LOCATIONS
JHIST_EMP_ID_ST_DATE_PK JOB_HISTORY
```

4.SQL>select table_name, index_name, column_name from user_ind_columns where table_name='EMPLOYEES';

Output:

```
SQL> select table_name, index_name, column_name from user_ind_columns where table_name='EMPLOYEES';
```

TABLE_NAME	INDEX_NAME	COLUMN_NAME
EMPLOYEES	EMP_EMAIL_UK	EMAIL
EMPLOYEES	EMP_EMP_ID_PK	EMPLOYEE_ID
EMPLOYEES	EMP_DEPARTMENT_IX	DEPARTMENT_ID

5. Dropping an index:

```
SQL> drop index emp_index;
```

Output:

```
SQL> drop index emp_index;  
Index dropped.
```

SEQUENCE

1. SQL>create sequence my_seq start with 10 increment by 10 maxvalue 100 nocache;

Output:

```
SQL> create sequence my_seq start with 10 increment by 10 maxvalue 100 nocache;  
Sequence created.
```

2. SQL>select my_seq.nextval from dual;

Output:

```
SQL> select my_seq.nextval from dual;

NEXTVAL
-----
      10
```

3. SQL>select my_seq.currval from dual;

Output:

```
SQL> select my_seq.currval from dual;

CURRVAL
-----
      10
```

4. SQL>create table dept(deptno number(6),dname varchar2(20),loc varchar2(10));

SQL>insert into dept values(my_seq.nextval,'Executive','US');

SQL>insert into dept values(my_seq.nextval,'Marketing','UK');

```
SQL> create table dept1(id number(3), dname varchar2(10));
Table created.
SQL> insert into dept1 values(my_seq.nextval, 'Admin');
1 row created.
```

```
SQL> select * from dept1;

ID DNAME
-----
  20 Admin
```

5. SQL>drop sequence my_seq;

```
SQL> drop sequence my_seq;
Sequence dropped.
```

EXERCISE 9

1. Grant user1 query privilege on the regions table. Also check, whether that user can use the privilege.
2. Grant another user1 query privilege on the regions table. Include an option for this user to further grant this privilege to user2. Also check, whether that user can use the privilege.
3. Take back the privileges from the user2.
4. Grant user3 privileges to query and manipulate on the COUNTRIES table. Make sure that the user cannot pass on these privileges to other users.
5. Take back the privileges on the COUNTRIES table granted to another user
6. Create a sequence that starts at 200 and have a maximum value of 1000. Increment the sequence by 10. Name the sequence DEPT_ID_SEQ.
7. Use the sequence created above to insert two records in the dept table.
8. Create a non unique index on the NAME column in the DEPT table.

GRIET

SQL LAB QUERIES

1. DISPLAY THE DEPT INFORMATION FROM DEPARTMENT TABLE

```
SQL> SELECT * FROM DEPT;
```

2. DISPLAY THE DETAILS OF ALL EMPLOYEES.

```
SQL> SELECT * FROM EMP;
```

3. DISPLAY THE NAME AND JOB FOR ALL EMPLOYEES.

```
SQL> SELECT ENAME, JOB FROM EMP;
```

4. DISPLAY THE NAME AND SALARY FOR ALL EMPLOYEES.

```
SQL> SELECT ENAME, SAL FROM EMP;
```

5. DISPLAY EMPLOYEE NUMBER AND TOTAL SALARY FOR EACH EMPLOYEE.

```
SQL> SELECT EMPNO, SAL + NVL(COMM, 0) FROM EMP;
```

6. DISPLAY EMPLOYEE NAME AND ANNUAL SALARY FOR ALL EMPLOYEES

```
SQL> SELECT EMPNO, (SAL + NVL(COMM, 0)) * 12 "ANNUAL SAL" FROM EMP;
```

7. DISPLAY THE NAMES OF ALL EMPLOYEES WHO ARE WORKING IN DEPARTMENT NUMBER 10.

```
SQL > SELECT ENAME FROM EMP WHERE DEPTNO = 10;
```

8. DISPLAY THE NAMES OF ALL EMPLOYEES WHO ARE WORKING AS CLERKS AND DRAWING A SALARY MORE THAN 3000.

```
SQL> SELECT ENAME FROM EMP WHERE JOB = 'CLERK' AND SAL > 3000;
```

9. DISPLAY EMPLOYEE NUMBER AND NAMES FOR EMPLOYEES WHO EARN COMMISSION.

```
SQL> SELECT EMPNO, ENAME FROM EMP WHERE COMM IS NOT NULL;
```

10. DISPLAY NAMES OF EMPLOYEES WHO DONOT EARN COMISSION.

```
SQL> SELECT ENAME FROM EMP WHERE COMM = 0 OR COMM IS NULL;
```

11.DISPLAY THE NAME OF EMPLOYEES WHO ARE WORKING AS CLERK,SALESMAN OR ANALYST AND DRAWING A SALARY MORE THAN 3000.

```
SQL> SELECT ENAME FROM EMP WHERE SAL >3000 AND JOB='CLERK' OR JOB =  
'SALESMAN' OR JOB = 'ANALYST';
```

12.DISPLAY THE NAMES OF EMPLOYEES WHO ARE WORKING IN THE COMPANY FOR THE PAST

FIVE YEARS.

```
SQL> SELECT ENAME FROM EMP WHERE SYSDATE-HIREDATE > 5;
```

13. DISPLAY THE NAMES OF EMPLOYEES WHO HAVE JOINED THE COMPANY BEFORE 30THJUNE 90 AND AFTER 31ST DEC 90.

```
SQL> SELECT ENAME FROM EMP WHERE HIREDATE NOT BETWEEN '30-JUNE-1980'  
AND  
'31-DEC-1981'
```

14.DISPLAY CURRENT DATE.

```
SQL> SELECT SYSDATE FROM DUAL;
```

15. DISPLAY THE LIST OF USERS IN YOUR DATABASE.

```
SQL> SELECT * FROM USER_USERS;
```

16.DISPLAY THE NAMES OF ALL TABLES FROM THE CURRENT USER.

```
SQL> SELECT * FROM USER_USERS;
```

17.DISPLAY THE NAME OF CURRENT USER.

```
SQL> SHOW USER;
```

18.DISPLAY THE NAMES OF EMPLOYEES WORKING IN DEPARTMENT NUMBER 10 OR 20 OR 40 OR

EMPLOYEES WORKING AS CLERKS , SALESMAN OR ANALYST.

SQL> SELECT ENAME FROM EMP WHERE DEPTNO = 10 OR DEPTNO =20 OR DEPTNO = 40 AND

JOB = 'CLERK' OR JOB = 'SALESMAN' OR JOB = 'ANALYST';

19.DISPLAY THE NAMES OF EMPLOYEES WHOSE NAMES START WITH ALPHABET S.

SQL> SELECT ENAME FROM EMP WHERE ENAME LIKE 'S%';

20. DISPLAY THE NAMES OF EMPLOYEE WHOSE NAME ENDS WITH ALPHABET S.

SQL> SELECT ENAME FROM EMP WHERE ENAME LIKE '%S';

21. DISPLAY THE NAMES OF EMPLOYEES WHOSE NAMES HAVE SECOND ALPHABET A IN THEIR

NAMES.

SQL> SELECT ENAME FROM EMP WHERE ENAME LIKE '_A%';

22. DISPLAY THE NAMES OF EMPLOYEES WHOSE NAME IS EXACTLY 5 CHARCTERS IN LENGTH.

SQL> SELECT ENAME FROM EMP WHERE LENGTH(ENAME)= 5;

23. DISPLAY THE NAMES OF EMPLOYEES WHO ARE NOT WORKING AS MANAGERS.

SQL> SELECT ENAME FROM EMP WHERE JOB != 'MANAGER';

24. DISPLAY THE NAMES OF EMPLOYEES WHO ARE NOT WORKING AS SALESMAN OR CLERK OR

ANALYST.

SQL> SELECT ENAME FROM EMP WHERE JOB != 'CLERK' OR JOB !='SALESMAN' OR

JOB != 'ANALYST';

25. DISPLAY ALL ROWS FROM EMP TABLE.

SQL> SELECT * FROM EMP;

26.DISPLAY THE TOTAL NUMBER OF EMPLOYEES WORKING IN THE COMPANY.

```
SQL> SELECT COUNT(*) FROM EMP;
```

27. DISPLAY THE TOTAL SALARY BEING PAID TO ALL EMPLOYEES.

```
SQL> SELECT SUM(SAL+NVL(COMM,0)) FROM EMP;
```

28.DISPLAY THE MAXIMUM SALARY FROM THE EMPLOYEE TABLE.

```
SQL> SELECT MAX(SAL+NVL(COMM,0)) FROM EMP;
```

29.DISPLAY THE MINIMUM SALARY FROM EMPLOYEE TABLE.

```
SQL> SELECT MIN(SAL+NVL(COMM,0)) FROM EMP;
```

30.DISPLAY THE AVERAGE SALARY FROM EMPLOYEE TABLE.

```
SQL> SELECT AVG(SAL+NVL(COMM,0)) FROM EMP;
```

31. DISPLAY THE MAXIMUM SALARY FROM EMPLOYEE TABLE BEING PAID TO CLERK.

```
SQL> SELECT MAX(SAL+NVL(COMM,0)) FROM EMP WHERE JOB = 'CLERK';
```

32 DISPLAY THE MAXIMUM SALARY FROM EMPLOYEE TABLE BEING PAID TO
DEPARTMENT NO 20.

```
SQL> SELECT MAX(SAL+NVL(COMM,0)) FROM EMP WHERE DEPTNO = 20;
```

33. DISPLAY THE MINIMUM SALARY FROM EMPLOYEE TABLE BEING PAID TO
SALESMAN.

```
SQL> SELECT MIN(SAL+NVL(COMM,0)) FROM EMP WHERE JOB = 'SALESMAN';
```

34. DISPLAY THE AVERAGE SALARY FROM EMPLOYEE TABLE DRAWN BY
MANAGERS.

```
SQL> SELECT AVG(SAL+NVL(COMM,0)) FROM EMP WHERE JOB = 'MANAGER';
```

35.DISPLAY THE TOTAL SALARY DRAWN BY ANALYST WORKING IN DEPT NO 30.

```
SQL> SELECT SUM(SAL+NVL(COMM,0)) FROM EMP WHERE JOB = 'ANALYST' AND  
DEPTNO = 30;
```

36. DISPLAY THE NAMES OF EMPLOYEES IN ASCENDING ORDER OF SALARY .

SQL> SELECT ENAME,SAL FROM EMP ORDER BY SAL;

37. DISPLAY THE NAMES OF EMPLOYEES IN DESCENDING ORDER OF SALARY.

SQL> SELECT * FROM EMP ORDER BY SAL DESC

38.DISPLAY THE DETAILS FROM EMP TABLE IN ORDER OF EMP NAME.

SQL> SELECT * FROM EMP ORDER BY ENAME;

39. DISPLAY EMPNO, ENAME, DEPTNO, AND SAL.SORT THE OUTPUT FIRST BASED ON NAME AND WITHIN NAME BY DEPTNO AND WITHIN DEPTNO BY SAL.

SQL> SELECT * FROM EMP ORDER BY ENAME,DEPTNO,SAL;

40. DISPLAY THE NAME OF THE EMPLOYEE ALONG WITH THEIR ANNUAL SALARY (SAL*12). THE NAME OF EMPLOYEE EARNING HIGHEST SALARY SHOULD APPEAR FIRST.

41. DISPLAY NAME,SAL,HRA,PF,DA,TOTALSAL FOR EACH EMPLOYEE.THE OUTPUT SHOULD BE IN THE ORDER OF TOTAL SAL ,HRA 15% OF SAL,DA 10% OF SAL , PF 5% OF SAL, TOTAL SAL WILL BE SAL+HRA+DA-PF.

42 DISPLAY THE DEPT NUMBERS AND TOTAL NUMBER OF EMPLOYEES IN EACH GROUP.

QL> SELECT DEPTNO,COUNT(*) FROM EMP GROUP BY DEPTNO;

43. DISPLAY VARIOUS JOBS AND TOTAL NUMBER OF EMPLOYEES WITHIN EACH JOB GROUP.

SQL> SELECT DISTINCT JOB,COUNT(*) FROM EMP GROUP BY JOB;

44. DISPLAY DEPT NUMBERS AND TOTAL SALARY FOR EACH DEPARTMENT.

SQL> SELECT DEPTNO,SUM(SAL+NVL(COMM,0)) FROM EMP GROUP BY DEPTNO;

45. DISPLAY DEPT NUMBERS AND MAXIMUM SALARY FOR EACH DEPARTMENT.

SQL> SELECT DEPTNO,MAX(SAL+NVL(COMM,0)) FROM EMP GROUP BY DEPTNO;

46. DISPLAY VARIOUS JOBS AND TOTAL SALARY FOR EACH JOB.

SQL> SELECT DISTINCT JOB,SUM(SAL+NVL(COMM,0)) FROM EMP GROUP BY JOB;

47.DISPLAY EACH JOB ALONG WITH MINIMUM SALARY BEING PAID IN EACH JOB GROUP.

```
SQL> SELECT DISTINCT JOB,MIN(SAL+NVL(COMM,0)) FROM EMP GROUP BY JOB;
```

48. DISPLAY THE DEPARTMENT NUMBERS WITH MORE THAN THREE EMPLOYEES IN EACH DEPT.

```
SQL> SELECT DEPTNO,COUNT(*) FROM EMP GROUP BY DEPTNO HAVING COUNT(*)>3;
```

49. DISPLAY THE VARIOUS JOBS ALONG WITH TOTAL SAL FOR EACH OF THE JOBS WHERE

TOTAL SAL > 40000.

