ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. [Computer Science and Engineering]

IV – SEMESTER

CSCP408 – Database Management Systems Lab

CSCP408 – DATABASE MANAGEMENT SYSTEMS LAB

CONTENTS

A1 Batch Dr. S. G. Santhi

A2 Batch Dr. R. Saminathan

1. Data Definition Language Statements

a. Without Constraint

Aim:

To work with DDL queries like create, alter, modify, rename and drop without any constraints.

Concept:

SQL	Structured Query Language (SQL) is a standard computer language for relational database management and data manipulation. SQL is used to query, insert, update and modify data.
DDL	A data definition language or data description language (DDL) is syntax similar to a computer programming language for defining data structures, especially database schemas.
Create	The *CREATE* TABLE statement is used to *create* a table in a database. Create table table_name (colname datatype, colname datatype……);
Alter	The SQL *ALTER TABLE* command is used to add, delete or modify columns in an existing table. Alter table table_name add column_name data_type; Alter table table_name drop column column_name; Alter table table_name modify column_name new_data_type;
Rename	Rename statement is used to rename the existing column name and table name. Alter table table_name rename old_col_name to new_col_new; alter table old_table_name rename to new_table_name;
Drop	The *DROP TABLE* statement removes a table added with the CREATE TABLE statement i.e., the entire schema of the table created is removed. Drop table table_name;

Queries:

Create:

Creation of a table:

```
SQL> create table std
     (
     sname varchar2(10),
     roll number(5),
     dept varchar(3),
     m1 number(3),
     m2 number(3),
     m3 number(3)
     );
```

Table created.

Describe the table std:

SQL> desc std;

Name	Null?	Type
SNAME		VARCHAR2(10)
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)

Alter - ADD:

Add columns to the table:

```
SQL>  alter table std add
      (
      total number(4),
      avg number(4),
      result varchar2(6)
      );
```

Table altered.

SQL> desc std;

Name	Null?	Type
SNAME		VARCHAR2(10)
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)
TOTAL		NUMBER(4)
AVG		NUMBER(4)
RESULT		VARCHAR2(6)

Alter - Modify:

Modify the data type of the field of the table:

SQL> alter table std modify sname varchar2(25);

Table altered.

SQL> desc std;

Name	Null?	Type
SNAME		VARCHAR2(25)
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)
TOTAL		NUMBER(4)
AVG		NUMBER(4)
RESULT		VARCHAR2(6)

Alter - Rename:

Rename the column name of the table:

SQL> alter table std rename total to sum;

Name	Null?	Type
SNAME		VARCHAR2(25)
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)
SUM		NUMBER(4)
AVG		NUMBER(4)
RESULT		VARCHAR2(6)

Renaming the Table:

SQL> alter table std rename to stud;

Table altered.

SQL> desc stud;

Name	Null?	Type
SNAME		VARCHAR2(25)
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)
TOTAL		NUMBER(4)
AVG		NUMBER(4)
RESULT		VARCHAR2(6)

Alter – Drop column:

Remove a column from the table:

SQL> Alter table std drop column (SNAME); **[Works with the version 8.1 onwards]**

Name	Null?	Type
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)
TOTAL		NUMBER(4)
AVG		NUMBER(4)
RESULT		VARCHAR2(6)

Table altered.

Drop - Table:

Remove the table from the database:

SQL> drop table stud;

Table dropped.

Result:

Thus, different DDL queries like create, alter, modify, rename and drop without any constraints is successfully executed and verified.

1. Data Definition Language Statements

b. With Constraint

Aim:

To work with DDL queries like create, alter, modify, rename and drop with any constraints like

- Not null
- Unique
- Primary key
- Foreign key
- Check
- Default

Concept:

Constraints	Constraints enable the RDBMS to enforce the integrity of the database automatically, without needing to create any triggers, rule or defaults.
Entity integrity	The **entity integrity** constraint states that primary keys can't be null. There must be a proper value in the primary key field. This is because the primary key value is used to identify individual rows in a table. If there were null values for primary keys, it would mean that we could not identify those rows.
Referential integrity	When one table has a foreign key to another table, the concept of referential integrity states that one may not add a record to the table that contains the foreign key unless there is a corresponding record in the linked table.
Not null	Constraint enforces that the column will not accept null values. The not null constraints are used to enforce domain integrity, as the check constraints
Check	Constraint is used to limit the values that can be placed in a column. The check constraints are used to enforce domain integrity.
Unique	Constraint enforces the uniqueness of the values in a set of columns, so no duplicate values are entered. The unique key constraints are used to enforce entity integrity as the primary key constraints
Primary key	Constraint is a unique identifier for a row within a database table. Every table should have a primary key constraint to uniquely identify each row and only one primary key constraint can be created for each table. The primary key constraints are used to enforce entity integrity
Foreign key	Constraint prevents any actions that would destroy link between tables with the corresponding data values. A foreign key in one table points to a primary key in another table. Foreign keys prevent actions that would leave rows with foreign key values when there are no primary keys with that value. The foreign key constraints are used to enforce referential integrity
Default	The *DEFAULT constraint* is used to insert a *default* value into a column.

Queries:

Create - Table with Constraints (Primary key, not null, unique and default value):

```
SQL> create table stud
     (
      rno number primary key,
      sname varchar2(30) not null,
      dept char(5),
      sem number,
      dob date,
      email_id varchar2(20) unique,
      faculty varchar2(15) default 'Engineering'
     );
```

Table created.

SQL> desc stud;

Name	Null?	Type
RNO	NOT NULL	NUMBER
SNAME	NOT NULL	VARCHAR2(30)
DEPT		CHAR(5)
SEM		NUMBER
DOB		DATE
EMAIL_ID		VARCHAR2(20)
FACULTY		VARCHAR2(15)

Create - Table with Constraints (Primary key, foreign key, check constraint)

```
SQL>  create table exam
      (
        regno number primary key,
        rno number references stud(rno),
        dept char(5) not null,
        mark1 number check (mark1<=100 and mark1>=0),
        mark2 number check (mark2<=100 and mark2>=0),
        mark3 number check (mark3<=100 and mark3>=0),
        mark4 number check (mark4<=100 and mark4>=0),
        mark5 number check (mark5<=100 and mark5>=0),
        total number,
        average number,
        grade char(1)
      );
```

Table created.

SQL> desc exam;

Name	Null?	Type
REGNO	NOT NULL	NUMBER
RNO		NUMBER
DEPT	NOT NULL	CHAR(5)
MARK1		NUMBER
MARK2		NUMBER
MARK3		NUMBER
MARK4		NUMBER
MARK5		NUMBER
TOTAL		NUMBER
AVERAGE		NUMBER
GRADE		CHAR(1)

Result:

Thus, different DML queries with different constraints are successfully executed and verified.

2. Data Manipulation Language Statements

(insert, update, select, delete and truncate)

Aim:

To work with DML queries like insert, update select, delete and truncate statements

Concept:

DML	A **data manipulation language** (**DML**) is a family of syntax elements similar to a computer programming language used for selecting, inserting, deleting and updating data in a database.
Insert	The insert into *statement* is used to *insert* new records in a table. *Insert into table_name (col1, col2….) Values (val1, val2….);* *Insert into table_name values(val1, val2,…….);* *Insert into table_name values(&col1, &col2,…..);*
Update	The update *query* is used to modify the existing records in a table. With where clause particular row or record can alone be update *Update table table_name set column_name = new_value where column_name = value;*
Delete	The delete q*uery* is used to *delete* the existing records from a table. With where clause particular row or record can alone be deleted *Delete from table_name where column_name = value;*
Select	*Select statement* is used to fetch the data from a database table which returns data in the form of result table. *Select column_name from table_name;* *Select * from table_name;* *Select distinct column_name(s) from table_name;* *Select column_name(s) from table_name where column_name in (value1, value2,…….);* *Select column_name from table_name where column_name between value1 and value2;* *Select * from table_name where <condition> order by column_name asc/desc;*
Truncate	The truncate table command is used to delete the complete data from an existing table without affecting its structure. *Truncate table table_name;*

```
SQL>  create table std
(
sname varchar2(10),
roll number(5),
dept varchar(3),
m1 number(3),
m2 number(3),
m3  number(3)
);
```

Table created.

