

Title: Create a New Database and Perform Following operations on that Database.

- a) Create table
- b) Alter the table
- c) Rename Table
- d) Drop the table. (Assume your own data).

Theory:

1. CREATE DATABASE query syntax and example.

It is a DDL command used to create new database. Later we have to 'USE' the database

- **Syntax:** CREATE DATABASE database_name;
 USE DATABASE database_name;
- **Example:** CREATE DATABASE students;
 USE students;

2. CREATE TABLE query syntax and example.

CREATE TABLE is a DDL command used to create a new table.

- **Syntax:** CREATE TABLE table_name (
 column1 datatype,
 column2 datatype,
 column3 datatype,

);
- **Example:** CREATE TABLE Persons (
 rollNo INT,
 lastName VARCHAR(25),
 firstName VARCHAR(25),
 address VARCHAR(255),
 birthDate DATE
);

3. ALTER TABLE query syntax and example.

ALTER TABLE is a DDL Command use to add, delete, or modify columns in an existing table. The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

- **Syntax:** ALTER TABLE *table_name*
 ADD *column_name datatype*;
- **Example:** ALTER TABLE Persons
 ADD email VARCHAR(255);

4. RENAME TABLE query syntax and example.

It is a DDL Command use to rename the table.

- **Syntax:** RENAME TABLE *table_name* TO *new_name*;
- **Example:** RENAME TABLE Persons TO firstYearStudents;

5. DROP TABLE query syntax and example.

The DROP TABLE query is used to drop an existing table in a database along with it's structure.

- **Syntax:** DROP TABLE *table_name*;
- **Example:** DROP TABLE firstYearStudents;

Queries and output:

1. CREATE DATABASE query syntax and example.

The screenshot shows the MySQL Workbench interface. On the left, the Navigator pane displays the schema structure. Under the 'Schemas' section, the 'students' schema is selected, which contains tables like 'branch', 'client', 'employee', etc. In the center, the Query Editor window titled 'Query 1' shows the following SQL code:

```
1 • CREATE DATABASE students;
2 • USE students;
3 • CREATE TABLE Persons (
    rollNo INT,
    lastName VARCHAR(25),
    firstName VARCHAR(25),
    address VARCHAR(255),
    birthDate DATE
);
10 • ALTER TABLE Persons ADD email VARCHAR(255);
12 • RENAME TABLE Persons TO firstYearStudents;
14 • DROP TABLE firstYearStudents;
```

2. CREATE TABLE query syntax and example.

The screenshot shows the MySQL Workbench interface. On the left, the Navigator pane displays the schema structure. Under the 'Schemas' section, the 'students' schema is selected, which contains a table named 'persons'. This table has columns: rollNo, lastName, firstName, address, birthDate, and email. In the center, the Query Editor window titled 'Query 1' shows the following SQL code:

```
1 • CREATE DATABASE students;
2 • USE students;
3
4 • CREATE TABLE Persons (
    rollNo INT,
    lastName VARCHAR(25),
    firstName VARCHAR(25),
    address VARCHAR(255),
    birthDate DATE
);
11 • ALTER TABLE Persons ADD email VARCHAR(255);
13 • RENAME TABLE Persons TO firstYearStudents;
15 • DROP TABLE firstYearStudents;
```

Navigator

SCHEMAS

- friends
- sakila
- students**
 - Tables
 - persons**
 - Columns
 - rollNo
 - lastName
 - firstName
 - address
 - birthDate
 - Indexes
 - Foreign Keys
 - Triggers
 - Views
 - Stored Procedures
 - Functions
- sys
- world

Administration Schemas

Information

Connection Details

- Local instance
- MySQL80
- Hocalhost
- B306
- root

Query 1

```

6      lastName VARCHAR(25),
7      firstName VARCHAR(25),
8      address VARCHAR(255),
9      birthDate DATE
10     );
11 • DESC Persons;
12
13 • ALTER TABLE Persons ADD email VARCHAR(255);
14
15 • RENAME TABLE Persons TO firstYearStudents;
16
17 • DROP TABLE firstYearStudents;
18

```

Result Grid | Filter Rows: Export: Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
rollNo	int	YES		NULL	
lastName	varchar(25)	YES		NULL	
firstName	varchar(25)	YES		NULL	
address	varchar(255)	YES		NULL	
birthDate	date	YES		NULL	

Result Grid | Form Editor | Field

3. ALTER TABLE query syntax and example.

Navigator

SCHEMAS

- friends
- sakila
- students**
 - Tables
 - persons**
 - Columns
 - rollNo
 - lastName
 - firstName
 - address
 - birthDate
 - email
 - Indexes
 - Foreign Keys
 - Triggers
 - Views
 - Stored Procedures
 - Functions
- sys
- world

Administration Schemas

Information

Connection Details

- Local instance
- MySQL80
- Hocalhost
- B306
- root
- Use
- root@localhost
- Use

Query 1

```

1 • CREATE DATABASE students;
2 • USE students;
3
4 • CREATE TABLE Persons (
5     rollNo INT,
6     lastName VARCHAR(25),
7     firstName VARCHAR(25),
8     address VARCHAR(255),
9     birthDate DATE
10    );
11 • DESC Persons;
12
13 • ALTER TABLE Persons ADD email VARCHAR(255);
14
15 • RENAME TABLE Persons TO firstYearStudents;
16
17 • DROP TABLE firstYearStudents;
18

```

Output

Action Output

#	Time	Action	Message
27	11:24:44	CREATE DATABASE students	Error Code: 1007. Can't create database 'students'; database with the same name exists
28	11:24:58	DESC Persons	Error Code: 1146. Table 'students.persons' doesn't exist
29	11:25:08	CREATE TABLE Persons (rollNo INT, lastName VARCHAR(25), firstName VARCHAR(25)...) ENGINE=InnoDB DEFAULT CHARSET=utf8	0 row(s) affected
30	11:25:18	DESC Persons	5 row(s) returned
31	11:26:35	ALTER TABLE Persons ADD email VARCHAR(255)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

CREATE TABLE

CREATE TABLE creates a new table in the current database. By default, tables are created using the InnoDB storage engine. Individual storage engines like MyISAM or MEMORY permit different options.

See also: : [create-table-files](#)

4. RENAMETABLE query syntax and example.

The screenshot shows the MySQL Workbench interface with the following details:

- Navigator:** Shows the schema structure. Under the 'students' schema, there is a table named 'firstYearStudents'. This table has columns: rollNo, lastName, firstName, address, birthDate, and email.
- Query Editor (Query 1):** Contains the following SQL code:

```

6     lastName VARCHAR(25),
7     firstName VARCHAR(25),
8     address VARCHAR(255),
9     birthDate DATE
10    );
11 • DESC Persons;
12
13 • ALTER TABLE Persons ADD email VARCHAR(255);
14
15 • RENAME TABLE Persons TO firstYearStudents;
16
17 • DROP TABLE firstYearStudents;
18
19
20
21
22

```
- Information Panel:** Shows connection details for 'Local instance MySQL80 Host localhost Port 3306 User root' and the current session.
- Output Panel:** Displays the execution log with the following entries:

#	Time	Action	Message
28	11:24:58	DESC Persons	Error Code: 1146. Table 'students.persons' doesn't exist
29	11:25:08	CREATE TABLE Persons (rollNo INT, lastName VARCHAR(25), firstName VARCHAR(25))	0 row(s) affected
30	11:25:18	DESC Persons	5 row(s) returned
31	11:26:35	ALTER TABLE Persons ADD email VARCHAR(255)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
32	11:29:56	RENAME TABLE Persons TO firstYearStudents	0 row(s) affected

5. DROP TABLE query syntax and example.

The screenshot shows the MySQL Workbench interface with the following details:

- Navigator:** Shows the schema structure. Under the 'students' schema, there is a table named 'firstYearStudents'.
- Query Editor (Query 1):** Contains the following SQL code:

```

6     lastName VARCHAR(25),
7     firstName VARCHAR(25),
8     address VARCHAR(255),
9     birthDate DATE
10    );
11 • DESC Persons;
12
13 • ALTER TABLE Persons ADD email VARCHAR(255);
14
15 • RENAME TABLE Persons TO firstYearStudents;
16
17 • DROP TABLE firstYearStudents;
18
19
20
21
22

```
- Information Panel:** Shows connection details for 'Local instance MySQL80 Host localhost Port 3306 User root' and the current session.
- Output Panel:** Displays the execution log with the following entries:

#	Time	Action	Message
29	11:25:08	CREATE TABLE Persons (rollNo INT, lastName VARCHAR(25), firstName VARCHAR(25))	0 row(s) affected
30	11:25:18	DESC Persons	5 row(s) returned
31	11:26:35	ALTER TABLE Persons ADD email VARCHAR(255)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
32	11:29:56	RENAME TABLE Persons TO firstYearStudents	0 row(s) affected
33	11:30:26	DROP TABLE firstYearStudents	0 row(s) affected

DBMS Practical no. 2

Title: Create a New Database and Perform Following operations on that Database.

- a) Create a table
- b) Insert values in that table
- c) Update the table
- d) Delete the contents of the table

Theory:

1. INSERT query syntax and example.

The INSERT INTO query is a DML command used to insert new records in a table.