```
SQL> SELECT DISTINCT(JOB),SUM(SAL+NVL(COMM,0)) FROM EMP GROUP BY JOB HAVING
```

```
SUM(SAL+NVL(COMM,0)) > 40000;
```

50. DISPLAY THE VARIOUS JOBS ALONG WITH TOTAL NUMBER OF EMPLOYEES IN EACH JOB.

```
SQL> SELECT DISTINCT(JOB),COUNT(*) FROM EMP GROUP BY JOB HAVING COUNT(*)>3;
```

51. DISPLAY THE NAME OF EMP WHO EARNS HIGHEST SAL.

```
SQL> SELECT ENAME FROM EMP WHERE SAL =(SELECT MAX(SAL) FROM EMP);
```

52. DISPLAY THE EMPLOYEE NUMBER AND NAME OF EMPLOYEE WORKING AS CLERK AND EARNING HIGHEST SALARY AMONG CLERKS.

```
SQL> SELECT ENAME,EMPNO FROM EMP WHERE JOB = 'CLERK' AND SAL =  
(SELECT MAX(SAL) FROM EMP WHERE JOB = 'CLERK');
```

53. DISPLAY THE NAMES OF THE SALESMAN WHO EARNS A SALARY MORE THAN THE HIGHEST SALARY OF ANY CLERK.

```
SQL> SELECT ENAME FROM EMP WHERE JOB = 'SALESMAN' AND SAL>  
(SELECT MAX(SAL) FROM EMP WHERE JOB = 'CLERK');
```

54. DISPLAY THE NAMES OF CLERKS WHO EARN SALARY MORE THAN THAT OF JAMES AND LESS THAN THAT OF SCOTT.

```
SQL> SELECT ENAME FROM EMP WHERE JOB = 'CLERK' AND  
SAL<(SELECT SAL FROM EMP WHERE ENAME LIKE 'SCOTT')
```

AND SAL > (SELECT SAL FROM EMP WHERE ENAME LIKE 'JAMES');

55. DISPLAY THE NAMES OF EMPLOYEES WHO EARN A SAL MORE THAN THAT OF JAMES OR THAT

SALARY GREATER THAN THAT OF SCOTT.

SQL> SELECT ENAME FROM EMP WHERE SAL > (SELECT SAL FROM EMP WHERE ENAME LIKE 'SCOTT');

56. DISPLAY THE NAMES OF THE EMPLOYEES WHO EARN HIGHEST SALARY IN THEIR RESPECTIVE DEPARTMENTS.

SQL> SELECT E.ENAME, E.SAL FROM EMP E WHERE E.SAL = (SELECT MAX(F.SAL) FROM EMP F GROUP BY F.DEPTNO HAVING E.DEPTNO = F.DEPTNO);

57. DISPLAY THE EMPLOYEE NAMES WHO ARE WORKING IN ACCOUNTING DEPT.

SQL> SELECT ENAME FROM EMP E, DEPT D WHERE E.DEPTNO = D.DEPTNO AND DNAME = 'ACCOUNTING';

58. DISPLAY THE EMPLOYEE NAMES WHO ARE WORKING IN CHICAGO.

SQL> SELECT ENAME FROM EMP E, DEPT D WHERE E.DEPTNO = D.DEPTNO AND LOC = 'CHICAGO';

59. DISPLAY THE JOB GROUPS HAVING TOTAL SALARY GREATER THAN THE MAXIMUM SALARY

FOR MANAGERS.

SQL> SELECT DISTINCT(JOB), SUM(SAL + NVL(COMM, 0)) FROM EMP GROUP BY JOB HAVING SUM(SAL + NVL(COMM, 0)) > (SELECT MAX(SAL) FROM EMP WHERE JOB = 'MANAGER');

60. DISPLAY THE NAMES OF EMPLOYEES FROM DEPARTMENT NUMBER 10 WITH SALARY

GREATER THAN THAT OF ANY EMPLOYEE WORKING IN OTHER DEPARTMENTS.

SQL> SELECT ENAME FROM EMP WHERE DEPTNO = 10 AND SAL > ANY(SELECT SAL FROM EMP WHERE DEPTNO != 10);

61. DISPLAY THE NAMES OF EMPLOYEE FROM DEPARTMENT NUMBER 10 WITH SALARY

GREATER THAN THAT OF ALL EMPLOYEE WORKING IN OTHER DEPARTMENTS.

```
SQL> SELECT ENAME FROM EMP WHERE DEPTNO = 10 AND  
SAL > ALL(SELECT SAL FROM EMP WHERE DEPTNO != 10);
```

62. DISPLAY THE NAMES OF EMPLOYEES IN UPPER CASE.

```
SQL> SELECT UPPER(ENAME) FROM EMP;
```

63. DISPLAY THE NAMES OF EMPLOYEES IN LOWER CASE.

```
SQL> SELECT LOWER(ENAME) FROM EMP;
```

64. DISPLAY THE NAMES OF EMPLOYEES IN PROPER CASE.

```
SQL> SELECT INITCAP(ENAME) FROM EMP;
```

65. FIND OUT THE LENGTH OF YOUR NAME USING APPROPRIATE FUNCTION.

```
SQL> SELECT LENGTH('&NAME') FROM DUAL;
```

66. DISPLAY THE LENGTH OF ALL EMPLOYEES' NAMES.

```
SQL> SELECT LENGTH(ENAME) FROM EMP;
```

67. DISPLAY THE NAME OF THE EMPLOYEE CONCATENATE WITH EMPNO.

```
SQL> SELECT CONCAT(ENAME,EMPNO) FROM EMP;
```

68. USE APPROPRIATE FUNCTION AND EXTRACT 3 CHARACTERS STARTING FROM 2 CHARACTERS

FROM THE FOLLOWING STRING 'ORACLE' I.E THE OUTPUT SHOULD BE 'RAC'.

```
SQL> SELECT SUBSTR('ORACLE',2,3) FROM DUAL;
```

69. FIND THE FIRST OCCURENCE OF CHARACTER A FROM THE FOLLOWING STRING 'COMPUTER

MAINTAINENCE CORPORATION'

```
SQL> SELECT INSTR('COMPUTER MAINTENANCE CORPORATION','A') FROM DUAL;
```

70. REPLACE EVERY OCCURENCE OF ALPHABET A WITH B IN THE STRING ALLEN'S.

```
SQL> SELECT REPLACE('ALLEN','A','B') FROM DUAL;
```

71. DISPLAY THE INFORMATION FROM EMP TABLE.WHEREVER JOB 'MANAGER' IS FOUND ., IT SHOULD BE DISPLAYED AS BOSS.

```
SQL> SELECT JOB,DECODE(JOB,'MANAGER','BOSS',' ') FROM EMP;
```

72. DISPLAY EMPNO,ENAME,DEPTNO FROM EMP TABLE.INSTEAD OF DISPLAY DEPARTMENT

NUMBERS DISPLAY THE RELATED DEPARTMENT NAME.

```
SQL> SELECT E.EMPNO, E.ENAME, D.DNAME
```

```
2 FROM EMP E, DEPT D
```

```
3 WHERE E.DEPTNO = D.DEPTNO;
```

73. DISPLAY YOUR AGE IN DAYS.

```
SQL> SELECT MONTHS_BETWEEN('22-FEB-87',SYSDATE)*30 FROM DUAL;
```

74. DISPLAY YOUR AGE IN MONTHS.

```
SQL> SELECT MONTHS_BETWEEN('22-FEB-87',SYSDATE) FROM DUAL;
```

75. DISPLAY CURRENT DATE AS 15TH AUGUST FRIDAY NINETEEN FORTY SEVEN.

```
SQL> SELECT TO_CHAR(SYSDATE,'DDTH MONTH YEAR') FROM DUAL;
```

76. DISPLAY THE FOLLOWING OUTPUT FOR EACH ROW FROM EMP TABLE AS 'SCOTT HAS JOINED

THE COMPANY ON WEDNESDAY 13TH AUGUST NINETEEN NINETY'.

```
SQL> SELECT ENAME || 'HAS JOINED THE COMPANY ON'
```

```
|| TO_CHAR(HIREDATE,'DAY DDTH MONTH YEAR') FROM EMP;
```

77. DISPLAY THE COMMON JOBS FROM DEPARTMENT NUMBER 10 AND 20.

```
SQL> SELECT JOB FROM EMP WHERE DEPTNO = 10 AND DEPTNO =20;
```

78. DISPLAY THE JOBS FOUND IN DEPARTMENT NUMBER 10 AND 20 ELIMINATE DUPLICATE JOBS.

```
SQL> SELECT DISTINCT(JOB) FROM EMP WHERE DEPTNO = 10 AND DEPTNO =20;
```

79. DISPLAY THE DETAILS OF EMPLOYEES WHO ARE IN SALES DEPT AND GRADE IS 3.

```
SQL> SELECT E.ENAME,D.DNAME,S.GRADE FROM EMP E,DEPT D,SALGRADE S
      WHERE E.DEPTNO = D.DEPTNO AND D.DNAME = 'SALES' AND S.GRADE = 3;
```

80. DISPLAY THOSE EMPLOYEES WHOSE NAME CONTAINS NOT LESS THAN 4 CHARS.

```
SQL> SELECT ENAME FROM EMP WHERE LENGTH(ENAME)>3;
```

81. DISPLAY THOSE DEPARTMENTS WHOSE NAME START WITH 'S' WHILE LOCATION NAME

END WITH 'O';

```
SQL> SELECT DNAME FROM DEPT WHERE DNAME LIKE 'S%' AND LOC LIKE '%O';
```

82. DISPLAY THOSE EMPLOYEES WHOSE MANAGER NAME IS JONES,AND ALSO DISPLAY THERE MANAGER NAME

```
SQL> SELECT E.ENAME EMPLOYEE ,M.ENAME MANAGER FROM EMP E,EMP M WHERE
      E.MGR=M.EMPNO AND M.ENAME='JONES';
```

83)DISPLAY NAME AND SALARY OF FORD IF HIS SAL IS EQUAL TO HIGH SAL OF HIS GRADE.

```
SQL>SELECT E.ENAME,E.SAL FROM EMP E,SALGRADE S WHERE E.ENAME='FORD' AND
      E.SAL=S.HISAL;
```

84)DISPLAY EMPLOYEE NAME,HIS JOB,HIS DEPT NAME,HIS MANAGER NAME,HIS GRADE AND MAKE OUT OF AN UNDER DEPT WISE.

```
SQL>SELECT E.ENAME EMPLOYEE,M.ENAME MANAGER,E.JOB,D.DNAME,S.GRADE
      FROM EMP E,DEPT D,SALGRADE S,EMP M
      GROUP BY
      M.ENAME,E.MGR,M.EMPNO,E.SAL,E.ENAME,E.JOB,D.DEPTNO,D.DNAME,S.GRADE,S.LO
      SAL,S.HISAL,E.DEPTNO HAVING E.MGR=M.EMPNO AND
      E.DEPTNO=D.DEPTNO AND E.SAL BETWEEN S.LOSAL AND S.HISAL
```

85)LIST OUT ALL THE EMPLOYEES NAME,HIS JOB, HIS DEPT NAME AND SALARY GRADE FOR EVERY ONE IN THE COMPANY EXCEPT 'CLERK'.SORT ON SALARY.

```
SQL>SELECT E.ENAME EMPLOYEE,M.ENAME MANAGER,S.GRADE,D.DNAME FROM
      EMP E,EMP M,SALGRADE S,DEPT D
      WHERE E.MGR=M.EMPNO AND E.ENAME!='CLARK' AND D.DEPTNO=E.DEPTNO AND
      E.SAL BETWEEN S.LOSAL AND S.HISAL ORDER BY E.SAL DESC ;
```

86)DISPLAY THE NAME OF THE EMPLOYEE WHO IS GETTING MAXIMUM SALSRY.

```
SQL> SELECT * FROM EMP WHERE SAL=(SELECT MAX(SAL) FROM EMP);
```

87)FIND OUT THE TOP FIVE EARNERS OF THE COMPANY.

```
SQL>SELECT * FROM EMP E WHERE 5>(SELECT COUNT(*) FROM EMP M WHERE  
E.SAL<M.SAL);
```

88)DISPLAY THOSE EMPLOYEES WHOSE SALARY IS EQUAL TO AVERAGE OF MAXIMUM AND MINIMUM SALARY.

```
SQL> SELECT * FROM EMP WHERE SAL IN (SELECT (MAX(SAL)+MIN(SAL))/2 FROM  
EMP);
```

89)DISPLAY COUNT OF EMPLOYEES IN EACH DEPARTMENT WHERE COUNT GREATER THAN OR EQUAL TO 3.

```
SQL> SELECT COUNT(*) FROM EMP WHERE 3<(SELECT COUNT(*) FROM EMP) GROUP  
BY DEPTNO;
```

90)DISPLAY DNAME WHERE ATLEAST 3 ARE WORKING AND DISPLAY ONLY DNAME.

```
SQL> SELECT D.DNAME FROM EMP E,DEPT D WHERE 3<(SELECT COUNT(*) FROM EMP)  
AND D.DEPTNO=E.DEPTNO GROUP BY D.DNAME,E.DEPTNO,D.DEPTNO;
```

91)DISPLAY NAME OF THOSE MANAGERS NAME WHOSE SALARY IS MORE THAN AVERAGE SALARY OF HIS EMPLOYEES.