Insert :

Insert values into the table:

```
SQL> insert into std values ('&name','&roll','&dept','&m1','&m2','&m3');
Enter value for name: Kanthi
Enter value for roll: 5353
Enter value for dept: CSE
Enter value for m1: 75
Enter value for m2: 85
Enter value for m3: 80
old   1: insert into std values ('&name','&roll','&dept','&m1','&m2','&m3')
new   1: insert into std values ('Kanthi','5353','CSE','75','85','80')
```

1 row created.

```
SQL> insert into std values ('Mathi',6363,'CSE',50,60,70);
```

1 row created.

```
SQL> insert into std values ('Ram',111,'CSE',60,70,80);
```

1 row created.

```
SQL> insert into std values ('Rahim',222,'CSE',65,65,80);
```

1 row created.

```
SQL> insert into std values ('Josepth',333,'CSE',70,70,85);
```

1 row created.

```
SQL>insert into std (roll, dept, m1, m2, m3,sname) values (55,'ECE',50,50,65,'Banu');
```

1 row created.

Select :

Display all the records available in the table selected.

SQL> select * from std;

SNAME	ROLL	DEP	M1	M2	M3
Kanthi	5353	CSE	75	85	80
Mathi	6363	CSE	50	60	70
Ram	111	CSE	60	70	80
Rahim	222	CSE	65	65	80
Josepth	333	CSE	70	70	85
Banu	55	ECE	50	50	65

6 rows selected.

```
SQL>  alter table std add
      (
      total number(4),
      avg number(4),
      grade varchar2(1)
      );
```

Table altered.

SQL> desc std;

Name	Null?	Type
SNAME		VARCHAR2(10)
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)
TOTAL		NUMBER(4)
AVG		NUMBER(4)
GRADE		VARCHAR2(1)

Update – using addition operator +:

SQL> update std set total = (m1+m2+m3);

6 rows updated.

SQL> select * from std;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	Grade
Kanthi	5353	CSE	75	85	80	240		
Mathi	6363	CSE	50	60	70	180		
Rahim	222	CSE	65	65	80	210		
Ram	111	CSE	60	70	80	210		
Josepth	333	CSE	70	70	85	225		
Banu	55	ECE	50	50	65	165		

6 rows selected.

Update – using division operator / :

SQL> update std set avg = total/3;

6 rows updated.

SQL> select * from std;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	Grade
Kanthi	5353	CSE	75	85	80	240	80	
Mathi	6363	CSE	50	60	70	180	60	
Rahim	222	CSE	65	65	80	210	70	
Ram	111	CSE	60	70	80	210	70	
Josepth	333	CSE	70	70	85	225	75	
Banu	55	ECE	50	50	65	165	55	

6 rows selected.

Update – using where condition:

SQL> update std set grade='S' where avg >=75;

2 rows updated.

SQL> update std set grade = 'A' where avg >= 70 and avg <75;

2 rows updated.

SQL> update std set grade ='B' where avg >= 60 and avg < 70;

1 row updated.

SQL> update std set grade ='C' where avg >=50 and avg < 60;

1 row updated.

SQL> select * from std;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	Grade
Kanthi	5353	CSE	75	85	80	240	80	S
Mathi	6363	CSE	50	60	70	180	60	B
Rahim	222	CSE	65	65	80	210	70	A
Ram	111	CSE	60	70	80	210	70	A
Josepth	333	CSE	70	70	85	225	75	S
Banu	55	ECE	50	50	65	165	55	C

6 rows selected.

Delete:

Deleting the particular record using where clause:

SQL> delete from std where roll = 55;

1 row deleted.

SQL> select * from std;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	Grade
Kanthi	5353	CSE	75	85	80	240	80	S
Mathi	6363	CSE	50	60	70	180	60	B
Rahim	222	CSE	65	65	80	210	70	A
Ram	111	CSE	60	70	80	210	70	A
Josepth	333	CSE	70	70	85	225	75	S

SQL> **insert into std (roll, dept, m1, m2, m3,sname) values (55,'ECE',50,50,65,'Banu');**

1 row created.

SQL> truncate table std;

Table truncated.

SQL> select * from std;

no rows created.

Result:

Thus, different DML statements like insert, update, delete, truncate and select with different where conditions are successfully executed and verified.

3. a) Transaction Control Statements

(commit, savepoint, rollback)

Aim:

To understand the transaction control statements like commit, savepoint and rollback.

Concept:

Properties of Transactions:

Transactions have the following four standard properties, usually referred to by the acronym ACID:

- **Atomicity:** Ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure, and previous operations are rolled back to their former state.
- **Consistency:** Ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation:** Enables transactions to operate independently of and transparent to each other.
- **Durability:** Ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Control Statements:

Commit	The commit saves all transactions to the database since the last commit or rollback statement permanently.
Rollback	Used to undo transactions that have not already been saved to the database. The rollback statement can only be used to undo transactions since the last commit or rollback statement was issued.
Savepoint	Creates points within groups of transactions, used to roll the transaction back to a certain point without rolling back the entire transaction. This serves only in the creation of a savepoints among transactional statements. The rollback is used to undo a group of transactions.
Release savepoint	The release savepoint is used to remove a savepoint created.
Set transaction	The set transaction statement can be used to initiate a database transaction. This command is used to specify characteristics for the transaction to be read only, or read write.

Queries:

Case 1: [Create the table student with the following Structure]

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5

SQL> savepoint s1;

Savepoint created.

SQL> insert into student values (555,'AKN','AGRI',8);

1 row created.

SQL> savepoint s2;

Savepoint created.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5
555	AKN	AGRI	8

7 rows selected.

SQL> delete from student where roll = 101;

1 row deleted.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5
555	AKN	AGRI	8

6 rows selected.

SQL> rollback to s2;

Rollback complete.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5
555	AKN	AGRI	8

7 rows selected.

SQL> rollback to s1;

Rollback complete.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5

6 rows selected.

Case 2:

SQL> insert into student values (555,'AKN','AGRI',8);

1 row created.

SQL> delete from student where roll = 101;

1 row deleted.

ROLL	SNAME	DEPT	SEM
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5
555	AKN	AGRI	8

SQL> rollback;

Rollback complete.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5

6 rows selected

Case 3:

SQL> insert into student values (555,'AKN','AGRI',8);

1 row created.

SQL> delete from student where roll = 101;

1 row deleted.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5
555	AKN	AGRI	8

6 rows selected.

SQL> commit;

Commit complete.

SQL> rollback s1;
rollback s1
*
ERROR at line 1:
ORA-02181: invalid option to ROLLBACK WORK

SQL> rollback;

Rollback complete.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5
555	AKN	AGRI	8

6 rows selected.

Result:

Thus, the transaction control statements like commit, savepoint and rollback is studied and verified**.**

3. b) Data Control Statements

(grant and revoke)

Aim:

To understand the data control statements like grant and revoke statements.

Concept:

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are Grant and Revoke. Only Database Administrator's or owner's of the database object can provide/remove privileges on a database object.

Grant	Grant is used to provide access or privileges on the database objects to the users.
Revoke	Revoke removes user access rights or privileges to the database objects

The Syntax of GRANT staement:

```
GRANT privilege_name
ON object_name
TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];
```

privilege_name	- is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT
object_name	- is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE
user_name	- is the name of the user to whom an access right is being granted
Public	- is used to grant access rights to all users
Roles	- are a set of privileges grouped together
With grant option	- allows a user to grant access rights to other users

SQL> GRANT SELECT ON employee TO user1;

This statement grants a SELECT permission on employee table to user1.

Use the WITH GRANT option carefully because for example if the GRANT SELECT privilege on employee table is given to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if even the SELECT privilege on employee is revoked from user1, still user2 will have SELECT privilege on employee table.

REVOKE Statement:

The Syntax for the REVOKE command is:
REVOKE privilege_name
ON object_name
FROM {user_name |PUBLIC |role_name}

SQL> REVOKE SELECT ON employee FROM user1;

This statement will REVOKE a SELECT privilege on employee table from user1. When the SELECT privilege on a table from a user is revoked, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. One cannot REVOKE privileges if they were not initially granted.

Privileges and Roles:

Privileges:

A privilege defines the access rights provided to a user on a database object. There are two types of privileges.

1) **System privileges** - This allows the user to CREATE, ALTER, or DROP database objects.
2) **Object privileges** - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Few CREATE system privileges are listed below:

System Privileges	Description
CREATE object	allows users to create the specified object in their own schema.
CREATE ANY object	allows users to create the specified object in any schema.

The above rules also apply for ALTER and DROP system privileges.

Few of the object privileges are listed below:

Object Privileges	Description
INSERT	allows users to insert rows into a table.
SELECT	allows users to select data from a database object.
UPDATE	allows user to update data in a table.
EXECUTE	allows user to execute a stored procedure or a function.