Syntax:

It is possible to write the INSERT INTO statement in two ways.

- a. The first way specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- b. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. The INSERT INTO syntax would be as follows:

```
INSERT INTO table_name VALUES (value1, value2, value3, ...);
```

Example:

```
Insert into employee values (1, "abc", "cba");
```

```
Insert into employee (employee_id, f_name) values (1, "xyz");
```

2. UPDATE query syntax and example.

The UPDATE query is a DML command used to modify the existing records in a table.

Syntax: UPDATE *table_name*
 SET *column1* = *value1*, *column2* = *value2*, ...
 WHERE *condition*;

Example:

```
Update employee set joining_date = 01-01-2000;  
update employee set l_name = "zyx" where employee_id=2;
```

3. DELETE query syntax and example.

The DELETE query is a DML command used to delete existing records in a table.

Syntax: DELETE FROM *table_name* WHERE *condition*;

Example: DELETE FROM employee where ref_name = "abc";

4. Delete all contents of table

The TRUNCATE is a DDL command used to delete all rows of table, while keeping structure intact.

Syntax: TRUNCATE TABLE *database_name*;

Example: TRUNCATE TABLE employee;

OUTPUT:

1. Create a Table

The screenshot shows the MySQL Workbench interface. In the left sidebar, under 'Schemas', the 'company' schema is selected. Inside 'Tables', there is a single table named 'employee'. The 'Columns' section lists three columns: 'employeeID' (INT), 'firstName' (VARCHAR(20)), and 'lastName' (VARCHAR(20)). The 'Query 1' tab contains the following SQL code:

```

1 • CREATE DATABASE company;
2 • USE company;
3
4 • CREATE TABLE employee (
5     employeeID INT,
6     firstName VARCHAR(20),
7     lastName VARCHAR(20)
8 );
9 • DESC employee;
10

```

The 'Result Grid' pane shows the structure of the 'employee' table:

Field	Type	Null	Key	Default	Extra
employeeID	int	YES		NULL	
firstName	varchar(20)	YES		NULL	
lastName	varchar(20)	YES		NULL	

The 'Information' pane at the bottom displays connection details for a MySQL 8.0 instance on localhost port 3306, connected as root.

2. Insert Values in the table

The screenshot shows the MySQL Workbench interface. The 'company' schema is selected in the left sidebar. The 'employee' table is selected in the 'Tables' section. The 'Query 1' tab contains the following SQL code:

```

5
6     employeeID INT,
7     firstName VARCHAR(20),
8     lastName VARCHAR(20)
9 );
10
11 • INSERT INTO employee VALUES (1, "abc", "cba");
12 • INSERT INTO employee (employeeID, firstName) VALUES (1, "xyz");
13 • SELECT * FROM employee;
14

```

The 'Result Grid' pane shows the data inserted into the 'employee' table:

employeeID	firstName	lastName
1	abc	cba
1	xyz	NULL

The 'Information' pane at the bottom displays connection details for a MySQL 8.0 instance on localhost port 3306, connected as root.

3. Update the table

The screenshot shows the MySQL Workbench interface. In the Navigator pane, under the company schema, the employee table is selected. The Query 1 tab contains the following SQL code:

```

14
15 • ALTER TABLE employee ADD joining_date DATE;
16
17 • UPDATE employee SET joining_date = "2000-01-01";
18 • UPDATE employee SET lastName = "temp" WHERE firstName="xyz";
19 • SELECT * FROM employee;
20
21
22 • DESC employee;

```

The Result Grid shows the updated data:

	employeeID	firstName	lastName	joining_date
1	abc	cba	2000-01-01	
1	xyz	temp	2000-01-01	

The Action Output pane shows the history of actions:

#	Time	Action	Message
5	16:00:43	SELECT * FROM employee LIMIT 0, 1000	2 row(s) returned
6	16:00:52	ALTER TABLE employee ADD joining_date DATE	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
7	16:00:52	UPDATE employee SET joining_date = "2000-01-01"	2 row(s) affected Rows matched: 2 Changed: 2 Warnings: 0
8	16:00:52	UPDATE employee SET lastName = "temp" WHERE firstName="xyz"	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0
9	16:00:52	SELECT * FROM employee LIMIT 0, 1000	2 row(s) returned

4. Delete specific row/s

The screenshot shows the MySQL Workbench interface. In the Navigator pane, under the company schema, the employee table is selected. The Query 1 tab contains the following SQL code:

```

17 • UPDATE employee SET joining_date = "2000-01-01";
18 • UPDATE employee SET lastName = "temp" WHERE firstName="xyz";
19 • SELECT * FROM employee;
20
21 • DELETE FROM employee WHERE firstName = "abc";
22 • SELECT * FROM employee;
23
24 • TRUNCATE TABLE employee;
25 • SELECT * FROM employee;

```

The Result Grid shows the deleted data:

	employeeID	firstName	lastName	joining_date
1	xyz	temp	2000-01-01	

The Action Output pane shows the history of actions:

#	Time	Action	Message
7	16:00:52	UPDATE employee SET joining_date = "2000-01-01"	2 row(s) affected Rows matched: 2 Changed: 2 Warnings: 0
8	16:00:52	UPDATE employee SET lastName = "temp" WHERE firstName="xyz"	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0
9	16:00:52	SELECT * FROM employee LIMIT 0, 1000	2 row(s) returned
10	16:04:06	DELETE FROM employee WHERE firstName = "abc"	1 row(s) affected
11	16:04:06	SELECT * FROM employee LIMIT 0, 1000	1 row(s) returned

5. Delete all the rows of the table

The screenshot shows the MySQL Workbench interface. In the top-left, the Navigator pane displays the database schema, including the 'company' schema which contains the 'employee' table. The 'Tables' section under 'company' lists 'employee'. Below it are 'Columns', 'Indexes', 'Foreign Keys', and 'Triggers'. Other schemas like 'friends', 'sakila', 'students', 'sys', and 'world' are also listed. The bottom-left shows 'Information' and 'No object selected'. The main area is 'Query 1' with the following SQL code:

```

19 •   SELECT * FROM employee;
20
21 •   DELETE FROM employee WHERE firstName = "abc";
22 •   SELECT * FROM employee;
23
24 •   TRUNCATE TABLE employee;
25 •   SELECT * FROM employee;
26
27

```

The 'Result Grid' tab is active, showing a table with columns: employeeID, firstName, lastName, joining_date. The results of the last query (SELECT * FROM employee) are displayed.

In the bottom-right, the 'Output' pane for 'employee 5' shows the execution history:

#	Time	Action	Message
9	16:00:52	SELECT * FROM employee LIMIT 0, 1000	2 row(s) returned
10	16:04:06	DELETE FROM employee WHERE firstName = "abc"	1 row(s) affected
11	16:04:06	SELECT * FROM employee LIMIT 0, 1000	1 row(s) returned
12	16:08:00	TRUNCATE TABLE employee	0 row(s) affected
13	16:08:01	SELECT * FROM employee LIMIT 0, 1000	0 row(s) returned

DBMS Practical no. 3

Title: Create a table and apply following clauses on it: Where , Having ,Group by, Order by clauses.

Theory:

1. Where Clause :

Where clause is used to specify the condition of DDL and DML queries .

Syntax : **SELECT** column1, column2, ...

FROM table_name

WHERE condition;

Example : **SELECT * FROM employee**

`WHERE dept="IT";`

2. Having Clause :

Having clause is used with aggregate functions instead of Where clause.

Syntax : `SELECT column_name(s) FROM table_name
GROUP BY column_name(s)
HAVING condition;`

Example : `SELECT dept, count(emp_id) FROM employee
GROUP BY dept HAVING COUNT(*)>=2;`

3. Group By Clause :

This group by clause groups all rows that have the same values into summary rows. It is often used with aggregate functions to group the result -set by one or more columns.

Syntax: `SELECT column1, column2 FROM table_name WHERE condition
ORDER BY column1, column2... ASC|DESC`

Example : `SELECT count(*), dept FROM employee GROUP BY dept;`

4. Order by clause :

The order by is used to sort result set in ascending or descending order. It sorts record in ascending order by default .to sort record in descending order we used desc keyword .

Example : `SELECT emp_name, dept, salary FROM employee ORDER BY salary;
SELECT emp_name, dept, salary FROM employee ORDER BY salary DESC;`

Output:

Table state:

	emp_id	dept	emp_name	salary
▶	1	Marketing	Sam	10000
	2	IT	Clement	30000
	3	Marketing	Rohan	10000
	4	HR	Aman	20000
	5	Marketing	Striver	15000
	6	IT	Harry	50000

1. Example of query using where clause.

The screenshot shows the MySQL Workbench interface. The Navigator pane on the left lists databases like company, sakila, students, sys, and world. The SQL Editor pane in the center contains a script with several SQL statements, including INSERT INTO and SELECT queries. The Result Grid pane at the bottom shows the output of the last two SELECT statements, which retrieves employees from the IT department. The SQL Editor pane also displays the execution log for these queries.