```
SQL>SELECT E.ENAME FROM EMP E,EMP M WHERE E.SAL IN(SELECT AVG(SAL) FROM  
EMP) AND E.MGR=M.EMPNO GROUP BY E.MGR,E.ENAME,E.SAL,M.EMPNO
```

92)DISPLAY EMPLOYEE NAME,SAL,COMM,NET PAY FOR THOSE EMPLOYEES WHOSE NET PAY ARE GREATER THAN OR EQUAL TO ANY OTHER EMPLOYEE SALARY OF THE COMPANY.

```
SQL> SELECT ENAME,SAL,COMM,SAL+NVL(COMM,0) NET_PAY FROM EMP E  
WHERE SAL+NVL(COMM,0)>ANY(SELECT SAL+NVL(COMM,0) FROM EMP M WHERE  
E.DEPTNO!=M.DEPTNO)
```

93)DISPLAY THOSE EMPLOYEES WHOSE SALARY IS LESS THAN HIS MANAGER BUT MORE THAN SALARY OF ANY OTHER MANAGER.

```
SQL>SELECT E.ENAME,E.SAL FROM EMP E WHERE E.SAL<(SELECT N.SAL FROM EMP N  
WHERE E.MGR=N.EMPNO) AND SAL>ANY(SELECT M.SAL FROM EMP M WHERE  
E.MGR!=M.EMPNO)
```


94)FIND OUT THE LEAST FIVE EARNERS.

```
SQL> SELECT * FROM EMP E WHERE 5>(SELECT COUNT(*) FROM EMP M WHERE  
E.SAL>M.SAL);
```

95)FIND OUT THE NO.OF EMPLOYEES WHOSE SALARY IS GREATER THAN THERE MANAGERS SALARY.

```
SQL> SELECT COUNT(*) FROM EMP E,EMP M WHERE E.MGR=M.EMPNO AND  
E.SAL>M.SAL;
```

96)DISPLAY THOSE MANAGERS WHO ARE NOT WORKING UNDER PRESIDENT BUT THEY ARE WORKING UNDER ANY OTHER MANAGER.

```
SQL>SELECT E.ENAME FROM EMP E WHERE MGR IN(SELECT M.EMPNO FROM EMP M  
WHERE M.ENAME!='KING' AND E.MGR=M.EMPNO);
```

97)DELETE THOSE DEPARTMENT WHERE NO EMPLOYEE WORKING.

```
SQL> DELETE EMP WHERE DEPTNO=(SELECT DEPTNO FROM EMP GROUP BY DEPTNO  
HAVING COUNT(DEPTNO)=0);
```

98)DELETE THOSE RECORDS FROM EMP TABLE WHOSE DEPTNO NOT AVAILABLE IN DEPT TABLE.

```
SQL> DELETE EMP WHERE DEPTNO NOT IN(SELECT DEPTNO FROM DEPT);
```

99)DISPLAY THOSE EARNERS WHOSE SALARY IS OUT OF THE GRADE AVAILABLE IN SALGRADE TABLE.

```
SQL> SELECT E.ENAME,E.SAL FROM EMP E,SALGRADE S WHERE E.SAL<LOSAL AND  
E.SAL>HISAL;
```

100) DISPLAY EMPLOYEE NAME,SAL,COMM WHOSE NET PAY IS GREATER THAN ANY OTHER IN TH COMPANY.

```
SQL> SELECT ENAME,SAL,COMM FROM EMP WHERE SAL+NVL(COMM,0)=(SELECT  
MAX(SAL+NVL(COMM,0)) FROM EMP);
```

101)DISPLAY THOSE EMPLOYEES WHO ARE GOING TO RETIRE ON 31-DEC-99.IF THE MAX JOB PERIOD IS 18 YEARS.

```
SQL>SELECT * FROM EMP WHERE MONTHS_BETWEEN('31-DEC-99',HIREDATE)>18*12;
```

102)DISPLAY THOSE EMPLOYEES WHOSE SALARY IS ODD VALUE.

```
SQL> SELECT * FROM EMP WHERE MOD(SAL,2)=1;
```

103)DISPLAY THOSE EMPLOYEES WHOSE SALARY CONTAINS ATLEAST 4 DIGITS.

```
SQL> SELECT * FROM EMP WHERE LENGTH(SAL)>=4;
```

104)DISPLAY THOSE EMPLOYEES WHOSE NAME CONTAINS A IN THEM.

```
SQL> SELECT * FROM EMP WHERE ENAME LIKE '%A%';
```

105)DISPLAY THOSE EMPLOYEES ,WHOSE FIRST 2 CHARACTERS FROM HIREDATE IS EQUAL TO LAST 2 CHARACTERS FROM SALARY.

```
SQL> SELECT * FROM EMP WHERE SUBSTR(HIREDATE,1,2) IN (SELECT  
SUBSTR(SAL,LENGTH(SAL)-2,2) FROM EMP);
```

106)DISPLAY THOSE EMPLOYEES WHOSE 10% SALARY IS EQUAL TO THE YEAR OF JOINING.

```
SQL> SELECT * FROM EMP WHERE 0.1*SAL IN (SELECT  
TO_NUMBER(TO_CHAR(HIREDATE,'YY')) FROM EMP);
```

107)DISPLAY THOSE EMPLOYEES WHO ARE WORKING IN SALES OR RESEARCH DEPT.

```
SQL> SELECT * FROM EMP WHERE DEPTNO IN (SELECT DEPTNO FROM DEPT WHERE  
DNAME IN('SALES','RESEARCH'));
```

108)DISPLAY THE GRADE OF JONES.

```
SQL>SELECT E.ENAME,E.SAL,S.GRADE FROM EMP E,SALGRADE S  
2 WHERE E.ENAME='JONES' AND E.SAL BETWEEN S.LOSAL AND S.HISAL;
```

109)DISPLAY THOSE EMPLOYEES WHO JOINED IN THE COMPANY BEFORE 15th OF THE MONTH.

```
SQL>SELECT * FROM EMP WHERE TO_NUMBER(TO_CHAR(HIREDATE,'DD'))<15;
```

110)DELETE THOSE EMPLOYEES WHERE NO.OF EMPLOYEES IN A PARTICULAR DEPT IS LESS THAN 3.

```
SQL> DELETE EMP WHERE 3>ANY(SELECT COUNT(*) FROM EMP GROUP BY DEPTNO);
```

111)DELETE THOSE EMPLOYEES WHO JOINED IN THE COMPANY 21 YEARS BACK FROM TODAY.

```
SQL> DELETE EMP WHERE MONTHS_BETWEEN(SYSDATE,HIREDATE)=21*12;
```

112)DISPLAY THE DEPARTMENT NAME WHERE THE NO.OF CHARACTERS OF WHICH IS EQUAL TO

THE NO. OF EMPLOYEES IN ANY OTHER DEPARTMENT.

```
SQL>SELECT DNAME FROM DEPT WHERE LENGTH(DNAME) IN (SELECT COUNT(*)  
FROM EMP GROUP BY DEPTNO);
```

113)DISPLAY THOSE EMPLOYEES WHO ARE WORKING AS MANAGERS.

```
SQL> SELECT * FROM EMP WHERE JOB='MANAGER';
```

114)COUNT THE NO.OF EMPLOYEES WHO ARE WORKING AS MANAGERS.

```
SQL> SELECT COUNT(*) FROM EMP GROUP BY JOB HAVING JOB='MANAGER';
```

115)DISPLAY THE NAMES OF THE DEPARTMENT THOSE EMPLOYEES WHO JOINED THE COMPANY

ON THE SAME DAY.

```
SQL>SELECT DNAME FROM DEPT WHERE DEPTNO IN(SELECT E.DEPTNO FROM EMP E  
WHERE E.HIREDATE IN(SELECT M.HIREDATE FROM EMP M WHERE  
E.EMPNO!=M.EMPNO));
```

116)DISPLAY THE MANAGER WHO IS HAVING MAXIMUM NO.OF EMPLOYEES WORKING UNDER HIM.

```
SQL> SELECT ENAME FROM EMP WHERE EMPNO=(SELECT MGR FROM EMP GROUP BY  
MGR HAVING COUNT(MGR)=(SELECT MAX(COUNT(MGR)) FROM EMP GROUP BY  
MGR));
```

117)LIST OUT THE EMPLOYEES NAME AND SALARY INCREASED BY 15% AND EXPRESSED

AS WHOLE NUMBER OF DOLLAR.

```
SQL>SELECT ENAME,ROUND(1.15*SAL/48,2)||'$' DOLLAR FROM EMP;
```

118)LIST ALL THE EMPLOYEES WITH HIREDATE IN THE FORMAT 'JUNE 04 1988'.

```
SQL> SELECT ENAME,TO_CHAR(HIREDATE,'MONTH DD YYYY') HIREDATE FROM EMP;
```

119)PRINT A LIST OF EMPLOYEES DISPLAYING 'LESS SALARY' IF SAL<1500.IF EXACTLY 1500

DISPLAY AS 'EXACT SALARY' AND IF GREATER THAN 1500 DISPLAY AS 'MORE SALARY'.

```
SQL> SELECT ENAME,SAL||' LESS SALARY' FROM EMP WHERE SAL<1500  
UNION  
SELECT ENAME,SAL||' EXACT SALARY' FROM EMP WHERE SAL=1500  
UNION  
SELECT ENAME,SAL||' MORE SALARY' FROM EMP WHERE SAL>1500;
```

120)WRITE A QUERY TO CALCULATE THE LENGTH OF EMPLOYEE HAS BEEN WITH THE COMPANY.

```
SQL> SELECT ENAME,LENGTH(ENAME) LENGTH FROM EMP;
```

121)DISPLAY THOSE MANAGERS WHO ARE GETTING SALARY LESS THAN HIS EMPLOYEE.

```
SQL>SELECT E.ENAME,E.SAL FROM EMP E WHERE E.EMPNO IN  
(SELECT M.MGR FROM EMP M WHERE M.MGR=E.EMPNO AND E.SAL<M.SAL)
```

122)PRINT THE DETAILS OF ALL THE EMPLOYEES WHO ARE SUB ORDINATES TO BLAKE.

```
SQL> SELECT * FROM EMP WHERE MGR=(SELECT EMPNO FROM EMP WHERE  
ENAME='BLAKE');
```

123) DISPLAY THOSE EMPLOYEES WHOSE MANAGER NAME IS JONES AND ALSO WITH HIS
MANAGER NAME.

```
SQL> SELECT E.ENAME EMPLOYEE,M.ENAME MANAGER FROM EMP E,EMP M WHERE  
M.EMPNO=E.MGR AND M.ENAME='JONES';
```

124)DEFINE VARIABLE REPRESENTING THE EXPRESSIONS USED TO CALCULATE ON EMPLOYEES

TOTAL ANNUAL REMUNERATION.