Roles:

Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if user define roles, user can grant or revoke privileges to users, thereby automatically granting or revoking privileges. Users can either create Roles or use the system roles pre-defined by oracle.

Some of the privileges granted to the system roles are:

System Role	Privileges Granted to the Role
CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.
RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.
DBA	ALL SYSTEM PRIVILEGES

Creating Roles:

The Syntax to create a role:

CREATE ROLE role_name
[IDENTIFIED BY password];

To create a role called "developer" with password as "pwd", the code will be as follows

CREATE ROLE testing
[IDENTIFIED BY pwd];

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when user GRANT or REVOKE privileges to the role, user definitely have to identify it with the password.

GRANT or REVOKE privilege to a role:

e.g.: Grant CREATE TABLE privilege to a user by creating a role ‘testing’:

First: Create a testing Role

SQL> CREATE ROLE testing

Second: Grant a CREATE TABLE privilege to the ROLE testing. User can add more privileges to the ROLE.

SQL> **GRANT CREATE TABLE TO testing;**

Third: Grant the role to a user.

SQL> GRANT testing TO user1;

e.g.: Revoke a CREATE TABLE privilege from the role 'testing':

SQL> REVOKE CREATE TABLE FROM testing;

The Syntax to drop a role from the database:

DROP ROLE role_name;

Drop a role called 'testing', user can write:

SQL> DROP ROLE testing;

Result:

Thus, the data control statements like grant and revoke is studied and verified with roles.

4. Data Projection Statements

(multiple - column, alias-name, arithmetic operations, distinct record, concatenation, where clause)

Aim:

To understand the data projection statements with multiple-column, alias-name, arithmetic operations, distinct record, concatenation and where clause

Queries: [Note: Use the previously created table std]

Required / Multiple – Column:

SQL> select sname, roll from std;

SNAME	ROLL
Kanthi	5353
Mathi	6363
Ram	111
Rahim	222
Josepth	333
Banu	55

6 rows selected.

Alias – name for the column of the table:

SQL> select sname Stud_name, Roll Roll_No from std;

STUD_NAME	***ROLL_NO***
Kanthi	5353
Mathi	6363
Ram	111
Rahim	222
Josepth	333
Banu	55

6 rows selected.

Arithmetic Operation:

SQL> select sname name, total Total, Total+10 New_total from std;

NAME	TOTAL	NEW_TOTAL
Kanthi	240	250
Mathi	180	190
Ram	210	220
Rahim	210	220
Josepth	225	235
Banu	165	175

6 rows selected.

Distinct Record / Row:

SQL> select distinct dept DeptName from std;

DEPTNAME
CSE
ECE

Concatenation:

SQL> select sname||roll Login_Id from std;

LOGIN_ID
Kanthi5353
Mathi6363
Ram111
Rahim222
Josepth333
Banu55

6 rows selected.

where clause:

SQL> select * from std where dept='ECE';

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	G
Banu	55	ECE	50	50	65	165	55	C

SQL> select sname, roll from std where dept ='ECE';

SNAME	ROLL
Banu	55

SQL> select * from std where total >200;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	G
Kanthi	5353	CSE	75	85	80	240	80	S
Ram	111	CSE	60	70	80	210	70	A
Rahim	222	CSE	65	65	80	210	70	A
Josepth	333	CSE	70	70	85	225	75	S

SQL> select * from std where total <200;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	G
Mathi	6363	CSE	50	60	70	180	60	B
Banu	55	ECE	50	50	65	165	55	C

Result:

Thus, the data projection statements with multiple-column, alias-name, arithmetic operations, distinct record, concatenation and where clause was performed and verified.

5. Data Selection Statements

(between…and, in, not in, like, relational operators, logical operators)

Aim:

To understand select statements with different options like between…and, in, not in, like, relational operators, logical operators

Queries: **[Note: Use the previously created table std.]**

SQL> create table std1 as select * from std where 1=2;

Table created.

[Creating table by copying only the structure of the existing table without its record]

SQL> desc std1;

Name	Null?	Type
SNAME		VARCHAR2(10)
ROLL		NUMBER(5)
DEPT		VARCHAR2(3)
M1		NUMBER(3)
M2		NUMBER(3)
M3		NUMBER(3)
TOTAL		NUMBER(4)
AVG		NUMBER(4)
GRADE		VARCHAR2(1)

SQL> select * from std1;

no rows selected

SQL> create table std2 as select * from std;

Table created.

[**Note:** Creating table by copying the structure of the existing table along with its records]

SQL> select * from std2;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	G
Kanthi	5353	CSE	75	85	80	240	80	S
Mathi	6363	CSE	50	60	70	180	60	B
Ram	111	CSE	60	70	80	210	70	A
Rahim	222	CSE	65	65	80	210	70	A
Josepth	333	CSE	70	70	85	225	75	S
Banu	55	ECE	50	50	65	165	55	C

6 rows selected.

between … and :

SQL> select * from std2 where m2 between 65 and 75;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	G
Ram	111	CSE	60	70	80	210	70	A
Rahim	222	CSE	65	65	80	210	70	A
Josepth	333	CSE	70	70	85	225	75	S

SQL> insert into std2 values ('Madhan',4343,'MEC',50,60,70,180,60,'B');

1 row created.

SQL> insert into std2 values ('Ragu',4433,'MCA',75,75,90,240,80,'S');

1 row created.

SQL> select * from std2;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	G
Kanthi	5353	CSE	75	85	80	240	80	S
Mathi	6363	CSE	50	60	70	180	60	B
Ram	111	CSE	60	70	80	210	70	A
Rahim	222	CSE	65	65	80	210	70	A
Josepth	333	CSE	70	70	85	225	75	S
Banu	55	ECE	50	50	65	165	55	C
Madhan	4343	MEC	50	60	70	180	60	B
Ragu	4433	MCA	75	75	90	240	80	S

8 rows selected.

in statement:

SQL> select sname, roll from std2 where dept in ('MEC','MCA');

SNAME	ROLL
Madhan	4343
Ragu	4433

not in statement:
SQL> select sname, dept, roll from std2 where dept not in ('CSE');

SNAME	DEP	ROLL
Banu	ECE	55
Madhan	MEC	4343
Ragu	MCA	4433

3 rows selected.

like statement:

SQL> select sname,roll from std2 where sname like 'R%';

SNAME	ROLL
Ram	111
Rahim	222
Ragu	4433

SQL> select sname, roll, dept from std2 where sname like '%a%';

SNAME	ROLL	DEP
Kanthi	5353	CSE
Mathi	6363	CSE
Ram	111	CSE
Rahim	222	CSE
Banu	55	ECE
Madhan	4343	MEC
Ragu	4433	MCA

7 rows selected.

SQL> select sname, roll from std2 where sname like 'R_m';

SNAME	ROLL
Ram	111

SQL> select sname, roll from std2 where sname like 'R%m';

SNAME	ROLL
Ram	111
Rahim	222

SQL> select sname, roll, dept from std2 where dept like '%E';

SNAME	ROLL	DEP
Kanthi	5353	CSE
Mathi	6363	CSE
Ram	111	CSE
Rahim	222	CSE
Josepth	333	CSE
Banu	55	ECE

6 rows selected.

SQL> select * from std2;

SNAME	ROLL	DEP	M1	M2	M3	TOTAL	AVG	G
Kanthi	5353	CSE	75	85	80	240	80	S
Mathi	6363	CSE	50	60	70	180	60	B
Ram	111	CSE	60	70	80	210	70	A
Rahim	222	CSE	65	65	80	210	70	A
Josepth	333	CSE	70	70	85	225	75	S
Banu	55	ECE	50	50	65	165	55	C
Madhan	4343	MEC	50	60	70	180	60	B
Ragu	4433	MCA	75	75	90	240	80	S

8 rows selected.

SQL> insert into std2 values ('Arun',4333,'MCA',75,75,90,240,80,'S');

1 row created.

SQL> insert into std2 values ('Arunchalam',2222,'ECE',75,75,90,240,80,'S');

1 row created.

SQL> insert into std2 values ('Deva',5555,'MEC',75,75,90,240,80,'S');

1 row created.

SQL> select sname, roll, dept from std2;

SNAME	ROLL	DEP
Kanthi	5353	CSE
Mathi	6363	CSE
Ram	111	CSE
Rahim	222	CSE
Josepth	333	CSE
Banu	55	ECE
Madhan	4343	MEC
Ragu	4433	MCA
Arun	4333	MCA
Arunchalam	2222	ECE
Deva	5555	MEC

11 rows selected.