Navigator

SCHEMAS

Filter objects

company

Tables

employee

Columns

Indexes

Foreign Keys

Triggers

Views

Stored Procedures

Functions

friends

sakila

students

sys

world

Administration Schemas

Information

Table: employee

Columns:

emp_id	int
dept	varchar(20)
emp_name	varchar(20)
salary	int

Object Info Session

SQL File 3*

Result Grid

emp_id	dept	emp_name	salary
2	IT	Clement	30000
6	IT	Harry	50000

employee 7 x

Action Output

#	Time	Action	Message
40	12:01:50	SELECT dept, count(emp_id) FROM employee GROUP BY dept HAVING COUNT(*)>=2 L...	2 row(s) returned
41	12:02:29	SELECT * FROM employee WHERE dept='IT' LIMIT 0, 1000	2 row(s) returned

2. Example of query using having clause

Navigator

SCHEMAS

- company
 - Tables
 - employee
 - Columns
 - Indexes
 - Foreign Keys
 - Triggers
 - Views
 - Stored Procedures
 - Functions
- friends
- sakila
- students
- sys
- world

Administration Schemas

Information

Table: employee

Columns:

emp_id	int
dept	varchar(20)
emp_name	varchar(20)
salary	int

Object Info **Session**

SQL File 3*

```

9 • INSERT INTO employee VALUES(2,"IT","Clement",3000);
10 • INSERT INTO employee VALUES(3,"Marketing","Rohan",1000);
11 • INSERT INTO employee VALUES(4,"HR","Aman",2000);
12 • INSERT INTO employee VALUES(5,"Marketing","Striver",15000);
13 • INSERT INTO employee VALUES(6,"IT","Harry",5000);
14 • SELECT * FROM employee;
15
16 • SELECT * FROM employee WHERE dept="IT";
17
18 • SELECT dept, count(emp_id) FROM employee GROUP BY dept HAVING COUNT(*)>=2;
19
20

```

Result Grid | Filter Rows: [] Export: [] Wrap Cell Content: []

dept	count(emp_id)
Marketing	3
IT	2

Result 8

Output

Action Output

#	Time	Action	Message
41	12:02:29	SELECT * FROM employee WHERE dept="IT" LIMIT 0, 1000	2 row(s) returned
42	12:03:22	SELECT dept, count(emp_id) FROM employee GROUP BY dept HAVING COUNT(*)>=2 L...	2 row(s) returned

3. Example of query using Group by clause.

Navigator

SCHEMAS

- company
 - Tables
 - employee
 - Columns
 - Indexes
 - Foreign Keys
 - Triggers
 - Views
 - Stored Procedures
 - Functions
- friends
- sakila
- students
- sys
- world

Administration Schemas

Information

Table: employee

Columns:

emp_id	int
dept	varchar(20)
emp_name	varchar(20)
salary	int

Object Info **Session**

SQL File 3*

```

15
16 • SELECT * FROM employee WHERE dept="IT";
17
18 • SELECT dept, count(emp_id) FROM employee GROUP BY dept HAVING COUNT(*)>=2;
19
20 • SELECT count(*), dept FROM employee GROUP BY dept;
21
22
23
24
25

```

Result Grid | Filter Rows: [] Export: [] Wrap Cell Content: []

count(*)	dept
3	Marketing
2	IT
1	HR

Result 10

Output

Action Output

#	Time	Action	Message
43	12:06:05	SELECT count(*) FROM employee GROUP BY dept LIMIT 0, 1000	3 row(s) returned
44	12:06:14	SELECT count(*), dept FROM employee GROUP BY dept LIMIT 0, 1000	3 row(s) returned

4. Example of query using order by clause.

Navigator

SCHEMAS

- company
 - Tables
 - employee
 - Columns
 - Indexes
 - Foreign Keys
 - Triggers
- Views
- Stored Procedures
- Functions

- friends
- sakila
- students
- sys
- world

Administration Schemas

Information

Table: employee

Columns:

emp_id	int
dept	varchar(20)
emp_name	varchar(20)
salary	int

Object Info **Session**

SQL File 3*

```

16 •   SELECT * FROM employee WHERE dept="IT";
17
18 •   SELECT dept, count(emp_id) FROM employee GROUP BY dept HAVING COUNT(*)>=2;
19
20 •   SELECT count(*), dept FROM employee GROUP BY dept;
21
22 •   SELECT emp_name, dept, salary FROM employee ORDER BY salary;
23
24 •   SELECT emp_name, dept, salary FROM employee ORDER BY salary DESC;
25
26
27

```

Result Grid | Filter Rows: Export: Wrap Cell Content:

emp_name	dept	salary
Sam	Marketing	10000
Rohan	Marketing	10000
Striver	Marketing	15000
Aman	HR	20000
Clement	IT	30000
Harry	IT	50000

employee 11 x

Output

Action Output

#	Time	Action	Message
44	12:06:14	SELECT count(*), dept FROM employee GROUP BY dept LIMIT 0, 1000	3 row(s) returned
45	12:09:31	SELECT emp_name, dept, salary FROM employee ORDER BY salary LIMIT 0, 1000	6 row(s) returned

Navigator

SCHEMAS

- company
 - Tables
 - employee
 - Columns
 - Indexes
 - Foreign Keys
 - Triggers
- Views
- Stored Procedures
- Functions

- friends
- sakila
- students
- sys
- world

Administration Schemas

Information

Table: employee

Columns:

emp_id	int
dept	varchar(20)
emp_name	varchar(20)
salary	int

Object Info **Session**

SQL File 3*

```

15
16 •   Execute the selected portion of the script or everything, if there is no selection
17
18 •   SELECT dept, count(emp_id) FROM employee GROUP BY dept HAVING COUNT(*)>=2;
19
20 •   SELECT count(*), dept FROM employee GROUP BY dept;
21
22 •   SELECT emp_name, dept, salary FROM employee ORDER BY salary;
23
24 •   SELECT emp_name, dept, salary FROM employee ORDER BY salary DESC;
25
26

```

Result Grid | Filter Rows: Export: Wrap Cell Content:

emp_name	dept	salary
Harry	IT	50000
Clement	IT	30000
Aman	HR	20000
Striver	Marketing	15000
Sam	Marketing	10000
Rohan	Marketing	10000

employee 12 x

Output

Action Output

#	Time	Action	Message
45	12:09:31	SELECT emp_name, dept, salary FROM employee ORDER BY salary LIMIT 0, 1000	6 row(s) returned
46	12:10:40	SELECT emp_name, dept, salary FROM employee ORDER BY salary DESC LIMIT 0, 1000	6 row(s) returned

DBMS Practical no. 4

Title: Implement the following Functions in SQL a) Date functions b) Time functions c) String functions d) Aggregate functions.

Theory:

A. Date and Time function :

Date Function :

```
SELECT NOW();  
SELECT CURDATE();  
SELECT DATE(SYSDATE());  
SELECT DATE_FORMAT("2020-10-18", "%d/%M/%Y");  
SELECT DATEDIFF("2020-10-18", "2003-03-12");  
SELECT CURTIME();  
SELECT DATE_ADD("2020-10-18", INTERVAL 10 DAY);  
SELECT DATE_SUB("2020-10-18", INTERVAL 10 DAY);  
SELECT TIME(SYSDATE());  
SELECT DAYNAME("2020-10-18");
```

Time Function :

```
SELECT current_time();  
SELECT TIME_TO_SEC("19:30:10");
```

B. String Function :

```
SELECT ASCII("R");  
SELECT CHAR(82);  
SELECT LENGTH(emp_name), emp_name from employee;  
SELECT BIT_LENGTH("Samuel"), emp_name from employee;  
SELECT CONCAT(emp_name, " works in ", dept) from employee;
```

```
SELECT FIELD("s", "h", "a", "r", "p"), emp_name from employee;  
SELECT FORMAT(88.38439, 2), emp_name from employee;  
SELECT LOWER(emp_name), emp_name from employee;  
SELECT UPPER(emp_name), emp_name from employee;  
SELECT LEFT(emp_name, 3), emp_name from employee;
```

C. Aggregate functions :

```
SELECT MIN(salary) FROM employee;  
SELECT MIN(salary) FROM employee WHERE dept="IT";  
  
SELECT MAX(salary) FROM employee;  
SELECT MAX(salary) FROM employee WHERE dept="Marketing";  
  
SELECT AVG(salary) FROM employee;  
SELECT AVG(salary) FROM employee WHERE dept IN("HR", "IT");  
  
SELECT SUM(salary) FROM employee;  
SELECT SUM(salary) FROM employee WHERE dept="Marketing";  
  
SELECT COUNT(*) FROM employee;  
SELECT COUNT(*) FROM employee WHERE dept != "HR";
```

Output:

1. Date and time functions:

Result Grid	
<code>NOW()</code>	2020-10-20 12:59:43
<code>CURDATE()</code>	2020-10-20