```
SQL> DEFINE REM=(SAL+NVL(COMM,0))*12 ON EMP;  
SQL> SELECT ENAME,&REM FROM EMP;
```

125)FIND OUT THE AVG SAL AND AVG TOTAL REMUNERATION FOR EACH JOB TYPE.

SQL> SELECT AVG(SAL),AVG(SAL+NVL(COMM,0)) FROM EMP GROUP BY JOB;

126)LIST ENAME,JOB ANNUAL SAL,DEPTNO,DNAME AND GRADE WHO EARN MORE THAN

30000 PER YEAR AND WHO ARE NOT CLERKS.

SQL> SELECT E.ENAME,E.JOB,E.SAL*12 ANN_SAL,S.GRADE,D.DNAME FROM EMP E,SALGRADE S,DEPT D

WHERE E.SAL*12>30000 AND E.DEPTNO=D.DEPTNO AND E.SAL BETWEEN S.LOSAL AND S.HISAL

AND E.JOB!='CLERK';

127)FIND OUT THE JOB THAT WAS FILLED IN THE FIRST HALF OF 1983 AND THE SAME JOB THAT WAS FILLED DURING THE SAME PERIOD ON 1984.

SQL> SELECT JOB FROM EMP WHERE TO_NUMBER(TO_CHAR(HIREDATE,'MM'))<6 AND TO_NUMBER(TO_CHAR(HIREDATE,'YY'))=83

INTERSECT

SELECT JOB FROM EMP WHERE TO_NUMBER(TO_CHAR(HIREDATE,'MM'))<6 AND TO_NUMBER(TO_CHAR(HIREDATE,'YY'))=84;

128)LIST OUT THE LOWEST PAID EMPLOYEES WORKING FOR EACH MANAGER,EXCLUDE ANY

GROUPS WHERE MIN SAL IS LESS THAN 1000 SORT THE OUTPUT BY SAL.

SQL> SELECT E.ENAME,E.MGR,E.SAL FROM EMP E WHERE SAL IN (SELECT MIN(SAL) FROM EMP WHERE MGR=E.MGR) AND E.SAL>1000 ORDER BY SAL;

129)CHECK WHETHER ALL THE EMPLOYEE NO'S ARE INDEED UNIQUE.

SQL> SELECT COUNT(EMPNO),COUNT(DISTINCT(EMPNO)) FROM EMP HAVING COUNT(EMPNO)=COUNT(DISTINCT(EMPNO));

130.FIND OUT THE EMPLOYEES BY NAME AND NUMBER ALONG WITH THIER MANAGER'S NAME AND NUMBER ALSO DISPLAY NOMANAGER WHO HAS NOMANAGER .

SQL>SELECT E.EMPNO,E.ENAME,M.EMPNO,'MANAGER',M.ENAME MANAGERNAME FROM EMP E,EMP M WHERE E.MGR=M.EMPNO UNION SELECT T.EMPNO,T.ENAME,T.MGR,'NOMANAGER',X.ENAME FROM EMP T,EMP X WHERE T.MGR IS NULL AND T.EMPNO=X.EMPNO

SQL>SELECT * FROM EMP E WHERE SAL =(SELECT MIN(SAL) FROM EMP WHERE JOB=E.JOB);

132.FIND OUT MOST RECENTLY HIRED EMPLOYEE IN EACH DEPT

SQL>SELECT E.ENAME,E.DEPTNO FROM EMP E
WHERE E.HIREDATE=(SELECT MAX(HIREDATE) FROM EMP WHERE
DEPTNO=E.DEPTNO)

133. DISPLAY ENAME AND SALARY FOR EACH EMPLOYEE WHO EARN A SALARY >
THE

AVERAGE OF THEIR DEPARTMENT

SQL>SELECT E.ENAME,E.SAL,E.DEPTNO FROM EMP E WHERE
E.SAL > ANY(SELECT AVG(F.SAL) FROM EMP F WHERE E.DEPTNO=F.DEPTNO)ORDER
BY E.DEPTNO

134. SELECT THE DEPARTMENT NUMBER WHERE THERE ARE NO EMPLOYEES

SQL>SELECT D.DEPTNO FROM DEPT D
WHERE (SELECT COUNT(*) FROM EMP E WHERE E.DEPTNO=D.DEPTNO) = 0;

135.IN WHICH YEAR DID MOST PEOPLE JOIN THE COMPANY .DISPLAY THE YEAR AND
NUMBER OF EMPLOYEES.

SQL>SELECT TO_CHAR(HIREDATE,'YYYY'),COUNT(*) FROM EMP
GROUP BY TO_CHAR(HIREDATE,'YYYY') HAVING COUNT(*)=(SELECT MAX(COUNT(*))
FROM EMP
GROUP BY TO_CHAR(HIREDATE,'YYYY'))

136. DISPLAY AVERAGE SAL FIGURE FOR THE DIPARTMENT

SQL>SELECT AVG(SAL) FROM EMP GROUP BY DEPTNO

137. EMPLOYEES WHO EARN MORE THAN LOWEST SAL IN DEPT NO 30

SQL> SELECT * FROM EMP WHERE SAL>(SELECT MIN(SAL) FROM EMP WHERE
DEPTNO=30)

138. EMPLOYEES WHO EARN MORE THAN EVERY EMPLOYEE IN DEPT NO 30

SQL> SELECT * FROM EMP WHERE SAL>(SELECT MAX(SAL) FROM EMP WHERE DEPTNO=30);

139. SELECT DEPT NAME ,DEPT NUMBER & SUM OF SALARIES.

SQL> SELECT D.DNAME,D.DEPTNO,SUM(E.SAL) FROM EMP E,DEPT D
GROUP BY D.DEPTNO,D.DNAME,E.DEPTNO
HAVING D.DEPTNO=E.DEPTNO

140. FIND ALL DEPT'S WHICH HAVE MORE THAN 3 EMPLOYEES.

SQL> SELECT D.DNAME FROM DEPT D WHERE
3<ANY(SELECT COUNT(*) FROM EMP GROUP BY DEPTNO HAVING D.DEPTNO

141.DISPLAY HALF OF ENAME IN UPPER CASE AND HALF IN LOWER

SQL> SELECT
CONCAT(LOWER(SUBSTR(ENAME,1,LENGTH(ENAME)/2)),UPPER(SUBSTR(ENAME,LENGTH(ENAME)/2+1,LENGTH(ENAME)))) FROM EMP;

142.FIND OUT MOST RECENTLY HIRED EMPLOYEE IN EACH DEPT

SQL>SELECT E.ENAME,E.DEPTNO FROM EMP E
WHERE E.HIREDATE=(SELECT MAX(HIREDATE) FROM EMP WHERE DEPTNO=E.DEPTNO)

143. CREATE A VIEW OF EMP TABLE

SQL> CREATE VIEW EMP2 AS SELECT * FROM EMP;
SQL> SELECT * FROM EMP2;

144.SELECT ENAME IF ENAME EXISTS MORE THAN ONCE

SQL> SELECT E.ENAME FROM EMP E WHERE 1<ANY (SELECT COUNT(F.ENAME) FROM EMP F
GROUP BY F.ENAME HAVING E.ENAME=F.ENAME)

145.DISPLAY ALL ENAMES IN REVERSE ORDER

SELECT REVERSE(ENAME) FROM EMP

146.DISPLAY THOSE EMPLOYEES WHOOSE JOINED MONTH AND DATE ARE EQUAL

SQL> SELECT ENAME FROM EMP ,SALGRADE S WHERE
TO_CHAR(HIREDATE,'MM')=S.GRADE;

147.DISPLAY THOSE EMPLOYEES WHOOSE JOINING DATE IS AVAILABLE IN DEPT NO.

SQL> SELECT ENAME FROM EMP ,DEPT WHERE
TO_CHAR(HIREDATE,'DD')=DEPT.DEPTNO;

148.DISPLAY THE EMPLOYEES NAMES AS FOLLOWS .

A ALLEN,B BLAKE.

SQL> SELECT CONCAT(SUBSTR(ENAME,1,1),CONCAT(' ',ENAME)) FROM EMP

149.LIST OF EMPLOYEE NAME ,SAL ,PF FROM EMP

SQL>SELECT ENAME,SAL,&PF_AS_PERCENT_OF_SAL*SAL/100 "PF" FROM EMP

150.FIND OUT MOST RECENTLY HIRED EMPLOYEE IN EACH DEPT

SQL>SELECT ENAME,HIREDATE FROM EMP WHERE HIREDATE>
ANY(SELECT HIREDATE FROM EMP) GROUP BY HIREDATE,ENAME;

151)DISPLAY EMPLOYEE NAME,HIS JOB, HIS MANAGER.DISPLAY ALSO EMPLOYEES
WHO ARE WITHOUT MANAGER.

SQL> SELECT E.ENAME,E.JOB,M.ENAME FROM EMP E,EMP M WHERE E.MGR=M.EMPNO
OR E.MGR IS NULL;

152)DISPLAY NAMES OF THOSE MANAGERS WHOSE SALARY IS MAOR THAN
AVERAGE SALARY OF THE COMPANY.

SQL>SELECT M.ENAME FROM EMP E,EMP M WHERE M.SAL IN(SELECT AVG(SAL)
FROM EMP)

AND E.MGR=M.EMPNO

Introduction to PL/SQL

PL/SQL (Procedural Language / SQL) is an extension to SQL incorporating many of the design features of programming languages in recent years. It allows the data manipulation and query statements of SQL to be included within block-structured and procedural units of code, making PL/SQL a powerful transaction processing language.

Structured Query Language is language based on set theory, so it is all about manipulating sets of data. SQL consists of a relatively small number of main commands such as SELECT, INSERT, CREATE and GRANT; in fact, each statement accomplishes what might take hundreds of lines of procedural code to accomplish. That's one reason SQL-Based databases are so widely used.

Language categories:

1. Procedural Programming Languages:

These allow the programmer to define an ordered series of steps to follow in order to produce a result.

Examples: PL/SQL, C, Visual Basic, Perl, Ada.

2. Object-oriented Programming Languages:

Based on the concept of an object, which is a data structure encapsulated with a set of routines, called methods that operate on the data.

Examples: Java, C++, JavaScript, and sometimes Perl and Ada95.

3. Declarative Programming Languages:

These allow the programmer to describe relationships between variables in terms of functions or rules; the language executor (interpreter or compiler) applies some fixed algorithm to these to produce a result.

Examples: Logo, LISP and prolog.

4. Markup Languages:

These define how to add information into a document that indicates its logical components or that provides layout instructions.

Examples: HTML, XML

Features of PL/SQL:

1. Block Structure

PL/SQL is a block-structured language, meaning that programs can be divided into logical blocks, each containing the language resources that are logically required in that unit. Variables may be declared locally to the block in which they will be used and error conditions (known as Exceptions) may be handled specially within the block to which they apply.

2. Support for SQL

PL/SQL allows the use of all the SQL data manipulation, cursor control, and transaction control commands as well as all SQL functions and operators in a safe and flexible manner.

3. Higher productivity

PL/SQL is same in all environments. PL/SQL is same for all the tools of ORACLE like (SQL Forms, SQL Reportwriter etc).

4. Better performance

With PL/SQL an entire block of statements can be sent to ORACLE server at one time reducing the network and communication overhead between the client and server.

5. Portability

Applications written in PL/SQL are portable to any operating system and platform on which ORACLE runs.

6. Integration with ORACLE

PL/SQL supports all the SQL data types, which integrate PL/SQL with ORACLE data dictionary.

The %TYPE and %ROWTYPE attributes are used to declare variables, with the declarations based on the definitions of database columns and rows.

PL/SQL:

- ❖ Data centric and tightly integrated into the database
- ❖ Proprietary to Oracle and difficult to port to other database systems
- ❖ Data manipulation is slightly faster in PL/SQL than in Java
- ❖ Easier to use than Java (depending on your background)

What is PL/SQL and what is it used for?

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL's language syntax, structure and data types are similar to that of ADA. The PL/SQL language includes

object oriented programming techniques such as encapsulation, function overloading, and information hiding (all but inheritance). PL/SQL is commonly used to write data-centric programs to manipulate data in an Oracle database.

A PL/SQL block is a basic, unnamed unit of a PL/SQL program. It consists of a set of SQL or PL/SQL statements and it performs a single logical function. The declarative part is the first part of a PL/SQL block and is used for declaring objects such as variables, constants, cursors and exceptions. The executable part is the mandatory part of a PL/SQL block, and contains SQL and PL/SQL statements for querying and manipulating data. The exception-handling part is embedded inside the executable part of a block and is placed at the end of the executable part.

An anonymous PL/SQL block is the basic unnamed unit of a PL/SQL program. Procedures and functions can be compiled separately and stored permanently in an Oracle database, ready to be executed.

Delimiters:

Delimiters are symbols or compound symbols, which have a special meaning to PL/SQL. You will recognize many of them as operators from SQL.

1. Simple Symbols: These consist of one character.

+	addition operator
-	subtraction operator
/	division operator
=	relational operator
>	relational operator
<	relational operator
(expression or list delimiter
)	expression or list delimiter
;	statement terminator
%	attribute indicator
,	item separator
.	component selector
@	remote access indicator
'	character string delimiter
:	host variable indicator

Compound Symbols: These consist of two characters.

**	Exponential operator
<>	relational operator
!=	relational operator
^=	relational operator

<=	relational operator
:=	assignment operator
=>	association operator
..	range operator
	concatenation operator
<<	label delimiter
>>	label delimiter
--	comment indicator
/*	comment indicator
*/	comment indicator

Note: Spaces are not allowed between the two component characters of these compound symbols.

Literals:

A literal is an explicit numeric, string, or Boolean value not represented by an identifier. The numeric literal *147* and the Boolean literal *FALSE* are examples.

1. Numeric Literals

Two kinds of numeric literals can be used in arithmetic expressions: integers and real literals.

a) An integer literal: (is an optionally signed whole number without a decimal point)

Ex:

40	9	-14	0	32767
----	---	-----	---	-------

b) A real literal:

Ex:

6.6667	0.0	-12.0	3.1415 9	+8300.00	0.5	25.
--------	-----	-------	-------------	----------	-----	-----

Numeric literals cannot contain dollar signs or commas, both can be written using scientific notation. Simply suffix the number with an *E* (or *e*) followed by an optionally signed integer.

Ex:

2E5	1.0E-7	3.14159e 0	-1E38	-9.5e-3
-----	--------	---------------	-------	---------

2. Character Literals

A character literal is an individual character enclosed by single quotes. The characters can be Letters, numerals, spaces, and special symbols.

Ex: 'Z' '%' '7' ' ' ' ' 'z' '('

3. String Literals

A character value can be represented by an identifier or explicitly written as a string literal, which is a sequence zero or more characters enclosed by single quotes.

Ex: *'HELLO, WORLD!'* *'XYZ Corporation'* *' 10 -NOV-91'*
'He said "Life is like licking honey from a thorn."'
'\$1,000,000'

4. Boolean Literals

Boolean literals are the predefined values *TRUE*, *FALSE*, and *NULL* (which stand for a missing, unknown, or inapplicable value). Boolean literals are values, not strings. For example, *TRUE* is no less a value than the number 25.

Comments:

PL/SQL supports two comment styles: Single –Line and Multi-Line.

1. Single-Line

Single-line comments begin with a double hyphen(--) anywhere on a line and extend to the end of the line.

Ex:---*Welcome to griet*

SELECT PERCENTAGE FROM STUDENT --get current percentage

2. Multi – line

Multi –line comments begin with a slash –asterisk (*/**), end with an asterisk-slash (**/*), and can span multiple lines.

Ex:

BEGIN

.....

/ Compute a 15% bonus for top-rated employees. */*

IF rating > 90 THEN

*Bonus := salary * 0.15 /* bonus is based on salary */*

ELSE

Bonus :-0;

END If;

...

*Area := pi * radius**2;*

END;

BlockTypes PL/SQL Structure:

A block lets the user to group logically related declarations and statements. The declarations are local to the block and cease to exist when the block completes.

A PL/SQL has three parts:

- | | | |
|-------------------------|---|--|
| Declarative part | - | for declaring variables and objects |
| Executable part | - | for manipulation of data in the database through the declared variables and objects. |
| Exception handling part | - | for handling exceptions that are raised during the execution. |

Anonymous

Procedure

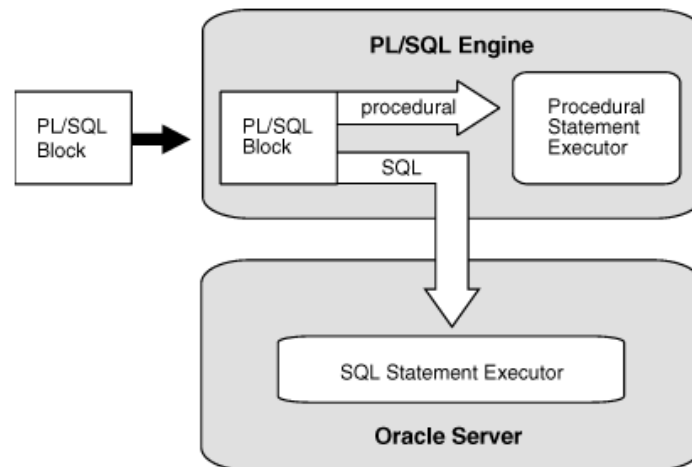
Function

[-- DECLARE declarations] BEGIN -- statements [EXCEPTION	PROCEDURE name IS BEGIN --statements	FUNCTION name RETURN datatype IS BEGIN --statements
--	---	--

PL/SQL Run Time Architecture:

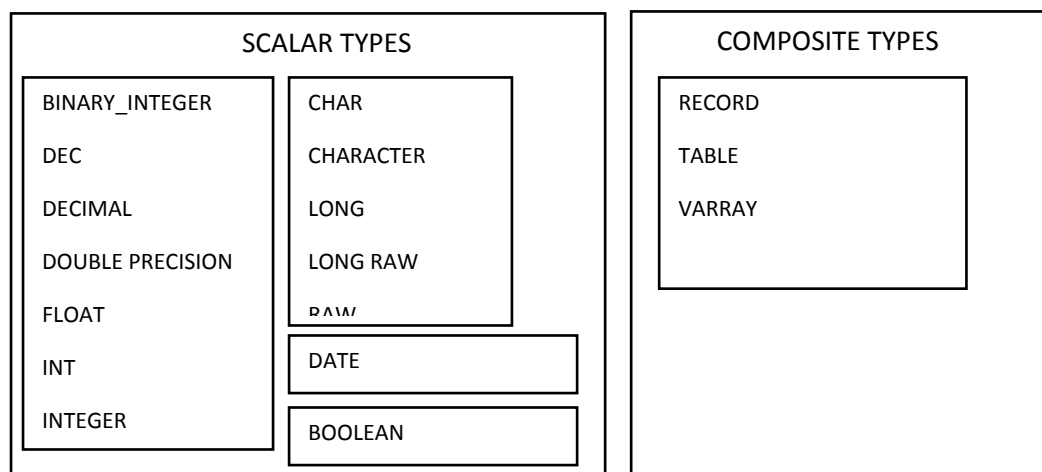
The PL/SQL compilation and run-time system is an engine that compiles and executes PL/SQL blocks and subprograms. The engine can be installed in an Oracle server or in an application development tool such as Oracle Forms.

In either environment, the PL/SQL engine accepts as input any valid PL/SQL block or subprogram. The following figure shows the PL/SQL engine processing an anonymous block. The PL/SQL engine executes procedural statements but sends SQL statements to the SQL engine in the Oracle database.



Datatypes:

PL/SQL supports four data type categories



1. Scalar Datatypes :

- ✧ They hold a single value
- ✧ Have no internal components
- ✧ Main data types are those that correspond to column types in oracle server tables
- ✧ Supports Boolean variables.

2.Composite Data types: A composite types has internal components that can be manipulated individually. Composite data types (Also known as collections) are of TABLE, RECORD, NESTED.

✧ Records allow groups of fields to be defined and manipulated in PL/SQL Blocks.

✧ `<variable><Tablename>%rowtype; rec emp%rowtype;`

3. Reference Data types: They hold values, called pointers, which designate other program items.

4. LOB Data types: They hold values called locations, specifying the location of large objects that are stored out of line.

Variables and Constants:

PL/SQL allows declaration of variables and constants.

Types of variables:

✧ **PL/SQL Variables:** All PL/SQL variables have a datatype, which specifies storage formats, constants, and valid range of values. PL/SQL supports four data type categories---

1. Scalar
2. Composite
3. Reference
4. LOB(large objects)

✧ **Non PL/SQL variables:** They include host language variables declared in pre-compiled programs, screen fields in Forms applications, and SQL*Plus or iSQL*Plus host variables.

Declaring Variables