Relational Statements:

SQL> select sname, roll, dept from std2 where roll > 5000;

SNAME	ROLL	DEP
Kanthi	5353	CSE
Mathi	6363	CSE
Deva	5555	MEC

SQL> select sname, roll, dept from std2 where roll < 5000;

SNAME	ROLL	DEP
Ram	111	CSE
Rahim	222	CSE
Josepth	333	CSE
Banu	55	ECE
Madhan	4343	MEC
Ragu	4433	MCA
Arun	4333	MCA
Arunchalam	2222	ECE

Logical Statements:

SQL> select sname, roll, dept from std2 where roll > 5000 and dept='CSE';

SNAME	ROLL	DEP
Kanthi	5353	CSE
Mathi	6363	CSE
Deva	5555	MEC

3 rows selected.

SQL> select sname, roll, dept from std2 where dept = 'ECE' or dept = 'MCA';

SNAME	ROLL	DEP
Banu	55	ECE
Ragu	4433	MCA
Arun	4333	MCA
Arunachalam	2222	ECE

4 rows selected.

SQL> **select sname, roll, dept from std2 where roll < 5000 and dept in('CSE','MCA');**

SNAME	ROLL	DEP
Ram	111	CSE
Rahim	222	CSE
Josepth	333	CSE
Ragu	4433	MCA
Arun	4333	MCA

SQL> select sname, roll, dept from std2 where roll < 5000 and dept = 'CSE' or dept = 'MCA';

SNAME	ROLL	DEP
Ram	111	CSE
Rahim	222	CSE
Josepth	333	CSE
Ragu	4433	MCA
Arun	4333	MCA

Result:

Thus, the select statements with different options like between…and, in, not in, like, relational operator, logical operator has been executed and the outputs are verified.

6. Aggregate Functions

(*count, maximum, minimum, sum, average, order by, group by, having*)

Aim:

To, understand the aggregate functions like count, Minimum, maximum, sum and average along with order by and group by and having clauses.

Queries: [Note: Use the previously created table std2.]

Order by:

[To sort the records in ascending or descending order based on a particular field]

SQL> select sname, roll, dept from std2 order by dept, sname;

SNAME	ROLL	DEP
Josepth	333	CSE
Kanthi	5353	CSE
Mathi	6363	CSE
Rahim	222	CSE
Ram	111	CSE
Arunchalam	2222	ECE
Banu	55	ECE
Arun	4333	MCA
Ragu	4433	MCA
Deva	5555	MEC
Madhan	4343	MEC

11 rows selected.

[To sort it in descending order]

SQL> select sname, roll, dept from std2 order by dept desc;

SNAME	ROLL	DEP
Madhan	4343	MEC
Deva	5555	MEC
Ragu	4433	MCA
Arun	4333	MCA
Banu	55	ECE
Arunchalam	2222	ECE
Kanthi	5353	CSE
Mathi	6363	CSE
Ram	111	CSE
Rahim	222	CSE
Josepth	333	CSE

11 rows selected.

SQL> select sname, roll, dept from std2 order by dept desc, sname;

SNAME	ROLL	DEP
Deva	5555	MEC
Madhan	4343	MEC
Arun	4333	MCA
Ragu	4433	MCA
Arunchalam	2222	ECE
Banu	55	ECE
Josepth	333	CSE
Kanthi	5353	CSE
Mathi	6363	CSE
Rahim	222	CSE
Ram	111	CSE

11 rows selected.

SQL> select sname, roll, dept, total from std2 where dept like 'CSE' and total >220;

SNAME	ROLL	DEP	TOTAL
Kanthi	5353	CSE	240
Josepth	333	CSE	225

Aggregate Functions:

Count:

[To count the number of records in a particular table.]

SQL> select count(dept) as Stud_strength from std2;

STUD_STRENGTH
11

Minimum:

SQL> select min(total) as Min_marks from std2;

MIN_MARKS
165

Maximum:

SQL> select max(total) as Max_marks from std2;

MAX_MARKS
240

Group by - sum:

SQL> select dept, sum(total) as sum_total_dept from std2 group by dept;

DEP	SUM_TOTAL_DEPT
CSE	1065
ECE	405
MCA	480
MEC	420

Group by - average:

SQL> select dept, avg(total) as Total_dept_marks from std2 group by dept;

DEP	TOTAL_DEPT_MARKS
CSE	213
ECE	202.5
MCA	240
MEC	210

Having Clause:

SQL> **select * from placement;**

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	27000
2	105	CSE	Wipro	24000
3	204	MECH	Hyundai	32000
4	102	IT	infosys	29500
5	103	CSE	infosys	28000
6	555	AGRI	Annamalai	40000

6 rows selected.

SQL> **select dept, sum(salary) as dept_salary from placement group by dept having sum(salary) > 45000;**

DEPT	DEPT_SALARY
CSE	52000
IT	56500

SQL> **select dept, sum(salary) as dept_salary from placement group by dept having sum(salary) < 45000;**

DEPT	DEPT_SALARY
AGRI	40000
MECH	32000

SQL> **select dept, sum(salary) as dept_salary from placement group by dept having sum(salary)< 45000 order by dept_salary;**

DEPT	DEPT_SALARY
MECH	32000
AGRI	40000

Result:

Thus, the aggregate functions like count, minimum, maximum, sum and average along with order by and group by and having clauses have been written and the outputs were verified.

7. Join Queries

(inner join, outer join (left, right, full), equi join and non - equi join, self-join and cartesian join or cross-join.)

Aim:

To understand the concept of different join queries like inner join, outer join (left, right, full), equi join, non - equi join, self-join and cartesian join or cross-join.

Concept:

<table>
<tr><td></td><td colspan="2">Joins: Joins are used to combine rows from two or more tables</td></tr>
<tr><td>1.</td><td>Inner join</td><td>An inner join or simple join is a join that returns rows of the tables that satisfy the join condition. That is it returns all rows when there is at least one match in both tables</td></tr>
<tr><td rowspan="4">2.</td><td colspan="2">Outer join: An outer join is a join similar to the equi join, but it will also return non-matched rows from the table.</td></tr>
<tr><td>Left outer join</td><td>Return all rows from the left table, and the matched rows from the right table</td></tr>
<tr><td>Right outer join</td><td>Return all rows from the right table, and the matched rows from the left table</td></tr>
<tr><td>Full outer join</td><td>Return all rows when there is a match in one of the tables</td></tr>
<tr><td rowspan="2">3.</td><td>Equi join</td><td>An equi join is an inner join statement that uses an equivalence operation (i.e: colA = colB) to match rows from different tables. The converse of an equi join is a nonequi join operation.</td></tr>
<tr><td>Non-equi join</td><td>An non-equi (or theta) join is an inner join statement that uses an unequal operation (i.e: <>, >, <, !=, BETWEEN, etc.) to match rows from different tables. The converse of an non-equi join is a equi join operation.</td></tr>
<tr><td>4.</td><td>Self join</td><td>A self join is a join in which a table is joined with itself. For example, when user require details about an employee and his manager (also an employee).</td></tr>
<tr><td>5.</td><td>Cartesian join or cross join</td><td>A Cartesian join or Cartesian product is a join of every row of one table to every row of another table. This normally happens when no matching join columns are specified. For example, if table A with 5 rows is joined with table B with 6 rows, a Cartesian join will return 30 rows.</td></tr>
</table>

Queries:

```
SQL>  create table student
      (
      roll number primary key,
      sname varchar(30),
      dept char(5),
      sem number
      );
```

Table created.

```
SQL>  create table placement
      (
      placementID number primary key,
      roll number,
      dept char(5),
      company varchar2(30),
      salary number
      );
```

Table created.

```
SQL> insert into student(roll, sname, dept, sem) values(101,'ram','IT',5);
```

1 row created.

```
SQL> insert into student(roll, sname, dept, sem) values(102,'rahim','IT',3);
```

1 row created.

```
SQL> insert into student(roll, sname, dept, sem) values(103,'saravanan','CSE',3);
```

1 row created.

```
SQL> insert into student(roll, sname, dept, sem) values(104,'Nataraj','IT',3);
```

1 row created.

```
SQL> insert into student(roll, sname, dept, sem) values(105,'Elango','CSE',5);
```

1 row created.

```
SQL> select * from student;
```

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

Inner Join or Join:

SQL> select * from student, placement where student.roll = placement.roll;

ROLL	SNAME	DEPT	SEM	PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
104	Nataraj	IT	3	1	104	IT	infosys	25000
105	Elango	CSE	5	2	105	CSE	Wipro	22000
102	rahim	IT	3	4	102	IT	infosys	25000
103	saravanan	CSE	3	5	103	CSE	infosys	25000

SQL> select student.roll, student.sname, placement.company, placement.salary from student, placement where student.roll = placement.roll;

ROLL	SNAME	COMPANY	SALARY
104	Nataraj	infosys	25000
105	Elango	Wipro	22000
102	rahim	infosys	25000
103	saravanan	infosys	25000

Outer Joins:

Left outer Join:

SQL> select * from student, placement where student.roll = placement.roll(+);

ROLL	SNAME	DEPT	SEM	PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
101	ram	IT	5					
102	rahim	IT	3	4	102	IT	infosys	25000
103	saravanan	CSE	3	5	103	CSE	infosys	25000
104	Nataraj	IT	3	1	104	IT	infosys	25000
105	Elango	CSE	5	2	105	CSE	Wipro	22000

SQL> select student.sname, placement.placementid, placement.roll, placement.company from student, placement where student.roll = placement.roll(+);

SNAME	PLACEMENTID	ROLL	COMPANY
ram			
rahim	4	102	infosys
saravanan	5	103	infosys
Nataraj	1	104	infosys
Elango	2	105	Wipro

Right Outer Join:

SQL> select * from student, placement where student.roll(+) = placement.roll;

ROLL	SNAME	DEPT	SEM	PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
104	Nataraj	IT	3	1	104	IT	infosys	25000
105	Elango	CSE	5	2	105	CSE	Wipro	22000
				3	204	MECH	Hyundai	30000
102	rahim	IT	3	4	102	IT	infosys	25000
103	saravanan	CSE	3	5	103	CSE	infosys	25000

SQL> select student.sname, placement.placementid, placement.roll, placement.company from student, placement where student.roll(+) = placement.roll;

SNAME	PLACEMENTID	ROLL	COMPANY
Nataraj	1	104	infosys
Elango	2	105	Wipro
	3	204	Hyundai
rahim	4	102	infosys
saravanan	5	103	infosys

Full Outer Join:

SQL> select * from student, placement where student.roll(+) = placement.roll
union all
select * from student, placement where placement.roll(+) = student.roll and placement.roll is null;

ROLL	SNAME	DEPT	SEM	PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
104	Nataraj	IT	3	1	104	IT	infosys	25000
105	Elango	CSE	5	2	105	CSE	Wipro	22000
				3	204	MECH	Hyundai	30000
102	rahim	IT	3	4	102	IT	infosys	25000
103	saravanan	CSE	3	5	103	CSE	infosys	25000
101	ram	IT	5					

Equi Join:

SQL> select student.roll, student.sname, placement.company from student, placement where student.roll = placement.roll;

ROLL	SNAME	COMPANY
104	Nataraj	infosys
105	Elango	Wipro
102	rahim	Infosys
103	saravanan	Infosys

Non - Equi Join:

SQL> select student.roll, student.sname, placement.company from student, placement where student.roll > placement.roll;

ROLL	SNAME	COMPANY
105	Elango	infosys
103	saravanan	infosys
104	Nataraj	infosys
105	Elango	infosys
104	Nataraj	infosys
105	Elango	infosys

6 rows selected.

SQL> select student.roll, student.sname, placement.company from student, placement where student.roll < placement.roll;

ROLL	SNAME	COMPANY
101	ram	infosys
102	rahim	infosys
103	saravanan	infosys
101	ram	Wipro
102	rahim	Wipro
103	saravanan	Wipro
104	Nataraj	Wipro
101	ram	Hyundai
102	rahim	Hyundai
103	saravanan	Hyundai
104	Nataraj	Hyundai
105	Elango	Hyundai
101	ram	infosys
101	ram	infosys
102	rahim	infosys

15 rows selected.

Self Join:

```
SQL>  create table employee
      (
      empid number,
      empname varchar2(20),
      reportingid number
      );
```

Table created.

```
SQL> insert into employee values(1,'Principal',null);
```

1 row created.

```
SQL> insert into employee values(2,'HOD',1);
```

1 row created.

```
SQL> insert into employee values(3,'PO',1);
```

1 row created.

```
SQL> insert into employee values(4,'Staff',2);
```

1 row created.

```
SQL>  insert into employee values(5,'N T Staff',2);
```

1 row created.

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

EMPID	EMPNAME	REPORTINGID
1	Principal	
2	HOD	1
3	PO	1
4	Staff	2
5	N T Staff	2

```
SQL> select e1.empid, e1.empname, e2.empname as Head_name from employee e1,  employee e2
where e1.reportingid = e2.empid;
```

EMPID	EMPNAME	HEAD_NAME
2	HOD	Principal
3	PO	Principal
4	Staff	HOD
5	N T Staff	HOD

Cross-Join or Cartesian Join:

SQL> select * from student, placement;

ROLL	SNAME	DEPT	SEM	PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
101	ram	IT	5	1	104	IT	infosys	25000
102	rahim	IT	3	1	104	IT	infosys	25000
103	saravanan	CSE	3	1	104	IT	infosys	25000
104	Nataraj	IT	3	1	104	IT	infosys	25000
105	Elango	CSE	5	1	104	IT	infosys	25000
101	ram	IT	5	2	105	CSE	Wipro	22000
102	rahim	IT	3	2	105	CSE	Wipro	22000
103	saravanan	CSE	3	2	105	CSE	Wipro	22000
104	Nataraj	IT	3	2	105	CSE	Wipro	22000
105	Elango	CSE	5	2	105	CSE	Wipro	22000
101	ram	IT	5	3	204	MECH	Hyundai	30000
102	rahim	IT	3	3	204	MECH	Hyundai	30000
103	saravanan	CSE	3	3	204	MECH	Hyundai	30000
104	Nataraj	IT	3	3	204	MECH	Hyundai	30000
105	Elango	CSE	5	3	204	MECH	Hyundai	30000
101	ram	IT	5	4	102	IT	infosys	25000
102	rahim	IT	3	4	102	IT	infosys	25000
103	saravanan	CSE	3	4	102	IT	infosys	25000
104	Nataraj	IT	3	4	102	IT	infosys	25000
105	Elango	CSE	5	4	102	IT	infosys	25000
101	ram	IT	5	5	103	CSE	infosys	25000
102	rahim	IT	3	5	103	CSE	infosys	25000
103	saravanan	CSE	3	5	103	CSE	infosys	25000
104	Nataraj	IT	3	5	103	CSE	infosys	25000
105	Elango	CSE	5	5	103	CSE	infosys	25000

25 rows selected.

Result:

Thus, the concept of different join queries like inner join, outer join (left, right, full), equi join, non - equi join, cross-joins and self-join are executed and verified successfully.

8. Sub - queries

(in, not in, some, any, all, exits, not exits)

Aim:

To understand the concept of sub-queries using the clauses like in, not in, some, any, all, exist and not exits.

Concept:

A Sub query or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.

A sub query is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

A sub queries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

There are a few rules that sub queries must follow:

1. Sub queries must be enclosed within parentheses.
2. 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.
3. An ORDER BY cannot be used in a sub query, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a sub query.
4. Sub queries that return more than one row can only be used with multiple value operators, such as the IN operator.
5. A sub query cannot be immediately enclosed in a set function.
6. The BETWEEN operator cannot be used with a sub query; however, the BETWEEN operator can be used within the sub query.

SOME	Compares a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row. SOME must match at least one row in the sub query and must be preceded by comparison operators. Suppose using greater than (>) with SOME means greater than at least one value.
ANY	Compares a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row. ANY must be preceded by comparison operators.
ALL	ALL is used to select all records of a SELECT STATEMENT. It compares a value to every value in a list or results from a query. The ALL must be preceded by the comparison operators and evaluates to TRUE if the query returns no rows. For example, ALL means greater than every value, means greater than the maximum value. Suppose ALL (1, 2, 3) means greater than 3.

IN	The IN operator checks a value within a set of values separated by commas and retrieve the rows from the table which are matching.
NOT IN	The IN operator checks a value within a set of values separated by commas and retrieve the rows from the table which are NOT matching.
EXISTS	Checks the existence of a result of a sub query. The EXISTS sub query tests whether a sub query fetches at least one row. When no data is returned then this operator returns 'FALSE' A valid EXISTS sub query must contain an outer reference and it must be a correlated sub query. The select list in the EXISTS sub query is not actually used in evaluating the EXISTS so it can contain any valid select list
NOT EXISTS	It works like EXISTS, except the WHERE clause in which it is used is satisfied if no rows are returned by the sub query.