Result Grid	
<code>DATE(SYSDATE())</code>	2020-10-20

DATE_FORMAT("2020-10-18", "%d/%m/%Y")	DATEDIFF("2020-10-18", "2003-03-12")	CURTIME()
18/October/2020	6430	12:59:43
DATE_ADD("2020-10-18", INTERVAL 10 DAY)	DATE_SUB("2020-10-18", INTERVAL 10 DAY)	TIME(SYSDATE())
2020-10-28	2020-10-08	12:59:44
DAYNAME("2020-10-18")	current_time()	TIME_TO_SEC("19:30:10")
Sunday	13:10:40	70210

2. String functions:

Result Grid	ASCII("R")	CHAR(82)	Result Grid
	82	BLOB	
BIT_LENGTH("Samuel")	emp_name	CONCAT(emp_name, " works in ", dept)	
48	Sam	Sam works in Marketing	
48	Clement	Clement works in IT	
48	Rohan	Rohan works in Marketing	
48	Aman	Aman works in HR	
48	Striver	Striver works in Marketing	
FIELD("s", "h", "a", "r", "p")	emp_name	FORMAT(88.38439, 2)	Result Grid
0	Sam	88.38	
0	Clement	88.38	
0	Rohan	88.38	
0	Aman	88.38	
0	Striver	88.38	
LOWER(emp_name)	emp_name	LEFT(emp_name, 3)	Result Grid
sam	Sam	Sam	
clement	Clement	Cle	
rohan	Rohan	Roh	
aman	Aman	Ama	
striver	Striver	Str	
UPPER(emp_name)	emp_name		
SAM	Sam		
CLEMENT	Clement		
ROHAN	Rohan		
AMAN	Aman		
STRIVER	Striver		

3. Aggregate functions:

Employee table:

	emp_id	dept	emp_name	salary
▶	1	Marketing	Sam	10000
	2	IT	Clement	30000
	3	Marketing	Rohan	10000
	4	HR	Aman	20000
	5	Marketing	Striver	15000
	6	IT	Harry	50000

7 •	SELECT MIN(salary) FROM employee;	7 •	SELECT MIN(salary) FROM employee;
8 •	SELECT MIN(salary) FROM employee WHERE dept="IT";	8 •	SELECT MIN(salary) FROM employee WHERE dept="IT";
9		9	

Result Grid	Filter Rows: []	Export: []	Wrap Cell Content: []
MIN(salary)			
▶ 10000			

10 •	SELECT MAX(salary) FROM employee;	11 •	SELECT MAX(salary) FROM employee WHERE dept="Marketing";		
11 •	SELECT MAX(salary) FROM employee WH	11 •	SELECT MAX(salary) FROM employee WHERE dept="Marketing";		
Result Grid	Filter Rows: []	Exp	Result Grid	Filter Rows: []	Exp
MAX(salary)			MAX(salary)		
▶ 50000			▶ 15000		

14 •	SELECT AVG(salary) FROM employee WHERE dept IN("HR", "IT");		
Result Grid	Filter Rows: []	Export: []	Wrap Cell Content: []
AVG(salary)			
▶ 22500.0000			

16 •	SELECT SUM(salary) FROM employee;	19 •	SELECT COUNT(*) FROM employee;		
Result Grid	Filter Rows: []	Exp	Result Grid	Filter Rows: []	Exp
SUM(salary)			COUNT(*)		
▶ 135000			▶ 6		

17 •	SELECT SUM(salary) FROM employee WHERE dept="Marketing";		
Result Grid	Filter Rows: []	Export: []	Wrap Cell Content: []
SUM(salary)			
▶ 35000			

20 •	SELECT COUNT(*) FROM employee WHERE dept != "HR";		
Result Grid	Filter Rows: []	Export: []	Wrap Cell Content: []
COUNT(*)			
▶ 5			

DBMS Practical no. 5

Title: Implementation of all types of Joins.

Theory:

1. Example of query using INNER JOIN.

The inner join keyword selects records that have same values in both tables.

Syntax : Select column_name() from table_name innerjoin table_name2
 on table_name.column_name=table_name2.column_name;

Example : SELECT id, loanID, branch, amount FROM customer INNER JOIN loans ON id=loans.cID;

2. Example of query using LEFT OUTER JOIN.

Left join keyword returns all records from the table (table1) , and the matched records from the right table(table2). The results is NULL from the right side if there is no match.

Syntax : Select column_name() from table_name leftjoin table_name2
 on table_name.column_name=table_name2.column_name;

Example : SELECT id, custName, loanID, branch, amount FROM customer LEFT OUTER JOIN loans
 ON customer.ID=loans.cID;

3. Example of query using RIGHT OUTER JOIN.

The right join keyword returns all records from the right table (table2) , and the matched records from the left table(table1). The results is NULL from the left side if there is no match.

Syntax : Select column_name() from table_name1right join table_name2on
 table_name1.column_name=table_name2.column_name;

Example : SELECT id, custName, loanID, branch, amount FROM customer RIGHT OUTER JOIN loans
 ON customer.ID=loans.cID;

4. Example of query using FULL OUTER JOIN.

The full outer join keyword returns all records when there ia a match in left(table1) or right(table2) table records.

Syntax : Select column_name(s) from table_name1 full join table_name2

on table_name1.column_name=table_name2.column_name where condition;

Example : SELECT id, custName, loanID, branch, amount FROM customer FULL OUTER JOIN loans
ON customer.ID=loans.cID;

Output:

Customer table:

	id	custName	loanID	balance
▶	1	Max	1	200000
	2	Merry	2	180000
	3	Michal	3	250000
*	4	Jimmy	NULL	400000
*	NULL	NULL	NULL	NULL

Loan table:

	loanIDs	cID	branch	amount
▶	1	1	Bangalore	100000
	2	3	Mumbai	150000
	3	2	Banglore	80000
*	NULL	NULL	NULL	NULL

1. Inner join

	id	loanID	branch	amount
▶	1	1	Bangalore	100000
	3	3	Mumbai	150000
	2	2	Banglore	80000

2. Left outer join:

	id	custName	loanID	branch	amount
▶	1	Max	1	Bangalore	100000
	2	Merry	2	Banglore	80000
	3	Michal	3	Mumbai	150000
*	4	Jimmy	NULL	NULL	NULL

3. Right outer join

	id	custName	loanID	branch	amount
▶	1	Max	1	Bangalore	100000
	3	Michal	3	Mumbai	150000
*	2	Merry	2	Banglore	80000

Title: Create table and Apply constraints such as NOT NULL, UNIQUE, Check, Default, Primary key ,Foreign key, on the table.

Theory:

- Creating Table:- Tables can be created inside Database which is under use.

```
CREATE TABLE tableName (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    ....
);
```

Eg. `CREATE TABLE firstYear(
 firstName varchar(20),
 lastName varchar(20),
 enrolmentNo varchar(9),
 birthDate DATE,
 CoursesEnrolled int
);`

- Constraints possible for columns of table:

- NOT NULL: States that column with this constraint cannot be null.
- DEFAULT: Sets a default value for column if value not specified.
- CHECK: Gives a condition which must be satisfied by the value to be placed.
- UNIQUE: Ensures that all values in a column are different.
- PRIMARY KEY: Ensures that value is Unique and NOT NULL, It is used to uniquely identify the row
- FOREIGN KEY: Refers to PRIMARY KEY of another table i.e. uniquely identify a row in other table.

Eg. `CREATE TABLE firstYear(
 firstName varchar(20) NOT NULL,
 lastName varchar(20),
 enrolmentNo varchar(9) PRIMARY KEY,
 age INT NOT NULL,
 coursesEnrolled INT DEFAULT 0,
 libraryID INT UNIQUE,
 favCourse VARCHAR(20),
 CHECK (age>=15)
);`

Example of queries:

- `INSERT INTO firstyear VALUES("Omkar", "Phansopkar", "FS19C0042", 15, 5, 2042, "Java");`
-- This query is valid
- `INSERT INTO firstyear VALUES("Sam", "Johnson", "FS19C0067", 13, 8, 2067, "Python");`
-- This query is not valid, since the age is 13 i.e. less than 15
- `INSERT INTO firstyear(firstName, enrolmentNo, age, libraryID, favCourse)
VALUES("Lokey", "FS19C0070", 20, 9, "DBMS")`
-- This query is valid
- `INSERT INTO firstyear(firstName, lastName, enrolmentNo, age, libraryID, favCourse)
VALUES("Josh", "Hazlewood", "FS19C0070", 15, 15, "JS");`
-- This query is not valid since the enrolmentNo which is primary key, has been given a duplicate value same as that of Lokey
- `INSERT INTO firstyear(firstName, lastName, enrolmentNo, age, libraryID, favCourse)
VALUES("Sam", "Curran", "FS19C0080", 20, 15, "MP");`
-- This query is valid

Exported CSV form of table:

firstName, lastName,	enrolmentNo,	age,	coursesEnrolled,	libraryID,	favCourse
Omkar, Phansopkar,	FS19C0042,	15,	5,	2042,	Java
Lokey, ,	FS19C0070,	20,	0,	9,	DBMS
Sam, Curran,	FS19C0080,	20,	0,	15,	MP