```
JoinDate      DATE ;
Emp_count     SMALLINT := 0 ;
acct_id VARCHAR2(5) NOT NULL := 'AEOO1' ;
pi            CONSTANT REAL := 3.14159 ;
radius        REAL := 1 ;
area          REAL := pi * radius ** 2 ;
valid         BOOLEAN DEFAULT FALSE ;
tax           SMALLINT DEFAULT 92;
```

Control Structures:

Three basic control structures are

- Sequence
- Selection
- Iteration

1. Sequence structure simply executes a sequence of statements in the order in which they occur.
2. Selection structure test a condition, then executes the statements of that block.
3. Iterative structure executes a sequence of statements repeatedly as long as a condition holds true.

1. Conditional (Selection) control:

i) IF statements

a) IF – THEN

```
IF condition THEN
    Sequence of statements;
END IF;
```

Example:

```
DECLARE
    Eng number;
    Maths number;
    Science nuber;
    Total number;
    Per number;
BEGIN
    Eng:=&englishmarks;
    Maths:=&mathsmarks;
    Science:=sciencemarks
    Total:=eng+maths+science;
    Per:=total/3;
    If(per>=75)then
        Dbms_output.put_line('Distinction');
    Elself(per>-60and per<75) then
        Dbms_output.put_line('Credit');
    Elself(per>=40 and per<60)then
        Dbms_output.put_line('Pass');
    Else
        Dbms_output.put_line('Fail');
    End if;
```

End if;

End if;

End;

b) IF – THEN – ELSE

IF condition THEN

Sequence of statements1;

ELSE

Sequence of statements2;

END IF;

c) IF – THEN – ELSIF

IF condition1 THEN

Sequence of statements1;

ELSIF condition2 THEN

Sequence of statements2;

ELSE

Sequence of statements3;

END IF;

Example:

Declare

Eng number;

Maths number;

Science nuber;

Total number;

Per number;

Begin

Eng:=&englishmarks;

Maths:=&mathsmarks;

Science:=sciencemarks

Total:=eng+maths+science;

Per:=total/3;

If(per>=75)then

Dbms_output.put_line('Distinction');

Elsif(per>-60and per<75) then

```

        Dbms_output.put_line('Credit');
    Elsif(per>=40 and per<60)then
        Dbms_output.put_line('Pass');
    Else
        Dbms_output.put_line('Fail');
    End if;
End;
```

d) Nested IF - IF statement can be nested in another IF statement.

2. Iterative (loops) Control:

i) LOOP- allows the user to execute a sequence of statements multiple times.

```

    LOOP
        Sequence of statements;
    END LOOP;
Ex:  Declare
        X number:=2;
    Begin
        Loop
            Dbms_output_line(x);
            X:=X+2;
            Exit when X=22;
        End loop;
    End;
```

ii)FOR

It allows the user to specify a range of integers, then executes a sequence of statements once for each integer in the range.

```

        FOR <var> IN <lower>..<upper> LOOP
            Sequence of statements;
        END LOOP;
```

iii) WHILE

It associates a condition with a sequence of statements.

```

        WHILE condition LOOP
            Sequence of statements;
        END LOOP;
```

Composite Datatypes:

1. PL/SQL Records

A record is a variable that may contain a collection of separate values, each individually addressable.

Syntax:

```
TYPE <record_name> IS RECORD
(
    <field_name> { datatype | variable%TYPE | table.column%TYPE |
                                     table%ROWTYPE },
    <field_name> { datatype | variable%TYPE | table.column%TYPE |
                                     table%ROWTYPE },
    .....
);
```

Ex:

```
TYPE emp_rec IS RECORD
(
    num          emp.empno%TYPE not null := 7788,
    name         emp.ename%TYPE,
    salary       number(10,2)
);
```

Declaring variables of RECORD composite type

```
e1 emp_rec;
```

Storing data into a record variable

```
select empno, ename, sal into e1 from emp where empno = 7654;
```

Referring individual fields in a record

```
record_name.field_name;
dbms_output.put_line( e1.num );
```

Nested Record

A record contains one of its members, as a variable of another record.

Accessing the members of inner record

```
record_name . member_name . field_name
```

Ex:-

```
declare
```

```
TYPE empdept is record
```

```
(
    dno          dept.deptno%TYPE,
    dname dept.dname%TYPE,
    eno          emp.empno%TYPE,
    ename emp.ename%TYPE,
    salary       emp.sal%TYPE
);
ed1 empdept;
begin
    select emp.deptno, dname, empno, ename, sal into ed1 from emp, dept
    where emp.deptno = dept.deptno and empno = 7788;
    ed1.salary := ed1.salary + ed1.salary * 0.1;
    dbms_output.put_line( ' Deptno : ' || ed1.dno );
    dbms_output.put_line( ' Dname : ' || ed1.dname );
    dbms_output.put_line( ' Empno : ' || ed1.eno );
    dbms_output.put_line( ' Ename : ' || ed1.ename );
    dbms_output.put_line( ' Salary : ' || ed1.salary );
end;
```

2. PL/SQL Index By Tables

- PL/SQL tables are modeled as database tables
- Index (Primary) key is associated with them to have array-like access to rows
- Size can be dynamically increased
- Column can be of any scalar type, but primary key must be of type BINARY_INTEGER

Syntax:

```
TYPE <type_name> IS TABLE OF
    <column_type | variable%TYPE | table.column%TYPE> [NOT NULL]
    INDEX BY BINARY_INTEGER;
```

Declaring variables of PL/SQL tables

```
variable_name PL/SQL_table_name;
```

Referring rows in a PL/SQL table, using a primary key value or subscript

PL/SQL_table_name(primary_key_value) where primary_key_value belongs to type BINARY_INTEGER

Inserting and fetching rows in PL/SQL table

Ex:-

DECLARE

TYPE enametype IS TABLE OF emp.ename%TYPE NOT NULL
INDEX BY BINARY_INTEGER;

TYPE saltype IS TABLE OF emp.sal%TYPE NOT NULL
INDEX BY BINARY_INTEGER;

namelist enametype;
sallist saltype;
i BINARY_INTEGER := 1;
j number(2) := 1;

BEGIN

for emprec in (select ename, sal from emp) loop
namelist(i) := emprec.ename;
sallist(i) := emprec.sal;
i := i + 1;

end loop;

dbms_output.put_line(' Name Salary ');
while j < i loop
dbms_output.put_line(namelist(j) || ' ' || sallist(j));
j := j + 1;
end loop;

END;

/

Accessing remote database using PL/SQL

DECLARE

name varchar2(20);
salary number(8,2);

BEGIN

select ename, sal into name, salary from SCOTT.emp@INSLTD
where empno = 7788;
dbms_output.put_line(name || ' ' || salary);

END;

Exception Handling

An error condition during a program execution is called an exception. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition.

There are two types of exceptions based on the how the exception is raised:

1. Implicitly raised Exception
 - a) Pre-defined exceptions
 - b) Non-predefined exceptions
2. Explicitly raised Exception
 - a) User defined exceptions

1. Predefined Exceptions:

PL/SQL declares predefined exceptions globally in package STANDARD, which defines the PL/SQL environment. These need not be declared.

Examples:

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	ORA-06530	-6530	Your program attempts to assign values to the attributes of an uninitialized (atomically null) object.
CASE_NOT_FOUND	ORA-06592	-6592	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	-6531	Your program attempts to apply collection methods other than EXISTS to an uninitialized (atomically null) nested table or varray, or the program attempts to assign values to the elements of an uninitialized

			nested table or varray.
CURSOR_ALREADY_OPEN	ORA-06511	-6511	Your program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor FOR loop automatically opens the cursor to which it refers. So, your program cannot open that cursor inside the loop.
DUP_VAL_ON_INDEX	ORA-00001	-1	Your program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	ORA-01001	-1001	Your program attempts an illegal cursor operation such as closing an unopened cursor.
INVALID_NUMBER	ORA-01722	-1722	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.) This exception is also raised when the LIMIT-clause expression in a bulk FETCH statement

			does not evaluate to a positive number.
LOGIN_DENIED	ORA-01017	-1017	Your program attempts to log on to Oracle with an invalid username and/or password.
NO_DATA_FOUND	ORA-01403	100	A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table. SQL aggregate functions such as AVG and SUM always return a value or a null. So, a SELECT INTO statement that calls an aggregate function never raises NO_DATA_FOUND. The FETCH statement is expected to return no rows eventually, so when that happens, no exception is raised.
NOT_LOGGED_ON	ORA-01012	-1012	Your program issues a database call without being connected to Oracle.
PROGRAM_ERROR	ORA-06501	-6501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	-6504	The host cursor variable and PL/SQL cursor variable involved in an

			assignment have incompatible return types. For example, when an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
SELF_IS_NULL	ORA-30625	-30625	Your program attempts to call a MEMBER method on a null instance. That is, the built-in parameter SELF (which is always the first parameter passed to a MEMBER method) is null.
STORAGE_ERROR	ORA-06500	-6500	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533	Your program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532	Your program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
SYS_INVALID_ROWID	ORA-	-1410	The conversion of a

	01410		character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	ORA-00051	-51	A time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	ORA-01422	-1422	A SELECT INTO statement returns more than one row.
VALUE_ERROR	ORA-06502	-6502	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.)
ZERO_DIVIDE	ORA-01476	-1476	Your program attempts to divide a number by zero.

Syntax

BEGIN

sequence of statements

EXCEPTION

when—exception name then

sequence of statements;

END;

Example:

declare

price item.actualprice%type;

begin

select actual price into price from item where qty=888;

exception

when no_data_found then

dbms_output.put_line('item missing');

end;

/

Exception Handlers

When an exception is raised the control passed to the EXCEPTION part. If the EXCEPTION part handles the exception then the normal execution can be resumed otherwise the program terminates.

Ex 1:

DECLARE

m_ename emp.ename%TYPE;

m_sal emp.sal%TYPE;

BEGIN

select ename, sal into m_ename, m_sal from emp

where hiredate between '01-JAN-88' and '31-DEC-88' ;

EXCEPTION

WHEN no_data_found THEN

dbms_output.put_line('No records are fetched');

WHEN too_many_rows THEN

dbms_output.put_line('more than one row is fetched');

END;

/

‘WHEN_OTHERS’ExceptionHandler: Instead of defining a separate handler for every exception type, WHEN_OTHERS exception handler can be defined to handle all errors that are not handled by any other defined exception handler.

Ex 2:

```
BEGIN
    insert into emp (empno,ename,sal,deptno)
        values(7788,'RICHARDS',5000,10);
EXCEPTION
    WHEN dup_val_on_index THEN
        dbms_output.put_line('Record already exists');
    WHEN OTHERS THEN
        dbms_output.put_line('some error has occurred');
END;
```

RAISE statement:

To raise an exception even when the exception has not actually occurred, RAISE statement is used.