Queries:

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

Simple Sub-query: *using = and > :*

SQL> select * from student where dept = (select dept from student where sname = 'ram');

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
104	Nataraj	IT	3

SQL> select * from student where sem > (select sem from student where sname = 'rahim');

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
105	Elango	CSE	5

Sub-query: *using ' in'*

SQL> select * from student where dept in (select dept from student where sname = 'rahim');

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
104	Nataraj	IT	3
102	rahim	IT	3

SQL> select sname, dept from student where roll in (select roll from placement where company = 'infosys');

SNAME	DEPT
rahim	IT
saravanan	CSE
Nataraj	IT

Sub-query: *using 'not in'*

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

SQL> select * from student where dept not in (select dept from student where sname = 'rahim');

ROLL	SNAME	DEPT	SEM
103	saravanan	CSE	3
105	Elango	CSE	5

SQL> select roll, sname from student where roll not in (select roll from placement);

ROLL	SNAME
101	ram

SQL> select sname, dept from student where roll not in (select roll from placement where company = 'infosys');

SNAME	DEPT
ram	IT
Elango	CSE

Sub-query: *using ' some'*

SQL> select * from placement where salary < some (23000,18000);

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
2	105	CSE	Wipro	22000

SQL> select * from placement where salary > some (23000,24000);

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

Note: in place 'some', 'any' can also be used to get the similar records.

Sub-query: *using ' all'*

SQL> select * from placement where salary < all (23000,28000);

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
2	105	CSE	Wipro	22000

SQL> select * from placement where salary > all (23000,28000);

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
3	204	MECH	Hyundai	30000

Sub-query: *using ' exists '*

SQL> select * from placement where exists (select * from placement where salary < 25000);

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

Sub-query: *using ' not exists '*

SQL> select * from placement where not exists (select * from placement where salary <25000);

no rows selected

Result:

Thus, the concept of sub-queries using the clauses like in, not in, some, any, all, exist and not exits written, executed and successfully verified.

9. Set Operations

(union, union all, intersect, minus)

Aim:

To perform the Set operations like union, union all, intersect and minus on tables

Concept:

Union	The UNION operator is used to combine the result-set of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns and must be in same order. The columns must also have similar data types. The UNION operator selects only distinct values by default Select column1 from table1 union select column1 from table2;
Union all	The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate selected rows. Select column1 from table1 union all select column1 from table2;
Intersect	The intersect operator returns only those rows returned by both queries. Select column1 from table1 intersect select column1 from table2;
Minus	The minus operator returns only rows returned by the first query but not by the second. Select column1 from table1 minus select column1 from table2;

Note: *Use the previously created tables placement and student*.

Union :

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5

SQL> select roll from student union select roll from placement;

ROLL
101
102
103
104
105
204

6 rows selected.

Union ALL:

SQL> select roll from student union all select roll from placement;

ROLL
101
102
103
104
105
104
105
204
102
103

10 rows selected.

Intersect:

SQL> select roll from student intersect select roll from placement;

ROLL
102
103
104
105

Minus:

SQL> select roll from student minus select roll from placement;

ROLL
101

SQL> select roll from placement minus select roll from student;

ROLL
204

Result:

Thus, the SQL statements for set operations like union, union all, intersect and minus is written and verified successfully.

10. Database Objects

(*Synonyms, Sequences, Views, Index*)

Aim:

To understand the concept of database objects like of synonym, sequence, view and index in oracle.

a) Synonym:

Concept:

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

Syntax

The syntax to create a synonym in Oracle is:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema .] synonym_name

 FOR [schema .] object_name [@ dblink];
```

OR REPLACE allows user to recreate the synonym (if it already exists) without having to issue a DROP synonym command.

PUBLIC means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.

schema is the appropriate schema. If this phrase is omitted, Oracle assumes that users are referring to their own schema. [Here the user name is the schema]

object_name is the name of the object for which user is creating the synonym. It can be one of the following:

- table
- view
- sequence
- stored procedure
- function
- package
- materialized view
- java class schema object
- user-defined object
- synonym

Queries:

Creating synonym for Student table.

SQL> **create synonym ss for student;**

Synonym created.

SQL> **select * from student;**

and

SQL> **select * from ss;** [Both give the same resultant of rows.]

ROLL	SNAME	DEPT	SEM
1	Kanthi	CSE	8
2	Mathi	IT	8
102	Nathan	CSE	8
202	Ragupathy	MCA	10

SQL> **drop synonym ss;**

Synonym dropped.

SQL> **select * from ss;**
select * from ss
*
ERROR at line 1:
ORA-00942: table or view does not exist

b) Sequences:

Concept:

In Oracle, user can create an auto number field by using sequences. A sequence is an object in Oracle that is used to generate a number sequence. This can be useful when the user need to create a unique number to act as a primary key.

Syntax:

Create Sequence:

```
CREATE SEQUENCE sequence_name
 MINVALUE value
 MAXVALUE value
 START WITH value
 INCREMENT BY value
  CACHE value;
```

To retrieve the next value in the sequence order, ***sequence_name*.NEXTVAL** is used.

In sequence, the *cache* option specifies how many sequence values will be stored in memory for faster access. Also, Creating the same sequence with the NOCACHE is also allowed.

Drop Sequence

Sequence can dropped using

```
DROP SEQUENCE sequence_name;
```

Queries:

Note: Use the previously created "**student**" table structure. If there is any record, delete all the records from the table.

Sequence Creation:

SQL> **create sequence roll_seq minvalue 1 start with 1 increment by 1 cache 20;**

Sequence created.

If there is any record, truncate the rows.

SQL> **truncate table student;**

Table truncated.

SQL> **select * from student;**

no rows selected

SQL> **insert into student values (roll_seq.nextval,'Kanthi','CSE',8);**

1 row created.

SQL> **insert into student values (roll_seq.Nextval,'Mathi','IT',8);**

1 row created.

SQL> **alter sequence roll_seq increment by 100;**

Sequence altered.

SQL> **insert into student values(roll_seq.nextval,'Nathan','CSE',8);**

1 row created.

SQL> **insert into student values(roll_seq.nextval,'Ragupathy','MCA',10);**

1 row created.

SQL> **select * from student;**

ROLL	SNAME	DEPT	SEM
1	Kanthi	CSE	8
2	Mathi	IT	8
102	Nathan	CSE	8
202	Ragupathy	MCA	10

SQL> **drop sequence roll_seq;**

Sequence dropped.

SQL> **insert into student values(roll_seq.nextval,'Raman','MECH',5);**
insert into student values(roll_seq.nextval,'Raman','MECH',5)
*
ERROR at line 1:
ORA-02289: sequence does not exist

c) Views:

Concept:

A VIEW is a virtual table, through which a selective portion of the data from one or more tables can be seen just like a real table. Views do not contain data of their own. SQL functions, WHERE, and JOIN statements can be added to a view and present the data as if the data were coming from one single table. They are used to restrict access to the database or to hide data complexity. A view is stored as a SELECT statement in the database. DML operations on a view like INSERT, UPDATE, DELETE affects the data in the original table upon which the view is based. A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

Note: Use the tables' placement and student for creating views.

Queries:

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5

SQL> create view placeview as
(select roll, dept, company from placement where salary > 25000);

View created.

SQL> select * from placeview;

ROLL	DEPT	COMPANY
204	MECH	Hyundai

SQL> create view studview as
(select student.roll, student.sname, student.dept, student.sem, placement.company, placement.salary from student, placement where student.roll = placement.roll);

SQL> select * from studview;

ROLL	SNAME	DEPT	SEM	Company	Salary
104	Nataraj	IT	3	infosys	25000
105	Elango	CSE	5	Wipro	22000
102	rahim	IT	3	infosys	25000
103	saravanan	CSE	3	infosys	25000

SQL> insert into placement (placementid, roll, dept, company, salary) values (6, 106, 'CSE', 'infosys',25000);

1 row created.

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000
6	106	CSE	infosys	25000

SQL> insert into student (roll, sname, dept, sem) values (106, 'kannan', 'CSE', 5);

1 row created.

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5

SQL> select * from studview;

ROLL	SNAME	DEPT	SEM	Company	Salary
104	Nataraj	IT	3	infosys	25000
105	Elango	CSE	5	Wipro	22000
102	rahim	IT	3	infosys	25000
103	saravanan	CSE	3	infosys	25000
106	kannan	CSE	5	infosys	25000

SQL> delete from studview where roll = 106;

1 row deleted.

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

SQL> update studview set salary = 27500 where roll = 102;

1 row updated.

SQL> select * from studview;

ROLL	SNAME	DEPT	SEM	Company	Salary
104	Nataraj	IT	3	infosys	25000
105	Elango	CSE	5	Wipro	22000
102	rahim	IT	3	infosys	27500
103	saravanan	CSE	3	infosys	25000

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	27500
5	103	CSE	infosys	25000

SQL> drop view studview;

View dropped.

SQL> select * from studview;
select * from studview
*
ERROR at line 1:
ORA-00942: table or view does not exist

d) Index:

Concept:

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Creating an index involves the CREATE INDEX statement, which allows user to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order.

Indexes can also be unique, similar to the UNIQUE constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index.