Queries:

```
3  CREATE TABLE firstyear(
4      firstName VARCHAR(20) NOT NULL,
5      lastName VARCHAR(20),
6      enrolmentNo VARCHAR(9) PRIMARY KEY,
7      age INT NOT NULL,
8      coursesEnrolled INT DEFAULT 0,
9      libraryID INT UNIQUE,
10     favCourse VARCHAR(20) DEFAULT "NONE",
11     CHECK (age>=15)
12 );
13
14 DESC TABLE firstyear;
15
16 INSERT INTO firstyear VALUES("Omkar", "Phansopkar", "FS19CO042", 15, 5, 2042, "Java");
17
18 INSERT INTO firstyear VALUES("Sam", "Johnson", "FS19CO067", 13, 8, 2067, "Python");
19
20 INSERT INTO firstyear(firstName, enrolmentNo, age, libraryID, favCourse)
21     VALUES("Lokey", "FS19CO070", 20, 9, "DBMS");
22
23 INSERT INTO firstyear(firstName, lastName, enrolmentNo, age, libraryID, favCourse)
24     VALUES("Josh", "Hazlewood", "FS19CO070", 15, 15, "JS");
25
26 INSERT INTO firstyear(firstName, lastName, enrolmentNo, age, libraryID, favCourse)
27     VALUES("Sam", "Curran", "FS19CO080", 20, 15, "MP");
28
29
30 SELECT * FROM firstyear;
31
32
```

Output:

Success

1:03 PM

2.08 seconds

Explore

SQL

Data

Chart

Export ▾



Activate Windows

Rows affected: 0

Go to Settings to activate Windows.

Success

1:03 PM

0.688 seconds

Explore

SQL

Data

Chart

Export ▾



id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	firstyear	NULL	ALL	NULL	NULL	NULL	NULL

Success

1:03 PM

1.403 seconds

Explore

SQL

Data

Chart

Export ▾



Rows affected: 1

Failed

1:03 PM

Explore

SQL

Data

Chart

Export ▾



UNKNOWN_CODE_PLEASE_REPORT: Check constraint 'firstyear_chk_1' is violated.

Success

1:03 PM

1.403 seconds

Explore

SQL

Data

Chart

Export ▾



Rows affected: 1

Failed

1:04 PM

Explore

SQL

Data

Chart

Export ▾



ER_DUP_ENTRY: Duplicate entry 'FS19CO070' for key 'firstyear.PRIMARY'

Success

1:03 PM

1.403 seconds

Explore

SQL

Data

Chart

Export ▾



Rows affected: 1

Success

3 rows

1:04 PM

0.717 seconds

Explore

SQL

Data

Chart

Export ▾



firstName	lastName	enrolmentNo	age	coursesEnrolled	libraryID	favCourse
Omkar	Phansopkar	FS19CO042	15	5	2042	Java
Lokey	NULL	FS19CO070	20	0	9	DBMS
Sam	Curran	FS19CO080	20	0	15	MP

Title: Write SQL code for creating of View. Perform Insert ,Modify, Delete records through view, Delete the View. Working with Nested -Query.

Theory:

1. Simple View

→Simple View in SQL is the view created by involving only single table. In other words we can say that there is only one base table in case of Simple View in SQL

Syntax : **CREATE VIEW** view-name **AS**
 SELECT column-name
 FROM table-name
 WHERE condition

Example : **CREATE VIEW** Brazil_Customers **AS**
 SELECT CustomerName, ContactName
 FROM Customers
 WHERE Country = 'Brazil';

2. Complex view

→On other hand, Complex View is created by involving more than one table i.e., multiple tables get projected in Complex view.

Syntax : **CREATE VIEW** view-name **AS**
 SELECT column-name
 FROM table-name
 JOIN aggregate-function
 GROUP BY column-name

Example : **CREATE VIEW** dept_income **AS**
 SELECT d.Name as DepartmentName, sum(e.salary) as
 TotalSalary
 FROM Employees e
 JOIN Departments d on e.DepartmentId = d.id
 GROUP BY d.Name

3. INSERT query syntax and example.

→ *INSERT* command is used to insert data into the row of a table.

Syntax : **INSERT INTO TABLE_NAME (col1, col2, col3,.. col N)
VALUES (value1, value2, value3,.... valueN);**

Example : **INSERT INTO fyfs (name, roll) VALUES ("Rupesh", "FS43");**

4. UPDATE query syntax and example.

→ *UPDATE* command is used to update or modify the value of a column in the table.

Syntax : **UPDATE table_name SET [column_name1=value1] [WHERE CONDITION]**

Example : **UPDATE students SET User_Name = "Rupesh" WHERE
Student_Id = '43'**

5. DELETE query syntax and example.

→ *DELETE* is used to remove one or more row from a table.

Syntax : **DELETE FROM table_name [WHERE condition];**

Example : **DELETE FROM javatpoint WHERE Author="Rupesh";**

6. DROP VIEW

→ *DROP VIEW* command is used to delete the view

Syntax : **DROP VIEW view_name**

Example : **DROP VIEW [Brazil Customers]**

7. NESTED QUERY

→Query written inside a query is called as SQL Nested Query

Syntax : **SELECT Column1,Column2... From Table_Name
WHERE Column_Name Operator(Select Column1,Column
From Table_Name_2)
Operator (Select Column1,Column2.....From
Table_Name_3)**

Example : **SELECT Model FROM Product
WHERE ManufacturerID IN (SELECT ManufacturerID
FROM
Manufacturer
WHERE Manufacturer = 'Dell')**

Outputs:

Initial tables setup:

Students table:

	name	marks	favSubject
▶	Sam Johnson	70 60	Python DBMS
	Markus	70	Java
	Stoinis	60	DBMS
	Smith	80	Python
	Williamson	30	Python
	Emily	20	Java
	Deviliers	90	JavaScript

academicSubjects table:

	sub_name	studentRating	teachingSemester
▶	DBMS	6.5	3
	Java	7	3
	Javascript	7.9	4
	Python	8.5	3
*	NULL	NULL	NULL

Creating Simple view:

The screenshot shows a database management interface with two tabs: "SQL File 4*" and "SQL File 5*". The left pane, titled "Navigator", displays the "SCHEMAS" tree, which includes "company", "friends", "sakila", and "students". The "students" schema is expanded, showing "Tables" (academicsubjects, students), "Views" (passedstudents), and "Functions". The right pane contains the following SQL code:

```
13  
14    -- Simple View  
15 • CREATE VIEW passedStudents AS SELECT * FROM students WHERE marks>=35;  
16 • SELECT * FROM passedStudents;  
17 • CREATE VIEW failedStudents AS SELECT * FROM students WHERE marks<35;  
18 • SELECT * FROM failedStudents;  
19  
20    -- Complex View
```

Below the code is a "Result Grid" table with the following data:

	name	marks	favSubject
▶	Sam	70	Python
	Johnson	60	DBMS
	Markus	70	Java
	Stoinis	60	DBMS
	Smith	80	Python
	Devilliers	90	JavaScript

The screenshot shows a database management interface with two tabs: "SQL File 4*" and "SQL File 5*". The left pane, titled "Navigator", displays the "SCHEMAS" tree, which includes "company", "friends", "sakila", and "students". The "students" schema is expanded, showing "Tables" (academicsubjects, students), "Views" (passedstudents), and "Functions". The right pane contains the following SQL code:

```
13  
14    -- Simple View  
15 • CREATE VIEW passedStudents AS SELECT * FROM students WHERE marks>=35;  
16 • SELECT * FROM passedStudents;  
17 • CREATE VIEW failedStudents AS SELECT * FROM students WHERE marks<35;  
18 • SELECT * FROM failedStudents;  
19  
20    -- Complex View
```

Below the code is a "Result Grid" table with the following data:

	name	marks	favSubject
▶	Williamson	30	Python
	Emily	20	Java

Creating complex view:

Schemas

```

19
20    -- Complex View
21 •   CREATE VIEW subjectToMarks AS SELECT favSubject, AVG(marks) FROM students GROUP BY favSubject;
22 •   SELECT * FROM subjectToMarks;
23 •   CREATE VIEW studentsToFavSubject AS SELECT favSubject, COUNT(*) AS No_of_students FROM students GROUP BY favSubject;
24 •   SELECT * FROM studentsToFavSubject;
25
26

```

Result Grid | Filter Rows: [] Export: [] Wrap Cell Content: []

favSubject	AVG(marks)
Python	60.0000
DBMS	60.0000
Java	45.0000
JavaScript	90.0000

Administration Schemas Information

Schemas

```

19
20    -- Complex View
21 •   CREATE VIEW subjectToMarks AS SELECT favSubject, AVG(marks) FROM students GROUP BY favSubject;
22 •   SELECT * FROM subjectToMarks;
23 •   CREATE VIEW studentsToFavSubject AS SELECT favSubject, COUNT(*) AS No_of_students FROM students GROUP BY favSubject;
24 •   SELECT * FROM studentsToFavSubject;
25
26

```

Result Grid | Filter Rows: [] Export: [] Wrap Cell Content: []

favSubject	No_of_students
Python	3
DBMS	2
Java	2
JavaScript	1

Administration Schemas Information

No object selected

studentsToFavSubject 57 x

Output

Insert delete and update on view:

Schemas

```

26
27    -- Insert, delete, update query on view
28 •   INSERT INTO passedStudents VALUES("Jonty", 36, "Javascript");
29 •   UPDATE passedStudents SET marks = 75 WHERE name = "Stoinis";
30 •   DELETE FROM passedStudents WHERE name = "Sam";
31 •   SELECT * FROM passedStudents;      -- Should reflect changes
32
33 •   DROP VIEW passedStudents;

```

Result Grid | Filter Rows: [] Export: [] Wrap Cell Content: []

name	marks	favSubject
Johnson	60	DBMS
Markus	70	Java
Stoinis	75	DBMS
Smith	80	Python
Devilliers	90	JavaScript
Jonty	36	Javascript

Administration Schemas Information

Drop view:

SCHEMAS

- company
- friends
- sakila
- students**
 - Tables
 - academicsubjects
 - students
 - Views
 - failedstudents
 - studentstofavsubject
 - subjectmarks
 - Stored Procedures
 - Functions
- sys
- world

Administration Schemas Information

```

31 • SELECT * FROM passedStudents; -- Should reflect changes
32
33 • DROP VIEW passedStudents;
34 • SELECT * FROM passedStudents; -- Should give error
35
36 -- Preparing 2nd table
37 • CREATE TABLE academicSubjects(
38     sub_name VARCHAR(20) PRIMARY KEY,
39     studentRating FLOAT,

```

Output

#	Time	Action
1	18:07:34	DROP VIEW passedStudents
2	18:07:34	SELECT * FROM passedStudents LIMIT 0, 1000

Message
0 row(s) affected
Error Code: 1146. Table 'students.passedstudents' doesn't exist

Nested query on single table:

SCHEMAS

- company
- friends
- sakila
- students**
 - Tables
 - academicsubjects
 - students
 - Views
 - failedstudents
 - studentstofavsubject
 - subjectmarks
 - Stored Procedures
 - Functions
- sys
- world

Administration Schemas Information

No object selected

```

50
51 -- Nested query on single table
52 -- Gives marks of students with marks more than max marks of students having favSubjects as DBMS or Java
53 • SELECT marks, name FROM students WHERE marks > ( SELECT MAX(marks) FROM students WHERE favSubject IN("DBMS", "Java") );
54

```

Result Grid

marks	name
80	Smith
90	Devilliers

Output

#	Time	Action
1	18:11:06	SELECT marks, name FROM students WHERE marks > (SELECT MAX(marks) FROM stud... 2 row(s) returned

Nested query using two tables:

SCHEMAS

- company
- friends
- sakila
- students**
 - Tables
 - academicsubjects
 - students
 - Views
 - failedstudents
 - studentstofavsubject
 - subjectmarks
 - Stored Procedures
 - Functions
- sys
- world

Administration Schemas Information

No object selected

```

55
56 -- Nested query using two tables
57 -- Gives students having favSubject having rating over 7 ( Ratings are in other table )
58 • SELECT * FROM students WHERE FavSubject IN( SELECT sub_name FROM academicSubjects WHERE studentRating > 7 );
59

```

Result Grid

name	marks	favSubject
Smith	80	Python
Willerson	30	Python
Devilliers	90	JavaScript
Jony	36	Javascript

Output

#	Time	Action
1	18:11:06	SELECT marks, name FROM students WHERE marks > (SELECT MAX(marks) FROM stud... 2 row(s) returned
2	18:14:41	SELECT * FROM students WHERE favSubject IN(SELECT sub_name FROM academicSu... 4 row(s) returned

Conclusion: Thus, we created, modified(Inserted updated deleted) and deleted views

Title : Implementation of DCL commands

- GRANT
- REVOKE

Implementation of TCL commands

- COMMIT
- ROLLBACK
- SAVEPOINT

Theory :**1. DCL :**

DCL is the abstract of Data Control Language. Data Control Language includes commands such as GRANT, and is concerned with rights, permissions, and other controls of the database system. DCL is used to grant/revoke permissions on databases and their contents. DCL is simple, but MySQL permissions are a bit complex. DCL is about security. DCL is used to control the database transaction. DCL statements allow you to control who has access to a specific object in your database.

1. GRANT
2. REVOKE

GRANT:

It provides the user's access privileges to the database. The MySQL database offers both the administrator and user a great extent of the control options. The administration side of the process includes the possibility for the administrators to control certain user privileges over the MySQL server by restricting their access to an entire database or usage limiting permissions for a specific table. It creates an entry in the security system that allows a user in the current database to work with data in the current database or execute specific statements.

Syntax :

Statement permissions:

```
GRANT { ALL | statement [ ,...n ] }  
TO security_account [ ,...n ]
```

Normally, a database administrator first uses CREATE USER to create an account, then GRANT to define its privileges and characteristics.

For example:

```
CREATE USER vatsa@'localhost' IDENTIFIED BY 'mypass';  
GRANT ALL ON MY_TABLE TO vatsa@'localhost';  
GRANT SELECT ON Users TO vatsa@'localhost';
```

REVOKE:

The REVOKE statement enables system administrators and to revoke (back permission) the privileges from MySQL accounts.

Syntax:

```
REVOKE  
priv_type [(column_list)]  
, priv_type [(column_list)] ...  
ON [object_type] priv_level  
FROM user [, user] ...  
  
REVOKE ALL PRIVILEGES, GRANT OPTION  
FROM user [, user] ...
```

For example:

```
REVOKE INSERT ON *.* FROM 'vatsa'@'localhost';
```

2. TCL :

Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.

COMMIT:

COMMIT command is used to permanently save any transaction into the database.

When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Following is commit command's syntax,

Syntax:

```
commit;
```

ROLLBACK:

This command restores the database to the last committed state. It is also used with the SAVEPOINT command to jump to a savepoint in an ongoing transaction.

If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not committed using the COMMIT command.

Following is rollback command's syntax,

Syntax:

```
ROLLBACK TO savepoint_name;
```

SAVEPOINT:

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

Syntax:

```
SAVEPOINT savepoint_name;
```

OUTPUT :

1. Practical no. 9 DCL

1) GRANT

01.

Creating user

02.