```
DECLARE
    acc_type char;
BEGIN
    acc_type := '&code' ;
    IF UPPER(acc_type) NOT IN ( 'S', 'C', 'F' ) THEN
        RAISE VALUE_ERROR ;
    ELSE
        dbms_output.put_line('Valid Account Type');
    END IF;
EXCEPTION
    WHEN value_error THEN
        dbms_output.put_line('Not a valid character');
    WHEN OTHERS THEN
        dbms_output.put_line('some error has occurred');
END;
```

2. Non-Predefined Exceptions:

To handle error conditions (typically ORA- messages) that have no predefined name, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A pragma is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package using the syntax:

```
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

Ex:

Declare

```
e_insert_excep EXCEPTION;
```

```
PRAGMA EXCEPTION_INIT(e_insert_excep,-014--);
```

```
BEGIN
```

```
INSERT INTO departments(department_id,department_name) VALUES(280,null);
```

```
EXCEPTION
```

```
WHEN e_insert_excep THEN
```

```
DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
```

```
END;
```

```
/
```

3. User Defined Exceptions

These exceptions are declared and defined by the user. They must be explicitly raised by the user using RAISE statement.

Declaring an exception

```
<userdefined_exception> EXCEPTION;
```

Ex:

```
DECLARE
```

```
in_sufficient_balance EXCEPTION;
```

```
balance number(8,2);
```

```
wamount balance%TYPE;
```

```
accno number(4);
```

```

BEGIN
    accno := &accountno;
    wamount := &amount ;
    select bal into balance from account where acc_no=accno ;
    IF balance – wamount < 500 THEN
        RAISE in_sufficient_balance;
    ELSE
        dbms_output.put_line('You can withdraw the amount '||wamount);
    END IF;
EXCEPTION
    WHEN in_sufficient_balance THEN
        dbms_output.put_line('You cannot withdraw that amount');
    WHEN OTHERS THEN
        dbms_output.put_line('some error has occurred');
END;

```

Ex:

```

DECLARE
    e_re emp%ROWTYPE;
    e1 EXCEPTION
    sal1 emp.salary%TYPE;
BEGIN
    SELECT salary INTO sal1 FROM emp WHERE ename = 'aarif';
    if sal1<5000 THEN
        RAISE e1;
    sal1:= 8500;
    UPDATE emp SET salary =sal1 WHERE ename ='aarif';
    END IF;
EXCEPTION
    WHEN no_data_found THEN
        RAISE_APPLICATION_ERROR (-20002,'aarif is not there. ');
    WHEN e1 THEN
        RAISE_APPLICATION_ERROR(-20002,'less salary');
END;

```

RAISE_APPLICATION_ERROR() :

To call RAISE_APPLICATION_ERROR, use the syntax

```
raise_application_error(  
    error_number, message[, {TRUE | FALSE}]);
```

where error_number is a negative integer in the range -20000 .. -20999 and message is a character string up to 2048 bytes long. If the optional third parameter is TRUE, the error is placed on the stack of previous errors. If the parameter is FALSE (the default), the error replaces all previous errors. RAISE_APPLICATION_ERROR is part of package DBMS_STANDARD, and as with package STANDARD, you do not need to qualify references to it.

An application can call raise_application_error only from an executing stored subprogram (or method). When called, raise_application_error ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

Ex:

Declare

```
s1 emp.sal%type;
```

begin

```
select sal into s1 from emp where ename='Sam';
```

```
if (no_data_found) then
```

```
    raise_application_error(20001,'sam is not there');
```

```
end if;
```

```
if(s1> 10000) then
```

```
    raise_application_error(20002,'sam if earning a lot');
```

```
end if;
```

```
update emp set sal=sal+500 where ename='Sam';
```

end;

Propagation rules for exceptions:

When an exception is raised, if PL/SQL cannot find a handler for it in the current block, the exception propagates. The exception searches in successive enclosing blocks until a handler is found or there are no more blocks to search.

Database Triggers

ORACLE allows you to define procedures that are implicitly executed when an INSERT, UPDATE, or DELETE statement is issued against the associated table. These procedures are called DATABASE TRIGGERS.

Triggers are similar to stored procedures discussed so far. A trigger can include SQL and PL/SQL statements to execute as a unit and can invoke other stored procedures. However procedures and triggers differ in the way that they are invoked.

While a procedure is explicitly executed by a user or an application, a trigger is implicitly executed (fired) by ORACLE when a triggering INSERT, UPDATE, or DELETE statement is issued, no matter when user is connected or what application is being used.

Triggers can only be defined on tables not views. However, triggers on the basetable(s) of a view are fired if an INSERT, UPDATE or DELETE statement is issued against a view.

Trigger – Levels

These triggers are written at three different levels

- Schema
- TABLE
- ROW

Schema level: Trigger as a trigger which is written at database level or user level. These triggers can be written by DBA only.

Table level Triggers are meant for providing security at object (Table) level.

Row level triggers are meant for validations (Any user can write table and ROW level triggers.)

Triggers are commonly used to :

1. Automatically generate derived column values
2. Prevent invalid transactions
3. Enforce complex security authorizations
4. Enforce complex business rules

Types of Triggers :

- ROW Triggers and Statement Triggers
- BEFORE or AFTER Triggers
- INSTRAD-OF triggers
- Triggers on System events and User Events.

Row Triggers:

A row trigger is fired each time the table is affected by triggering statement. For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement. If a triggering statement affects no rows, a row trigger is not executed at all.

A Row trigger is fired once for each row affected by the command. These triggers are used to check for the validity of the data in the triggering statements and rows affected.

Statement Triggers:

A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects (even no rows are affected). For example, if A DELETE statement deletes several rows from a table, a statement level DELETE triggering is fired only once, regardless of how many rows are deleted from the table.

Before and after triggers:

When defining a trigger, you can specify the TRIGGER TIMING whether the trigger action is to be executed before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers.

BEFORE and AFTER triggers fired by DML statements can be defined only on tables, not on views. However, triggers on the on the base tables of a view fired if an INSERT, UPDATE, or DELETE statement is issued against the view.

BEFORE and AFTER triggers fired by DDL statements can be defined only on the database or a schema, not on particular tables.

Syntax:

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
BEFORE | AFTER | INSTEAD OF
DELETE | [OR] INSERT | [OR] UPDATE [ OF <column> [ , <column> . . ] ]
ON <table>
[FOR EACH ROW [ WHEN <condition> ] ]
BEGIN
    --PL/SQL block
END;
```

before means trigger will fire before any DML statement

after means trigger will fire after any DML statement

For each row option creates a trigger that will be fired once for each row that is updated. This type of trigger is called ROW LEVEL TRIGGER.

If for each row is omitted, then trigger is executed once for that statement. (STATEMENT LEVEL TRIGGER)

When – This condition can be specified to further restrict the trigger is executed.

Restrictions may include the checks for old and new values.

Example:

Ex 1:

```
CREATE OR REPLACE TRIGGER upper_trigger
BEFORE INSERT OR UPDATE OF ENAME ON EMP
FOR EACH ROW
BEGIN
    :NEW.ENAME := UPPER( :NEW.ENAME );
    :NEW.JOB    := UPPER( :NEW.JOB );
END ;
```

INSTEAD OF Triggers:

These triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements. These triggers are called INSTEAD-OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of the triggering statement.

Parts of Triggers:

A trigger has 3 basic parts

1. A triggering event or statement
2. A trigger restriction
3. A trigger action

A triggering event or statement is the SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE, or DELETE statement for a specific table. A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire. The trigger action is not executed if the trigger restriction evaluates to FALSE.

A trigger action is the procedure (PL/SQL) block that contains the PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluates to true.

Advantages of database triggers:

- Data is generated on it's own
- Replicate table can be maintained
- To enforce complex integrity constraints
- To edit data modifications
- To auto increment a field.

Some key points about Triggers

- ❖ Triggers always have an action associated with it.
- ❖ Trigger gets fired automatically when an event occurs. Hence, a user need not to execute the Trigger code explicitly.
- ❖ Trigger is always transparent to the user.
- ❖ A table can have many triggers associated with it.
- ❖ In order to create triggers on a table, a user should have the create trigger privilege. If not so, then DBA can issue the following command to give the privilege.
SQL>GRANT CREATE TRIGGER TO USER;
- ❖ Triggers are more powerful than any referential integrity constraints.
- ❖ Triggers can be used to enforce business rules.
- ❖ Triggers cannot include Transaction processing Statements. (commit, rollback, savepoint)
- ❖ Only one trigger is possible for any type of DML statement. (INSERT, UPDATE, DELETE) per table.

Dropping Triggers:

Sql>Drop trigger triggername;

Drop command drops the trigger definition along with its associated permissions.

Disabling Triggers

Alter Trigger Triggername Disable;

Sql>Alter trigger triggl disable;

Disabling all the triggers associated with a table

Sql>alter table emp disable all triggers;

Enabling Triggers

Sql>alter tigger triggl enable;

Enabling triggers on the table level

Sql>alter table emp enable all triggers;

Viewing the information about triggers:

```
SQL>select * From user_objects where object_type='TRIGGER';
```

```
SQL>select      OBJECT_NAME,OBJECT_TYPE      From      user_objects      where  
object_type='TRIGGER';
```

Replace option retains the trigger permissions in order to drop and re-create a trigger with the same.

Additional Practices:

Example:

Ex 1:

```
CREATE OR REPLACE TRIGGER upper_trigger  
BEFORE INSERT OR UPDATE OF ENAME ON EMP  
FOR EACH ROW  
BEGIN  
    :NEW.ENAME := UPPER( :NEW.ENAME) ;  
    :NEW.JOB    := UPPER(:NEW.JOB);  
END;
```

Ex 2:

```
CREATE OR REPLACE TRIGGER emp_sal_trig  
AFTER DELETE OR UPDATE OF SAL ON EMP  
FOR EACH ROW  
BEGIN  
    IF DELETING OR UPDATING THEN  
        INSERT INTO TEMP VALUES( :old.emp, :old.ename, :old.sal);  
    END IF;  
END;
```

Ex 3:

```
CREATE OR REPLACE TRIGGER emp_instead_trig  
INSTEAD OF INSERT ON EMP  
FOR EACH ROW  
BEGIN  
    IF INSERTING THEN  
        INSERT INTO TEMP VALUES( :new.empno, :new.ename, :new.sal);  
    END IF;  
END;
```

Ex 4:

Prompt “Creating additional table SALS containing salary ranges...”