```
CREATE INDEX index_name ON table_name;
```

Single-Column Indexes:

A single-column index is one that is created based on only one table column. The basic syntax is as follows:

```
CREATE INDEX index_name
ON table_name (column_name);
```

Unique Indexes:

Unique indexes are used not only for performance, but also for data integrity. A unique index does not allow any duplicate values to be inserted into the table. The basic syntax is as follows:

```
CREATE UNIQUE INDEX index_name
on table_name (column_name);
```

Composite Indexes:

A composite index is an index on two or more columns of a table. The basic syntax is as follows:

```
CREATE INDEX index_name
on table_name (column1, column2);
```

Whether to create a single-column index or a composite index, take into consideration the column(s) that user may use very frequently in a query's WHERE clause as filter conditions.

Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.

Implicit Indexes:

Implicit indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

The DROP INDEX Command:

An index can be dropped using SQL **DROP** command. Care should be taken when dropping an index because performance may be slowed or improved.

The basic syntax is as follows:

```
DROP INDEX index_name;
```

Situations in which Indexes to be avoided:

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

- Indexes should not be used on small tables.
- Tables that have frequent, large batch update or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

Query:

SQL> **create index rroll on placement(roll);**

Index created.

SQL> create index rroll2 on placement(roll) global partition by range(roll)
(partition a values less than (200),
partition b values less than (500),
partition c values less than (maxvalue));

Index created.

SQL> drop index rroll;

Index dropped.

Result:

Thus, the concept of database objects like synonym, sequence, views and index are understood and its usages are verified.

11. Cursors

Aim:

To understand the use of cursor in oracle databases.

Concept:

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

User can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, user can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement, which returns more than one row.

The syntax for creating an explicit cursor is:

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor involves four steps:

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_student IS SELECT roll, sname, dept FROM student;
```

Opening the Cursor

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

```
OPEN c_student;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

```
FETCH c_student into c_roll, c_sname, c_dept;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close above-opened cursor as follows:

```
CLOSE c_student;
```

Queries:

Implicit Cursor:

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	27500
5	103	CSE	infosys	25000

Type the following statements in the notepad and save it as cursor1.lst

```
DECLARE
   total_rows number(2);
   BEGIN
      UPDATE placement SET salary = salary + 500;
      IF sql%notfound THEN
         dbms_output.put_line('no customers selected');
      ELSIF sql%found THEN
         total_rows := sql%rowcount;
         dbms_output.put_line( total_rows || ' students salary updated  ');
      END IF;
   END;
/
```

Note: Type ‘ / ‘ at the end of the last executable statement in the PL/SQL block and leave empty line after it to avoid the error “Input truncated to 1 characters”

Set Serveroutput [On | Off] is used to enable or disable the output made by the DBMS_OUTPUT package to display it in the user screen.

SQL> set serveroutput on;

SQL> @g:\oracle\cursor1.lst;

5 students salary updated

PL/SQL procedure successfully completed.

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	25500
2	105	CSE	Wipro	22500
3	204	MECH	Hyundai	30500
4	102	IT	infosys	28000
5	103	CSE	infosys	25500

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	ram	IT	5
102	rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
106	kannan	CSE	5

Explicit Cursor:

Type the following statements in the notepad and save it as cursor2.lst

```
DECLARE
  c_roll student.roll%type;
  c_sname student.sname%type;
  c_dept student.dept%type;
  CURSOR c_student is SELECT roll, sname, dept FROM student;
BEGIN
  OPEN c_student;
  LOOP
    FETCH c_student into c_roll, c_sname, c_dept;
    EXIT WHEN c_student%notfound;
    dbms_output.put_line(c_roll || '    ' || c_sname || '       ' || c_dept);
  END LOOP;
  CLOSE c_student;
END;
/
```

SQL> @g:\oracle\cursor2.lst;

ROLL	SNAME	DEPT
101	ram	IT
102	rahim	IT
103	saravanan	CSE
104	Nataraj	IT
105	Elango	CSE
106	kannan	CSE

PL/SQL procedure successfully completed.

Result:

Thus, the PL/SQL for cursor in oracle is written and executed successfully.

12. a) Procedures

Aim:

To understand the concept of procedures in oracle

Concept:

A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program, which is called the calling program.

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

- **Functions**: these subprograms return a single value, mainly used to compute and return a value.
- **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may have a parameter list. Like anonymous PL/SQL blocks and, the named blocks a subprograms will also have following three parts:

Declarative Part: It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
Executable Part: This is a mandatory part and contains statements that perform the designated action.
Exception-handling: This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Queries:

The following statement creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

Execute the above code using SQL prompt to produce the following result.

Procedure created.

Executing a Standalone Procedure

A standalone procedure can be called in two ways:

- Using the EXECUTE keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named 'greetings' can be called with the EXECUTE keyword as:

SQL> EXECUTE greetings;

The above call would display:

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block:

```
BEGIN
   greetings;
END;
/
```

The above call would display:

Hello World

PL/SQL procedure successfully completed.

Deleting a Standalone Procedure

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is:

```
SQL> DROP PROCEDURE procedure-name;
```

The procedure can also be deleted from another PL/SQL block

```
BEGIN
   DROP PROCEDURE greetings;
END;
/
```

Parameter Modes in PL/SQL Subprograms

IN: An IN parameter allow to pass a value to the subprogram. It is a read-only parameter. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. It can only pass a constant, literal, initialized variable, or expression as an IN parameter. It can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. It is the default mode of parameter passing. Parameters are passed by reference.
OUT: An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. It can change its value and reference the value after assigning it. The actual parameter must be variable and it is passed by value.
IN OUT: An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. Actual parameter is passed by value.

Procedure 1: IN & OUT Mode

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

```
DECLARE
   a number;
   b number;
   c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;

BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

Procedure 2: IN & OUT Mode

This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

Above code is executed at SQL prompt, to produces the following result:

Square of (23): 529

PL/SQL procedure successfully completed.

Result:

Thus, the SQL function is written and executed successfully.

12. b) Functions

Aim:

To understand the concept of functions in oracle

Concept:

A PL/SQL function is same as a procedure except that it returns a value.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- *RETURN* clause specifies that data type users are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Procedure 1:

```
CREATE OR REPLACE FUNCTION totalstudent
RETURN number IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total FROM student;
  RETURN total;
END;
/
```

Type the above statements in the notepad and save it as procedure1.lst

SQL> @g:\oracle\procedure1.lst;

Function created.

Calling a Function

To create a function, give a definition of what the function has to do. To use a function, call that function to perform the defined task from the main function or from outside. When a program calls a function, program control is transferred to the called function or main function.

A called function performs defined task and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function, simply pass the required parameters along with function name and function returned value could be stored.

Procedure 2:

```
DECLARE
  c number(2);
BEGIN
  c := totalstudent();
  dbms_output.put_line('Total no. of students :  ' || c);
END;
/
```

Type the above statements in the notepad and save it as procedure2.lst

SQL> @g:\oracle\procedure2.lst;

Total no. of Customers: 6

PL/SQL procedure successfully completed.

Result:

Thus, the SQL function is written and executed successfully.

13. Triggers

Aim:

To understand the concept of a trigger in oracle.

Concept:

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are written to be executed in response to any of the following events:

A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).

- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically.
- Enforcing referential integrity.
- Event logging and storing information on table access.
- Auditing.
- Synchronous replication of tables.
- Imposing security authorizations.
- Preventing invalid transactions.

Creating Triggers

The syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

where,

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the *trigger_name*.

- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows user to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

Queries:

The following program creates a **row level** trigger for the student table that would fire for INSERT or UPDATE or DELETE operations performed on the student table. The trigger will display the salary difference between the old values and new values.

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON placement
FOR EACH ROW
WHEN (NEW.roll > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

Type the above statements in the notepad and save it as Trigger1.lst

SQL> @g:\oracle\trigger1.lst;

Trigger created.

Note:

- OLD and NEW references are not available for table level triggers, rather use them for record level triggers.
- To query the table in the same trigger, then it should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but trigger can be written on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

Triggering a Trigger:

Note: Use the previously created placement table or create with the following data.

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	27000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	27500
5	103	CSE	infosys	25000

SQL> insert into placement values (6,555,'AGRI','Annamalai',40000);

Old salary:
New salary: 40000
Salary difference:

1 row created.

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	27000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	27500
5	103	CSE	infosys	25000
6	555	AGRI	Annamalai	40000

6 rows selected.

SQL> update placement set salary = salary + 1000 where roll = 103;

Old salary: 27000
New salary: 28000
Salary difference: 1000

1 row updated.

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	27000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	27500
5	103	CSE	infosys	26000
6	555	AGRI	Annamalai	40000

6 rows selected.

Result:

Thus, the trigger is created and its functions are studied and verified successfully.

14. Exceptions

Aim:

To understand the concept of exception handling mechanism in PL/SQL.

Concept:

An error condition during a program execution is called an exception in PL/SQL. 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:

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling:

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling goes here >
   WHEN exception1 THEN
       exception1-handling-statements
   WHEN exception2  THEN
      exception2-handling-statements
   WHEN exception3 THEN
      exception3-handling-statements
   ........
   WHEN others THEN
      exception3-handling-statements
END;
```

The default exception will be handled using *WHEN others THEN*.

Note: Use the previously created placement table or create with the following data.

SQL> select * from placement;

PLACEMENTID	ROLL	DEPT	COMPANY	SALARY
1	104	IT	infosys	27000
2	105	CSE	Wipro	22000
3	204	MECH	Hyundai	30000
4	102	IT	infosys	27500
5	103	CSE	infosys	25000

Exception 1:

Type the below statements in the notepad and save it as Exception1.lst

```
DECLARE
  c_placementid placement.placementid%type := 8;
  c_roll placement.roll%type;
  c_dept placement.dept%type;
BEGIN
  SELECT  roll, dept INTO  c_roll, c_dept FROM placement WHERE placementid =
c_placementid;
  DBMS_OUTPUT.PUT_LINE ('PlacementID: '||  c_placementid);
  DBMS_OUTPUT.PUT_LINE ('Dept: ' || c_dept);
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such Student Record Present!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

SQL> @g:\oracle\Exception1.lst;

No such Student Record Present!

PL/SQL procedure successfully completed.

Change the 2nd line in the above as
"c_placementid placement.placementid%type := **5**; "

SQL> @g:\oracle\Exception1.lst;

PlacementID: 5
Dept: CSE

PL/SQL procedure successfully completed.

Raising Exceptions

The database server raises exceptions automatically whenever there is any internal database error, but the programmer can raise exceptions explicitly by using the command **RAISE**.

Syntax of raising an exception:

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

User can use above syntax in raising Oracle standard exception or any user-defined exception.

User-defined Exceptions

PL/SQL allows users to define their own exceptions according to the need of the program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

The syntax for declaring an exception:

```
DECLARE my-exception EXCEPTION;
```

Note: Use the previously created student table or create with the following data

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	Ram	IT	5
102	Rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5

6 rows selected

Exception 2:

Type the below statements in the notepad and save it as Exception2.lst

```
DECLARE
   c_roll student.roll%type := &cc_roll;
   c_sname  student.sname%type;
   c_dept student.dept%type;

   -- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_roll <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  sname, dept INTO  c_sname, c_dept FROM student WHERE roll
= c_roll;

      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_sname);
      DBMS_OUTPUT.PUT_LINE ('Dept: ' || c_dept);
   END IF;
EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('Roll number must be greater than zero!');
   WHEN no_data_found THEN
      dbms_output.put_line('No such Student record Found!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

SQL> @g:\oracle\Exception2.lst;

Enter value for cc_roll: 102

old 2: c_roll student.roll%type := &cc_roll;
new 2: c_roll student.roll%type := 102;

Name: rahim
Dept: IT

PL/SQL procedure successfully completed.

SQL> @g:\oracle\Exception2.lst;

Enter value for cc_roll: 1

old 2: c_roll student.roll%type := &cc_roll;
new 2: c_roll student.roll%type := 1;

No such Student record Found!

PL/SQL procedure successfully completed.

SQL> @g:\oracle\Exception2.lst;

Enter value for cc_roll: 0

old 2: c_roll student.roll%type := &cc_roll;
new 2: c_roll student.roll%type := 0;

Roll number must be greater than zero!

PL/SQL procedure successfully completed.

Result:

Thus, the concept of exception handling is studied, written and executed successfully.

15. Packages

Aim:

To understand the concept of packages in oracle.

Concept:

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

User can have many global variables defined and multiple procedures or functions inside a package.

Queries:

Note: Use the previously created student table or create with the following data

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
101	Ram	IT	5
102	Rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5

6 rows selected

Step 1: THE PACKAGE SPECIFICATION

Type the below code in the notepad and save it as “Package1.lst”

```
CREATE OR REPLACE PACKAGE stud_package AS
  -- Adds a student record
  PROCEDURE addstudent(p_roll   student.roll%type,
  p_sname  student.sname%type,
  p_dept  student.dept%type,
  p_sem  student.sem%type);

  -- Removes a student record
  PROCEDURE delstudent(p_roll  student.roll%TYPE);

  --Lists all student record
  PROCEDURE liststudent;

END stud_package;
/
```

SQL> @g:\oracle\package1.lst;

Package created.

Step 2: CREATING THE PACKAGE BODY

Type the below code in the notepad and save it as "Package2.lst"

```
CREATE OR REPLACE PACKAGE BODY stud_package AS

  PROCEDURE addstudent(p_roll  student.roll%type,
    p_sname  student.sname%type,
    p_dept  student.dept%type,
    p_sem  student.sem%type)
  IS
  BEGIN
    INSERT INTO student (roll,sname,dept,sem) VALUES (p_roll, p_sname, p_dept, p_sem);
  END addstudent;

  PROCEDURE delstudent(p_roll   student.roll%type) IS
  BEGIN
     DELETE FROM student WHERE roll = p_roll;
  END delstudent;

  PROCEDURE liststudent IS

  CURSOR p_student is SELECT  sname FROM student;

  TYPE s_list is TABLE OF student.sname%type;
  name_list s_list := s_list();
  counter integer :=0;
  BEGIN
    FOR n IN p_student LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter)  := n.sname;
    dbms_output.put_line('Student(' ||counter|| ') : '||name_list(counter));
    END LOOP;
  END liststudent;
END stud_package;
/
```

SQL> @g:\oracle\package2.lst;

Package body created.

Step 3 : USING THE PACKAGE

Type the below code in the notepad and save it as “Package3.lst”

```
DECLARE
  code student.roll%type:= &ccode;
BEGIN
     stud_package.addstudent(222, 'Rajni', 'MECH', 3);
     stud_package.addstudent(333, 'Suban', 'TECH', 7);
     stud_package.liststudent;
     dbms_output.put_line('-----------------------------');
     stud_package.delstudent(code);
     stud_package.liststudent;
END;
/
```

SQL> @g:\oracle\pack3.lst

Enter value for ccode: 222

```
  old   2:    code student.roll%type:= &ccode;
new   2:    code student.roll%type:= 222;

Student(1) : rahim
Student(2) : saravanan
Student(3) : Nataraj
Student(4) : Elango
Student(5) : Ram
Student(6) : Rajni
Student(7) : Suban
-----------------------------
Student(1) : rahim
Student(2) : saravanan
Student(3) : Nataraj
Student(4) : Elango
Student(5) : Ram
Student(6) : Suban

PL/SQL procedure successfully completed.
```

SQL> select * from student;

ROLL	SNAME	DEPT	SEM
102	Rahim	IT	3
103	saravanan	CSE	3
104	Nataraj	IT	3
105	Elango	CSE	5
101	Ram	IT	5
333	Suban	TECH	7

6 rows selected.

Result:

Thus the concept of package specification and definition is done and successfully verified.