Granting permissions

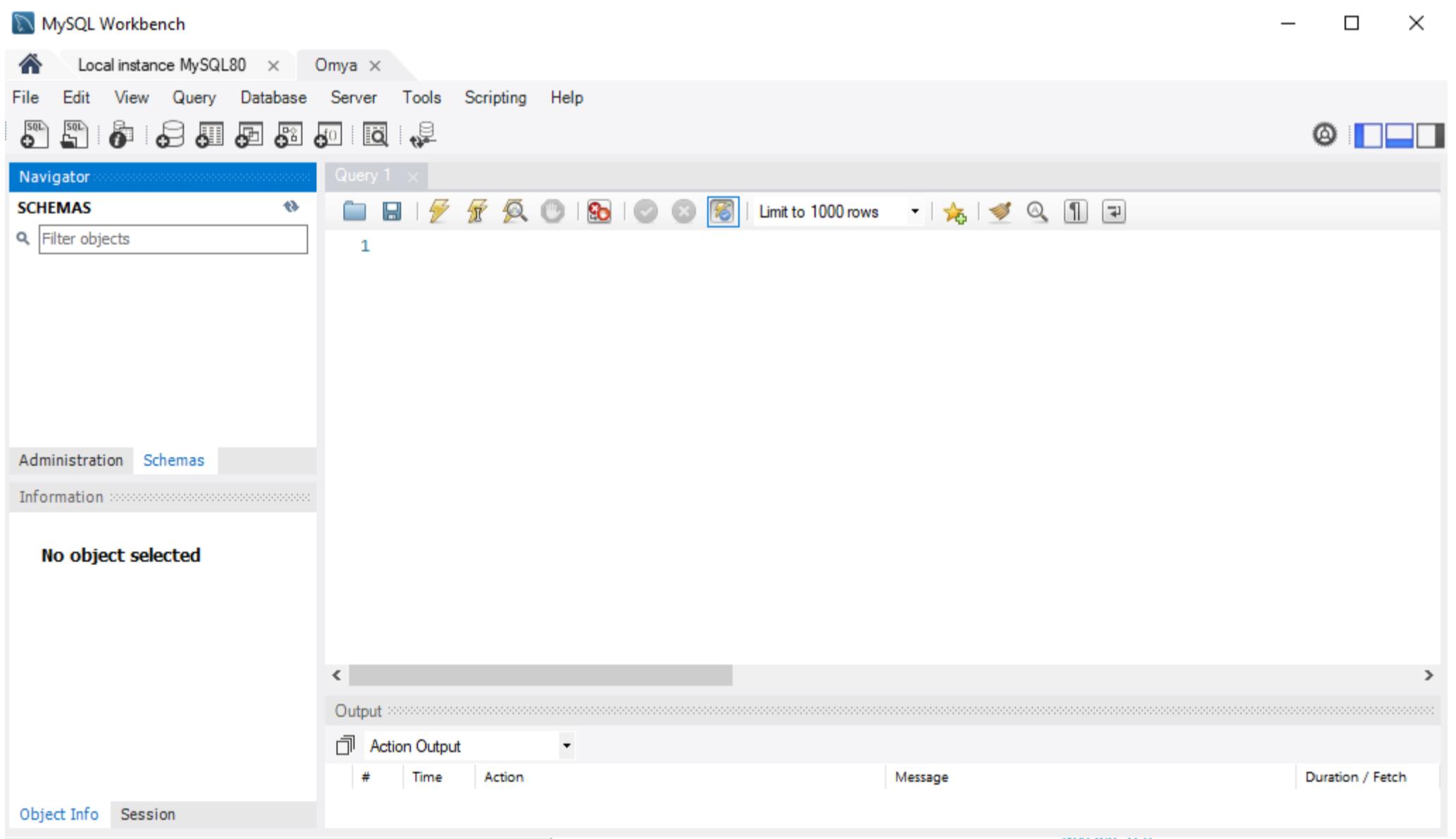
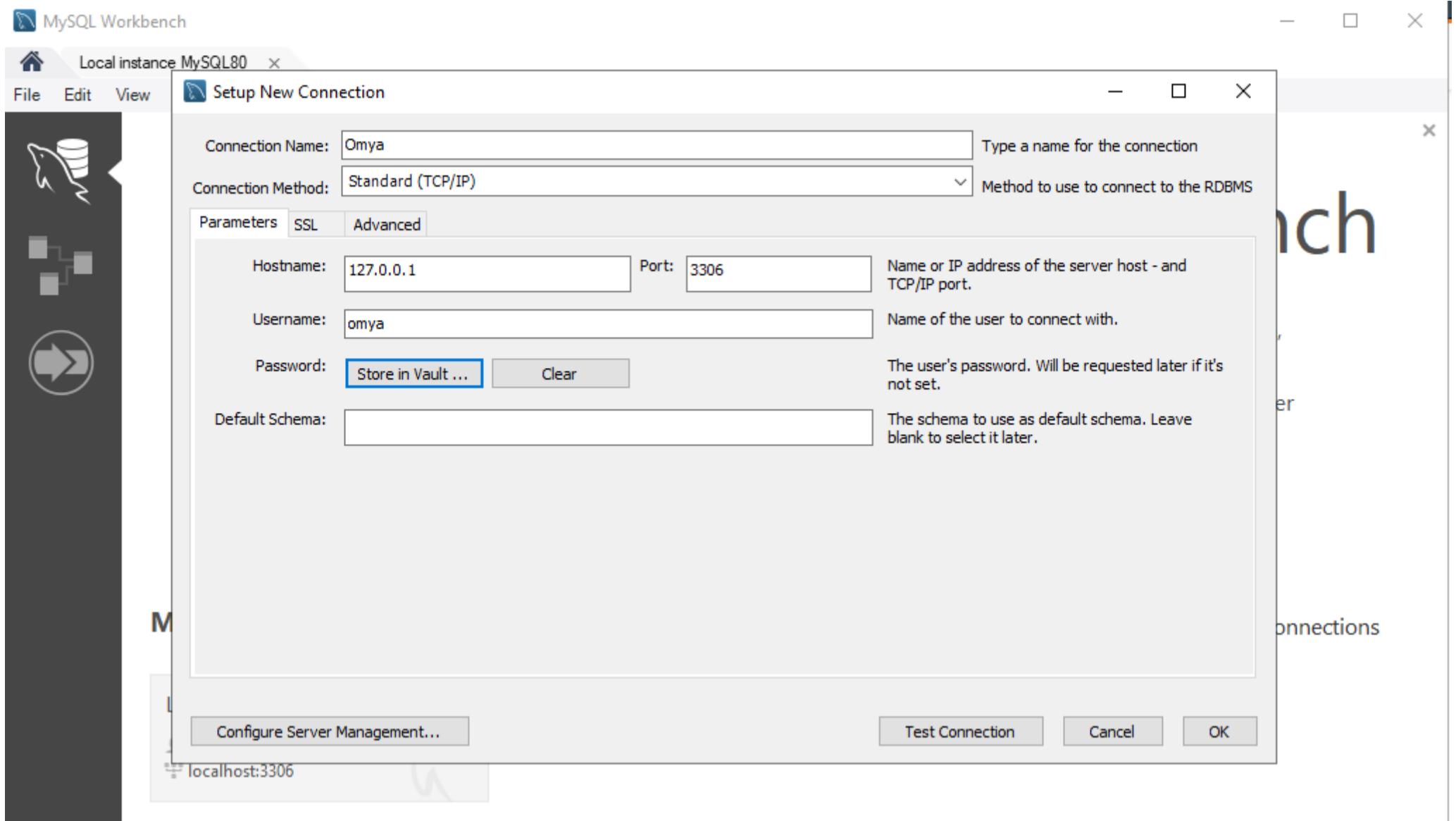
The screenshot shows the MySQL Workbench interface. In the top navigation bar, the title is "MySQL Workbench" and the connection is "Local instance MySQL80". The main area has two tabs: "SQL File 4*" and "SQL File 3*". The "SQL File 4*" tab contains the following SQL code:

```
1 • CREATE USER "omya"@"127.0.0.1" IDENTIFIED BY "omya";
2 • GRANT SELECT ON students.academicsubjects TO "omya"@"127.0.0.1";
3 • SHOW GRANTS FOR "omya"@"127.0.0.1";
4
5
```

Below the code, there is a "Result Grid" section with the message "Grants for omya@127.0.0.1" and a "GRANT USAGE ON *.* TO `omya` @`127.0.0.1`" entry. A tooltip "Export recordset to an external file" is visible over the export button. The bottom section shows the "Result 1" tab with the output of the query "select current_user() LIMIT 0, 1000", which returned 1 row(s) and took 0.000 sec / 0.000 sec.

03.

Making new connection



04.

Use allowed commands

Automatically synchronize your data using Data Sync Tool : Reason #28 to upgrade

Query 1 History +

```
1 USE gpm
2 SELECT * FROM customertable|
```

1 Result Profiler Messages Table Data Info

(Read Only) | (Read Only)

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)

select * from customertable LIMIT 0, 1000

Total: 0.002 sec 5 row(s) Ln 2, Col 28 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Build complex queries using drag-n-drop interface : Reason #13 to upgrade

Query 1 History +

```
1 USE gpm
2 SELECT * FROM customertable
3 INSERT INTO customertable VALUES (6, "Ram", 4)|
```

1 Messages Table Data Info

1 queries executed, 0 success, 1 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)

Error Code: 1142
INSERT command denied to user 'rupesh2'@'localhost' for table 'customertable'

Execution Time : 0 sec
Transfer Time : 0 sec
Total Time : 0.044 sec

All

: 0 sec Total: 0.044 sec Ln 3, Col 47 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Data Sync tool helps restore missing or damaged data at table, row or column level : Reason #34 to upgrade

```
Query 1 History +  
1 CREATE USER "rupesh2"@"127.0.0.1" IDENTIFIED BY "rupesh"  
2 GRANT SELECT ON gpm.customertable TO "rupesh2"@"127.0.0.1"  
3 SHOW GRANTS FOR "rupesh2"@"127.0.0.1"  
4 GRANT INSERT ON gpm.customertable TO "rupesh2"@"127.0.0.1"  
5  
6
```

1 Messages 2 Table Data 3 Info

1 queries executed, 1 success, 0 errors, 0 warnings

Query: grant insert on gpm.customertable to "rupesh2"@"127.0.0.1"

0 row(s) affected

Execution Time : 0.124 sec
Transfer Time : 0 sec
Total Time : 0.124 sec

All

Total: 0.124 sec

Ln 4, Col 59

Connections: 2

[Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

Print your database schema using Visual Schema Designer : Reason #9 to upgrade

```
Query 1 History +  
1 USE gpm  
2 SELECT * FROM customertable  
3 INSERT INTO customertable VALUES (6, "Ram", 4)|
```

1 Messages 2 Table Data 3 Info

1 queries executed, 1 success, 0 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)

1 row(s) affected

Execution Time : 0.208 sec
Transfer Time : 0 sec
Total Time : 0.209 sec

All

Total: 0.209 sec

Ln 3, Col 47

Connections: 2

[Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

Print your database schema using Visual Schema Designer : Reason #9 to upgrade

Query 1 History +

```

1 USE gpm
2 SELECT * FROM customertable
3 INSERT INTO customertable VALUES (6, "Ram", 4)

```

1 Messages **2 Table Data** **3 Info**

1 queries executed, 1 success, 0 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)

1 row(s) affected

Execution Time : 0.208 sec
Transfer Time : 0 sec
Total Time : 0.209 sec

All

Total: 0.209 sec Ln 3, Col 47 Connections: 2 [Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

Import external data at scheduled intervals : Reason #40 to upgrade

Query 1 History +

```

1 USE gpm
2 SELECT * FROM customertable
3 INSERT INTO customertable VALUES (6, "Ram", 4)

```

1 Result **2 Profiler** **3 Messages** **4 Table Data** **5 Info**

(Read Only) | Limit rows First row 0 # of rows 1000

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)
6	Ram	4

select * from customertable LIMIT 0, 1000

Total: 0.003 sec 6 row(s) Ln 3, Col 47 Connections: 2 [Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