```
DROP TABLE SALS;
```

```
CREATE TABLE SALS
```

```
    (JOB VARCHAR(29) primary key,
```

```
    MINSAL NUBMER(7,2),
```

```
    MAXSAL NUMBER(7,2)
```

```
    );
```

```
INSERT INTO SALS VALUES('CLERK', 800, 1300);
```

```
INSERT INTO SALS VALUES('ANALYST', 3000, 3500);
```

```
INSERT INTO SALS VALUES('SALESMAN',1250,1600);
```

```
INSERT INTO SALS VALUES('MANAGER', 2450, 2975);
```

```
INSERT INTO SALS VALUES('PRESIDENT', 5000, 5500);
```

```
create or replace trigger check_salary_EMP
```

```
after insert or update of SAL, JOB on EMP
```

```
for each row
```

```
when (new.JOB != 'PRESIDENT')
```

```
declare
```

```
    minsal number;
```

```
    maxsal number;
```

```
begin
```

```
    select MINSAL, MAXSAL into minsal, maxsal from SALS
```

```
    where JOB = :new.JOB;
```

```
    if :new.SAL < minsal or :new.SAL > maxsal then
```

```
        raise_application_error(-20230, 'Salary has been exceeded');
```

```
    elsif :new.SAL < :old.SAL then
```

```
        raise_application_error(-20230, 'Salary has been decreased');
```

```
    elsif :new.SAL > 1.1*:old.SAL then
```

```
        raise_application_error(-20235, 'More than 10% salary increase');
```

```
    end if;
```

```
end;
```

CURSORS:

ORACLE uses private context work areas to execute SQL statements and store information. PL/SQL allows users to name the private work areas and access the stored information; these named private work areas are called cursors.

There are two types of CURSORS

- IMPLICIT CURSOR
- EXPLICIT CURSOR

1. Implicit Cursor:

Implicit cursors are declared by PL/SQL implicitly for all DML statements and for single row queries.

Implicit cursors are also known as SQL cursors as they are accessed by ORACLE only.

2. Explicit Cursor:

Explicit cursors are declared explicitly by the user. They are for queries only, and allow multiple rows to be processed from the query.

These are user-defined cursors that are defined in the declarative part of a PL/SQL block by naming it and specifying a query.

To control the cursor three commands are used:

OPEN	To initialize the cursor that identifies the active set.
FETCH	To retrieve the rows individually.
CLOSE	To release the cursor.

Handling Explicit Cursor

Explicit cursor is name used to refer to an area where you can place multiple rows retrieved by SELECT statement.

STEPS for controlling the cursor:

The following are the required steps to process an explicit cursor.

- Declare the cursor in declare section
- Open the cursor using OPEN
- Fetch rows from the cursor FETCH
- Close the cursor after the process is over using CLOSE

Declare a cursor

A cursor is declared in DECLARE section using CURSOR statement.

Syntax

Cursor <cursorname> is

Select <column(s)> from <tablename> where

<condition>;

Cursor Attributes

Cursor attributes allow to get information regarding cursor. For example, you can get the number of rows fetched so far from a cursor using ROWCOUNT

Syntax to access cursor attributes

Cursor_name%Attribute

The following is the list of available cursor attributes:

Attribute	Data type	Significance	Recommended time to use
FOUND	BOOLEAN	TRUE if most recent found a row in the table; Otherwise FALSE	After opening and fetching from the cursor but before closing it (will be NULL before first fetch)
NOT FOUND	BOOLEAN	This is just logical inverse of FOUND	Same as above
ROWCOUNT	BOOLEAN	Number of rows fetched so far	Same as above (except it will be zero before the first fetch)
ISOPEN	BOOLEAN	Is evaluates to TRUE, if the cursor is opened	Rarely used cursor attribute

Ex:

SET SERVEROUTPUT ON – SQL*plus Environment command

DECLARE

Cursor emp_cur is

Select empno, ename, job, sal from EMP where empno >= 7521;

Emp_rec_emp_cur%rowtype;

BEGIN

/* open the cursor */

Open emp_cur;


```

        /* fetch a record from cursor*/
        FETCH emp_cur into emp_rec;
        DBMS_OUTPUT.PUT_LINE(emp_rec.empno || emp_rec.ename|| emp_rec.sal);
        CLOSE emp_cur; -- closing the cursor
END;

```

Note: This program reads and prints only one record from the cursor

Ex:

DECLARE

Cursor emp_cur is

Select empno, ename, job, sal from EMP where empno>= 7521;

Emp_rec emp_cur%rowtype;

BEGIN

/* open the cursor */

Open emp_cur;

/* fetch all the records of the cursor one by one */

LOOP

FETCH emp_cur into emp_rec;

/* Exit loop if reached end of cursor NOTFOUND is the cursor attribute*/

exit when emp_cur%NOTFOUND;

DBMS_OUTPUT.PUT_LINE (emp_rec.empno || emp_rec.ename|| emp_rec.sal);

ENDLOOP;

--closing the cursor

CLOSE emp_cur;

END;

/

Implicit CURSOR

PL/SQL provides some attributes, which allow us to evaluate what happened when the implicit cursor was last used.

They are

SQL%ROWCOUNT - no. of rows processed by the SQL command.

SQL%FOUND - TRUE if at least one row was processed.

SQL%NOTFOUND - TRUE if no rows were processed.

Ex 1:

```

declare
    total number;
begin
    update emp set sal = sal + sal * 0.1
    where deptno = 30;
    total := SQL%ROWCOUNT;
    insert into temp values ( total, sysdate ) ;
    dbms_output.put_line( total || 'rows updated' );
end;

```

Ex 2:

```

begin
    update emp set comm = sal * 0.1
    where deptno = 30
    and sal > ( select avg(sal) from emp );
    if SQL%FOUND then
        commit;
        dbms_output.put_line( 'Transaction committed' );
    else
        rollback;
        dbms_output.put_line( 'No records found' );
    end if;
end;

```

Ex:

```

declare
    cursor c1 is select empno,ename,salary from emp order by empno desc;
    mrec emp%rowtype;
begin
    open c1;
    for i in 1..2
    loop
        fetch c1 into mrec.empno,mrec.ename,mrec.salary;
        insert into empdup(empno,ename,salary) values(mrec.empno,mrec.ename,
mrec.salary);
    end loop;
end;

```

```

        exit when c1%notfound;
    end loop;
    close c1;
End:
/
Ex:
DECLARE
    cursor c2 is select empno,ename,salary from emp;
    mrec emo%rowtype;
BEGIN
    open c2;
    loop
        fetch c2 into mrec.empno,mrec.ename,mrec.salary;
        exit when c2%rowcount=&EnterNoOfRows;
        insert into empdup(empno,ename,salary)
        values(mrec.empno,mrec.ename,mrec.salary);
    end loop;
    close c2;
END;

/* Program to extract top 5 records from the students and insert them into the toppers table */
DECLARE
    cursor c1 is select name, grade from student order by marks desc;
    mrec student%rowtype;
BEGIN
    open c1;
    loop
        fetch c1 into mrec.name,mrec.grade;
        exit when c1%rowcount=5;
        insert into toppers (name,grade) values(mrec.name,mrec.grade);
    end loop;
    close c1;
    dbms_output.put_line('Successfully inserted rows in Toppers table');

```

END;

Parameterized Cursors:

Cursors can take parameters. A cursor parameter can appear in a query.

CURSOR [(parameter [, parameter, ...])] IS QUERY;

Ex:

declare

 m_sal emp.sal%TYPE;

 m_empno emp.sal%TYPE;

 budget number := 2000;

 amount number;

 x number;

 CURSOR c1(dno number) is

 select empno, sal from emp where deptno = dno;

begin

 x := 10;

 OPEN c1(x);

 loop

 FETCH c1 into m_empno, m_sal;

 exit when c1%NOTFOUND;

 if budget > 0 then

 amount := m_sal * 0.2;

 m_sal := m_sal + amount;

 update emp set sal = m_sal

 where empno = m_empno;

 dbms_output.put_line(m_empno || ' ' || m_sal);

 budget := budget - amount;

 else

 dbms_output.put_line(m_empno || ' no hike ');

 end if;

 end loop;

 CLOSE c1;

end;

Cursor FOR Loop

```
declare
    CURSOR c1 is
        select ename, sal from emp where deptno = 20;
begin
    FOR emprec IN c1 loop
        insert into temp values (emprec.ename,emprec.sal);
    end loop;
end;
```

PROCEDURES AND FUNCTIONS

- Procedure is a subprogram which contains of a set of sql statement.
- Procedures are not very different from functions.
- A procedure or function is a logically grouped set of SQL and PL/SQL statements that perform a specific task.
- A stored or function is a named pl/sql code block that have been compiled and stored in one of the oracle engine's system tables.
- To make a procedure or function dynamic either of them can be passed parameters before execution.
- A procedure or function can then change the way it works depending upon the parameters passed prior to its execution.
- Procedures and function are made up of a declarative part, an executable part and an optional exception-handling part.
- A declaration part consists of declarations of variables.
- A executable part consists of the logic i.e. sql statements..... and exception handling part handles any error during run-time.

The oracle engine performs the following steps to execute a procedure or function.... Verifies user access, verifies procedure or function validity and executes the procedure or function.

Some of the advantages of using procedures and functions are: security, performance, memory allocation, productivity, integrity. Most important is the difference between procedures and the functions: a function must return a value back to the caller. A function can return only one value to the calling pl/sql block. By defining multiple out parameters in a procedure, multiple values can be passed to the caller. The out variable being

global by nature, its value is accessible by any pl/sql code block including the calling pl/sql block.

Syntax for stored procedure:

```
CREATE OR REPLACE PROCEDURE [schema] procedurename (argument { IN, OUT, IN
OUT} data type,...) {IS, AS}
variable declaration; constant declarations; BEGIN
pl/sql subprogram body;
EXCEPTION
exception pl/sql block;
END;
```

IN: specifies that a value for the argument must be specified when calling the procedure or function.

argument: is the name of an argument to the procedure or function. Parentheses can be omitted if no arguments are present.

OUT: specifies that the procedure and that the procedure passes a value for this argument back to its calling environment after execution. By default it takes IN. data type: is the data type of an argument.

Procedure Using No Argument .. And Using Cursor

Ex:

```
CREATE OR REPLACE PROCEDURE P2 IS
cursor cur1 is select * from emp;
begin
for erec in cur1
loop
dbms_output.put_line(erec.ename);
end loop;
end;
```

Procedure Using Argument:

Ex:

```
CREATE OR REPLACE PROCEDURE ME( X IN NUMBER ) IS
BEGIN
dbms_output.put_line(x*x);
end;
```

```
sql> exec me(3);
```

FUNCTION using argument:

```
CREATE OR REPLACE FUNCTION RMT(X IN NUMBER ) RETURN NUMBER IS
```

```
BEGIN
```

```
dbms_output.put_line(x*x);
```

```
--return (x*x);
```

```
End;
```

Executing the function:

```
begin
```

```
dbms_output.put_line(rmt(3));
```

```
end;
```

IN and OUT procedure example

```
create or replace procedure test (a number, b out number) is
```

```
identify number;
```

```
begin
```

```
select ordid into identify from item where
```

```
itemid= a;
```

```
if identity < 1000 then
```

```
b := 100;
```

```
endif;
```

```
end;
```

in out mode example

```
create or replace procedure sample ( a in number, b in out nnumber) is
```

```
identiy number;
```

```
begin
```

```
select ordid, prodid into identity, b from item where itemid =a;
```

```
if b<600 then
```

```
b :=b+100;
```

```
end if;
```

```
end;
```

```
--now procedure is called by passing parameter
```

```
declare
```

```
a number;
```

```

b number;
begin
sample(3000,b)
dbms_output.put_line(1th value of b is 11 b);
end;

```

Difference between Procedures and Functions

- A function must return a value back to the caller.
- A function can return one value to the calling pl/sql block.
- By defining multiple out parameters in a procedure, multiple values can be passed to the caller.

Syntax for stored function:

```

CREATE OR REPLACE FUNCTION functionname (argument datatype, ...)
RETURN datatype IS
variable declarations;
BEGIN
pl/sql subprogram body;
EXCEPTION
exception pl/sql block;
END;

```

Creating a function:

Ex: Function to display salary of employees based on empno.

```

create or replace function f1(mempno number) return number is msal number(5);
BEGIN
    select salary into msal from emp where empno=mempno;
    return(msal);
END;

```

Calling a function

```

BEGIN
dbms_output.put_line(f1(3));
END;
/

```

Ex:

```

CREATE OR REPLACE FUNCTIONRMY(X NUMBER)

```


RETURN NUMBER IS

BEGIN

Dbms_output.put_line(x*x);

END;

/

Ex: Function to update salary by 5000 based on empno

Create or replace function f2(mempno number) return number is msal number(5);

Begin

select salary into msal emp where empno=mempno;

msal:=msal+5000;

update emp set salary = msal where empno=mempno;

return (msal);

END;

/

Ex: Function to display the 'day' based on the choice

create or replace function f3(mday number) return varchar is display varchar(10);

BEGIN

if mday = 1 then

display:= 'Sunday';

elsif mday=2 then

display:= 'Monday';

elsif mday= 3 then

display:= 'Tuesday';

elsif mday= 4 then

display:= 'Wednesday';

elsif mday= 5 then

display:= 'Thursday';

elsif mday= 6 then

display:= 'Friday';

elsif mday= 7 then

display:= 'Saturday';

end if;

```
        return (display);  
END;  
/
```

Packages

Packages are containers of related procedures, functions and variables. When we create different procedures and functions, they are stored as independent objects.

To group these objects together, we are creating packages.

Each package contains the following parts:

- Package specification
- Package body

Package Specification

It contains all declarations for variable, cursor, procedures and functions that are to be made public. All public objects of package are visible outside the package.

Syntax

```
CREATE OR REPLACE PACKAGE package_name  
IS /* declare public objects of package */  
END;
```

Package body

It defines all the objects of the package. Objects declared in the specification are called as public objects directly and the objects directly defined in the body without declaring in the specification, are called as PRIVATE members.

Syntax

The package body

```
CREATE or REPLACE PACKAGE BODY package_name  
IS  
[ declarations of variables and types ]  
[ specifications and SELECT statement of cursors ]  
[ specification and body of modules ]  
[ BEGIN executable statements ]  
[ EXCEPTION exception handlers ]  
END [ package_name ];
```

In the body you can declare other variables, but you do not repeat the declarations in the specification.

The body contains the full implementation of cursors and modules. In the case of cursor, the package body contains both the specification and SQL statement for the cursor. In the case of a module the package body contains both the specification and body of the module.

The BEGIN keyword indicates the presence of an execution or initialization section for the package. This section can also optionally include an exception section.

As with a procedure or function, you can add the name of the package, as a label, after the END keyword in the specification and package.

The body of a package can contain

- Procedures declared in the package specification
- Functions declared in the package specification
- Definitions of cursors declared in the package specification
- Local procedures and functions, not declared in the package specification
- Local variables

Ex:

```
CREATE OR REPLACE PACKAGE SAMPLEPAK
IS
    PROCEDURE PROC1( N NUMBER, N1 OUT NUMBER);
    FUNCTION FUN1 (N NUMBER) RETURN NUMBER;
    END;
/
CREATE OR REPLACE PACKAGE BODY SAMPLEPACK
IS
    PROCEDURE PROC1(N NUMBER, N1 OUT NUMBER)
    IS
        BEGIN PROC1;
            N1 :=N*5;
        END PROC1;
    FUNCTION FUN1(N NUMBER) RETURN NUMBER
    IS
        N1 MEMBER;
```

```

BEGIN
    N1 := N*2;
    RETURN N1;
END FUN1;
END SAMPLEPAK;
/

```

Execution

```

VARIABLE N NUMBER
EXECUTE SAMPLEPAK.PROC1(5,:N)
    PRINT N
EXECUTE :N :=SAMPLEPAK.FUN1(4)
PRINT N

```

Ex: Program to define private member

```

CREATE OR REPLACE PACKAGE SAMPLEPAK
IS
    PROCEDURE PROC1(N NUMBER, N1 OUT NUMBER);
    FUNCTION FUN1(N NUMBER) RETURN NUMBER;
END;
/

CREATE OR REPLACE PACKAGE BODY SAMPLEPAK
IS
    PROCEDURE TEST  --private member definition
    IS
        BEGIN
            DBMS_OUTPUT.PUT_LINE('I AM A PRIVATE MEMBER');
        END;

    PROCEDURE PROC1(N NUMBER, N1 OUT MEMBER)
    IS
        BEGIN
            TEST;      --private member called
            N1 :=N*5;

            RND PROC1;
        FUNCTION FUN1(N NUMBER) RETURN NUMBER

```

```

IS
    N1 NUMBER;
BEGIN
    N1 :=N*2;
    RETURN N1;
END FUN1;
END SAMPLEPAK;
/

```

Execution

```

VARIABLE N NUMBER
EXECUTE SAMPLEPAK.PROC1(5,:N)
PRINT N
EXECUTE :N :=SAMPLEPAK.FUN1(4)
PRINT N

```

Note: A private member cannot be accessed by referring package object. They called only through public members of the package object.

Advantages of packages:

Packages offer several advantages Modularity, easier application design, information hiding, added functionality and better performance.

Calling packaged subprograms:

Packaged subprograms must be referenced using dot notation, as the following example shows:

```
emp_actions.hire_employee(name,title, ...);
```

This tells the PL/SQL compiler that hire_employee is found in the package emp_actions.

A packaged subprogram can be called from a database trigger, another stored subprogram, an ORACLE Precompiler application, an ORACLE tool such as SQL*Plus.

Calling from another stored subprogram:

A stored subprogram can call a packaged subprogram. For instance the following call to the packaged procedure hire_employee might appear in a stand-alone subprogram.

```
emp_actions.hire_employee(name,title, ...);
```

Calling from an ORACLE tool: A stored program can be called from SQL*Plus, SQL*Forms, and SQL*DBA.

TASK - 10

1. Write a PL/SQL code to retrieve the employee name, join date and designation from employee database of an employee whose number is input by the user.

Program:

```
/*Employee details*/

DECLARE

v_name varchar2(25);

v_joindate date;

v_dsgn employees.job_id%type;

BEGIN

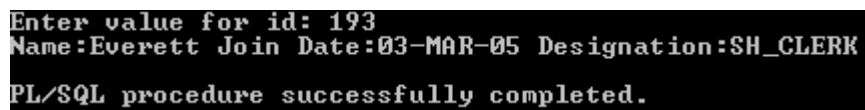
select last_name,hire_date,job_id into v_name,v_joindate,v_dsgn from employees where
employee_id=&id;

DBMS_OUTPUT.PUT_LINE('Name:'||v_name||' Join Date:'||v_joindate||'
Designation:'||v_dsgn);

END;

/
```

Output:

A screenshot of a terminal window showing the output of a PL/SQL procedure. The text is as follows:
Enter value for id: 193
Name:Everett Join Date:03-MAR-05 Designation:SH_CLERK
PL/SQL procedure successfully completed.
The text is displayed in a monospaced font on a dark background.

2. Write a PL/SQL code to calculate tax for an employee of an organization.

Program:

```
/*Calculate Tax*/

DECLARE

v_sal number(8);

v_tax number(8,3);
```

```

v_name varchar2(25);

BEGIN

select salary,last_name into v_sal,v_name from employees where employee_id=&id;

if v_sal<10000 then

    v_tax:=v_sal*0.1;

elseif v_sal between 10000 and 20000 then

    v_tax:=v_sal*0.2;

else

    v_tax:=v_sal*0.3;

END IF;


DBMS_OUTPUT.PUT_LINE('Name:'||v_name||' Salary:'||v_sal||'Tax:'||v_tax);

END;

/

```

Output:



```

Enter value for id: 101
Name:Kochhar Salary:17000 Tax:3400
PL/SQL procedure successfully completed.

```

3. Write a PL/SQL program to display top 10 employee details based on salary using cursors.

Program:

```

/*Top 10 salary earning employee details*/

DECLARE

cursor c_emp_cursor is select employee_id, last_name, salary from employees order by
salary desc;

v_rec c_emp_cursor%rowtype;

v_i number(3):=0;

BEGIN

open c_emp_cursor;

loop

```



```

v_i:=v_i+1;
fetch c_emp_cursor into v_rec;
exit when v_i>10;
DBMS_OUTPUT.PUT_LINE(v_rec.employee_id||' '||v_rec.last_name||' '||v_rec.salary);
END LOOP;
close c_emp_cursor;
END;
/

```

Output:

```

100 King 24000
101 Kochhar 17000
102 De Haan 17000
145 Russell 14000
146 Partners 13500
201 Hartstein 13000
108 Greenberg 12008
205 Higgins 12008
147 Errazuriz 12000
168 Ozer 11500

PL/SQL procedure successfully completed.

```

4. Write a PL/SQL program to update the commission values for all employees with salary less than 5000 by adding 1000 to existing employees.

Program:

```

/*Updation*/
declare
cursor c_emp is select salary,commission_pct from employees;
v_emp c_emp%rowtype;
v_temp number(7,2);
v_temp1 number;
BEGIN
open c_emp;
loop
fetch c_emp into v_emp;
exit when c_emp%notfound;

```

```

v_temp1:=v_emp.commission_pct;
v_temp:=(v_emp.salary*v_emp.commission_pct)+1000;
v_temp:=v_temp/v_emp.salary;
if(v_emp.salary<5000) then
update employees set commission_pct=v_temp where employee_id=v_temp.employee_id;
end if;
DBMS_OUTPUT.PUT_LINE('Commission % updated from '||v_temp1||' to '||v_temp);
end loop;
END;
/

```

Output:

```

Commission % updated from .2 to .3
Commission % updated from .15 to .29
Commission % updated from .15 to .29
Commission % updated from .1 to .24
Commission % updated from .3 to .39
Commission % updated from .25 to .36
Commission % updated from .2 to .32
Commission % updated from .2 to .32
Commission % updated from .15 to .29
Commission % updated from .1 to .24
Commission % updated from to
Commission % updated from to

```

TASK – 11

1. Write a trigger on the employee table which shows the old values and new values of ename after any updations on ename on Employee table.

Program:

```
create or replace trigger t_emp_name after update of last_name on salary_table FOR EACH ROW
```

```
begin
```

```
DBMS_OUTPUT.PUT_LINE('Name updated from '||:OLD.last_name||' to '||:NEW.last_name);
```

```
END;
```

```
/
```

Output:

```
SQL> @C:/Users/Kshore/Plsql/temp.sql
Trigger created.

SQL> update salary_table set last_name='Smith' where employee_id=198;
Name updated from OConnell to Smith
1 row updated.

SQL> update salary_table set last_name='John' where employee_id=157;
Name updated from Sully to John
1 row updated.

SQL> update salary_table set last_name='Mike' where employee_id=201;
Name updated from Hartstein to Mike
1 row updated.
```

2. Write a PL/SQL procedure for inserting, deleting and updating in employee table.

Program:

```
create or replace procedure proc_dml (p_id emp.employee_id%type, p_sal number,p_case number)
```

```
is
```

```
BEGIN
```

```
case p_case
```

```
when 1 then
```

```
DBMS_OUTPUT.PUT_LINE('Insertion...');
```

```

        insert into emp(employee_id,last_name,email,hire_date,job_id)
values(p_id,'Franco','FJames','12-JAN-02','ST_CLERK');

    when 2 then

        DBMS_OUTPUT.PUT_LINE('Deletion...');

        delete from emp where employee_id=p_id;

    when 3 then

        DBMS_OUTPUT.PUT_LINE('Updation...');

        update emp set salary=p_sal where employee_id=p_id;

    end case;

DBMS_OUTPUT.PUT_LINE('DML operation performed on '||SQL%rowcount||' rows');

END;

/

DECLARE

v_id employees.employee_id%type:=&id;

v_sal employees.salary%type:=&sal;

v_case number:=&case1or2or3;

begin

proc_dml(v_id,v_sal,v_case);

END;

/

```

Output:

```

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for employee_id: 210
Enter value for salary: 20000
Enter value for case1or2or3: 1
Insertion...
DML operation performed on 1 rows

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for employee_id: 101
Enter value for salary: 21000
Enter value for case1or2or3: 3
Updation...
DML operation performed on 1 rows

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for employee_id: 210
Enter value for salary: 20000
Enter value for case1or2or3: 2
Deletion...
DML operation performed on 2 rows

PL/SQL procedure successfully completed.

```

3. Write a PL/SQL function that accepts department number and returns the total salary of the department.

Program:

create function func_dept (p_dept number) return number is

v_total number;

BEGIN

select sum(salary) into v_total from employees where department_id=p_dept;

return v_total;

END;

/

DECLARE

v_dept number:=&department_id;

v_total number;

BEGIN

v_total:=func_dept(v_dept);

DBMS_OUTPUT.PUT_LINE('Total salary in Department '||v_dept||' is '||v_total);

END;

/

Output:

```

SQL> @C:/Users/Kshore/Plsql/temp1.sql
Function created.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for department_id: 40
Total salary in Department 40 is 6500

PL/SQL procedure successfully completed.

```

Task-12

1. Write a PL/SQL program to handle predefined exceptions.

Program:

```

declare
v_id number(6):=&employee_id;
v_sal employees.salary%type;
v_name employees.last_name%type;
v_job employees.job_id%type;
begin
select last_name, salary into v_name, v_sal from employees where employee_id=v_id;
DBMS_OUTPUT.PUT_LINE(v_name||q['s salary is '||v_sal);
select job_id into v_job from employees where last_name=v_name;
DBMS_OUTPUT.PUT_LINE(v_name||q['s job is '||v_job);
EXCEPTION
    when no_data_found then
        DBMS_OUTPUT.PUT_LINE('No employee with ID:'||v_id);
    when too_many_rows then
        DBMS_OUTPUT.PUT_LINE('Many employees with Name:'||v_name);

```

when others then

DBMS_OUTPUT.PUT_LINE('Some other error occurred');

end;

/

Output:

```
SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for employee_id: 101
Kochhar's salary is 17000
Kochhar's job is AD_UP

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for employee_id: 100
King's salary is 24000
Many employees with Name:King

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for employee_id: 210
No employee with ID:210

PL/SQL procedure successfully completed.
```

2. Write a PL/SQL program to handle user defined exception.

Program:

DECLARE

v_dept number:=&department_id;

e_nodept exception;

BEGIN

update employees set salary=salary+1050 where department_id=v_dept;

IF SQL%notfound then

raise e_nodept;

ELSE

DBMS_OUTPUT.PUT_LINE(SQL%rowcount||' rows updated');

END IF;

EXCEPTION

when e_nodept then

DBMS_OUTPUT.PUT_LINE('No Department with ID:'||v_dept)

END;

/

Output:

```
SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for department_id: 500
No Department with ID:500

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for department_id: 40
1 rows updated

PL/SQL procedure successfully completed.
```

3)Write a PL/SQL code to create

a. Package specification.

Program:

create or replace package pack_dml is

 procedure proc_dml(p_id number,choice number);

END pack_dml;

/

Output:

```
SQL> @C:/Users/Kshore/Plsql/test1.sql
Package created.
```

b. Package body for the insert, retrieve, update and delete operations on student table.

Program:

create or replace package body pack_dml is

 procedure proc_dml(p_id number,choice number) is

 v_name varchar2(20);

 v_total number;

 BEGIN

 case choice

 when 1 then

 DBMS_OUTPUT.PUT_LINE('Insertion...');

 insert into student values(p_id,'Franco',90);

 when 2 then


```

        DBMS_OUTPUT.PUT_LINE('Deletion...');
        delete from student where sid=p_id;
        when 3 then
        DBMS_OUTPUT.PUT_LINE('Updation...');
        update student set total=total+1 where sid=p_id;
        when 4 then
        select sname,total into v_name,v_total from student where sid=p_id;
        DBMS_OUTPUT.PUT_LINE('Total marks of '||v_name||' is '||v_total);
    end case;

    DBMS_OUTPUT.PUT_LINE('DML operation performed on '||SQL%rowcount||
    rows');
END proc_dml;
END pack_dml;
/

BEGIN
pack_dml.proc_dml(&StudentID,&choice1or2or3or4);
END;
/

```

Output:

```

SQL> select * from student;

```

SID	SNAME	TOTAL
10	John	90
20	Mike	92
30	Smith	69
40	Robert	80
50	Michael	73

```

SQL> @C:/Users/Kshore/Plsql/temp.sql
Package body created.
SQL>

```

```
SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for studentid: 60
Enter value for choice1or2or3or4: 1
Insertion...
DML operation performed on 1 rows

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for studentid: 20
Enter value for choice1or2or3or4: 2
Deletion...
DML operation performed on 1 rows

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for studentid: 30
Enter value for choice1or2or3or4: 3
Updation...
DML operation performed on 1 rows

PL/SQL procedure successfully completed.

SQL> @C:/Users/Kshore/Plsql/test1.sql
Enter value for studentid: 10
Enter value for choice1or2or3or4: 4
Total marks of John is 90
DML operation performed on 1 rows

PL/SQL procedure successfully completed.
```