2. REVOKE

Compare the contents of two databases using Data Sync Tool : Reason #27 to upgrade

```
Query 1 History +
1 CREATE USER "rupesh2"@"127.0.0.1" IDENTIFIED BY "rupesh"
2 GRANT SELECT ON gpm.customertable TO "rupesh2"@"127.0.0.1"
3 SHOW GRANTS FOR "rupesh2"@"127.0.0.1"
4 GRANT INSERT ON gpm.customertable TO "rupesh2"@"127.0.0.1"
5 REVOKE INSERT ON gpm.customertable FROM "rupesh2"@"127.0.0.1"
6
7
```

1 Messages 2 Table Data 3 Info
1 queries executed, 1 success, 0 errors, 0 warnings

Query: revoke insert on gpm.customertable FROm "rupesh2"@"127.0.0.1"
0 row(s) affected

Execution Time : 0.120 sec
Transfer Time : 0 sec
Total Time : 0.121 sec

All Total: 0.121 sec Ln 5, Col 62 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Create schema visually using Visual Schema Designer : Reason #5 to upgrade

```
Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 INSERT INTO customertable VALUES (6, "Ram", 4)
```

1 Messages 2 Table Data 3 Info
1 queries executed, 0 success, 1 errors, 0 warnings

Query: insert into customertable values (6, "Ram", 4)
Error Code: 1142
INSERT command denied to user 'rupesh2'@'localhost' for table 'customertable'

Execution Time : 0 sec
Transfer Time : 0 sec
Total Time : 0 sec

All Total: 0 sec Ln 3, Col 47 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

2. TCL

1) COMMIT

Schema Optimizer helps fit more data into memory thereby improving response times : Reason #66 to upgrade

```
Query 1 History +  
1 USE gpm  
2 SELECT * FROM customertable  
3 INSERT INTO customertable VALUES (6, "Ram", 4)  
4 COMMIT
```

1 Messages 2 Table Data 3 Info
1 queries executed, 1 success, 0 errors, 0 warnings
Query: commit
0 row(s) affected
Execution Time : 0 sec
Transfer Time : 0 sec
Total Time : 0.001 sec

All Total: 0.001 sec Ln 4, Col 7 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

2) SAVEPOINT

Schema Sync tool brings two or more schemas in complete sync : Reason #26 to upgrade

```
Query 1 History +  
1 USE gpm  
2 SELECT * FROM customertable  
3 START TRANSACTION  
4 SAVEPOINT A  
5 INSERT INTO customertable VALUES (6, "Ram", 4)  
6 SAVEPOINT B  
7 INSERT INTO customertable VALUES (7, "Sham", 3)  
8 SAVEPOINT C  
9 ROLLBACK TO A  
10 COMMIT  
11  
12
```

1 Messages 2 Table Data 3 Info
1 queries executed, 1 success, 0 errors, 0 warnings
Query: Start transaction
0 row(s) affected
Execution Time : 0.114 sec
Transfer Time : 0 sec
Total Time : 0.114 sec

All Total: 0.114 sec Ln 3, Col 18 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Data Sync can be scheduled : Reason #29 to upgrade

```
Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12
```

1 Messages **2 Table Data** **3 Info**

1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint A

0 row(s) affected

Execution Time : 0 sec
Transfer Time : 0 sec
Total Time : 0 sec

All

Total: 0 sec Ln 4, Col 12 Connections: 2 [Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

Data Sync can be scheduled : Reason #29 to upgrade

```
Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12
```

1 Result **2 Profiler** **3 Messages** **4 Table Data** **5 Info**

(Read Only)

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)
6	Ram	4

Limit rows First row 0 # of rows 1000

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.070 sec 6 row(s) Ln 5, Col 47 Connections: 2 [Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

Data Sync can be scheduled : Reason #29 to upgrade

```
Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12
```

1 Messages 2 Table Data 3 Info
1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint B

0 row(s) affected

Execution Time : 0 sec
Transfer Time : 0.004 sec
Total Time : 0.004 sec

All Total: 0.004 sec Ln 6, Col 12 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

3) ROLLBACK

Schema Sync tool brings two or more schemas in complete sync : Reason #26 to upgrade

```
Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12
```

1 Result 2 Profiler 3 Messages 4 Table Data 5 Info

customerID	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.002 sec 5 row(s) Ln 2, Col 28 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Schema Sync tool brings two or more schemas in complete sync : Reason #26 to upgrade

```
Query 1 History +  
1 USE gpm  
2 SELECT * FROM customertable  
3 START TRANSACTION  
4 SAVEPOINT A  
5 INSERT INTO customertable VALUES (6, "Ram", 4)  
6 SAVEPOINT B  
7 INSERT INTO customertable VALUES (7, "Sham", 3)  
8 SAVEPOINT C  
9 ROLLBACK TO A  
10 COMMIT  
11  
12
```

1 Messages 2 Table Data 3 Info
1 queries executed, 1 success, 0 errors, 0 warnings
Query: Start transaction
0 row(s) affected
Execution Time : 0.114 sec
Transfer Time : 0 sec
Total Time : 0.114 sec

All Total: 0.114 sec Ln 3, Col 18 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Data Sync can be scheduled : Reason #29 to upgrade

```
Query 1 History +  
1 USE gpm  
2 SELECT * FROM customertable  
3 START TRANSACTION  
4 SAVEPOINT A  
5 INSERT INTO customertable VALUES (6, "Ram", 4)  
6 SAVEPOINT B  
7 INSERT INTO customertable VALUES (7, "Sham", 3)  
8 SAVEPOINT C  
9 ROLLBACK TO A  
10 COMMIT  
11  
12
```

1 Messages 2 Table Data 3 Info
1 queries executed, 1 success, 0 errors, 0 warnings
Query: savepoint A
0 row(s) affected
Execution Time : 0 sec
Transfer Time : 0 sec
Total Time : 0 sec

All Total: 0 sec Ln 4, Col 12 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Data Sync can be scheduled : Reason #29 to upgrade

```

Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12

```

1 Result **2 Profiler** **3 Messages** **4 Table Data** **5 Info**

customerID	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)
6	Ram	4

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.070 sec 6 row(s) Ln 5, Col 47 Connections: 2 [Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

Data Sync can be scheduled : Reason #29 to upgrade

```

Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12

```

1 Messages **2 Table Data** **3 Info**

1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint B

0 row(s) affected

Execution Time : 0 sec
Transfer Time : 0.004 sec
Total Time : 0.004 sec

All

Total: 0.004 sec Ln 6, Col 12 Connections: 2 [Upgrade to SQLyog Professional/Enterprise/Ultimate](#)

Generate schema change histories, ideal for database versioning : Reason #24 to upgrade

```

Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12

```

1 Result **2 Profiler** **3 Messages** **4 Table Data** **5 Info**

(Read Only) | Limit rows First row 0 # of rows 1000

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)
6	Ram	4
7	Sham	3

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.002 sec 7 row(s) Ln 7, Col 48 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Generate schema change histories, ideal for database versioning : Reason #24 to upgrade

```

Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12

```

1 Messages **2 Table Data** **3 Info**

1 queries executed, 1 success, 0 errors, 0 warnings

Query: savepoint C

0 row(s) affected

Execution Time : 0 sec
Transfer Time : 0 sec
Total Time : 0.001 sec

All

Total: 0.001 sec Ln 8, Col 12 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Generate schema change histories, ideal for database versioning : Reason #24 to upgrade

```
Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO B
10 COMMIT
11
12
```

1 Messages 2 Table Data 3 Info

1 queries executed, 1 success, 0 errors, 0 warnings

Query: ROLLBACK to B

0 row(s) affected

Execution Time : 0.043 sec
Transfer Time : 0 sec
Total Time : 0.044 sec

All

Total: 0.044 sec Ln 9, Col 14 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Generate schema change histories, ideal for database versioning : Reason #24 to upgrade

```
Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO B
10 COMMIT
11
12
```

1 Result 2 Profiler 3 Messages 4 Table Data 5 Info

(Read Only)

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)
6	Ram	4

Limit rows First row 0 # of rows 1000

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.003 sec 6 row(s) Ln 2, Col 28 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Generate schema change histories, ideal for database versioning : Reason #24 to upgrade

```

Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12

```

1 Messages **2 Table Data** **3 Info**

1 queries executed, 1 success, 0 errors, 0 warnings

Query: ROLLBACK to A

0 row(s) affected

Execution Time : 0 sec
Transfer Time : 0 sec
Total Time : 0 sec

All

Total: 0 sec Ln 9, Col 14 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Generate schema change histories, ideal for database versioning : Reason #24 to upgrade

```

Query 1 History +
1 USE gpm
2 SELECT * FROM customertable
3 START TRANSACTION
4 SAVEPOINT A
5 INSERT INTO customertable VALUES (6, "Ram", 4)
6 SAVEPOINT B
7 INSERT INTO customertable VALUES (7, "Sham", 3)
8 SAVEPOINT C
9 ROLLBACK TO A
10 COMMIT
11
12

```

1 Result **2 Profiler** **3 Messages** **4 Table Data** **5 Info**

(Read Only) | (Read Only) | (Read Only) | (Read Only) | (Read Only)

customerId	customerName	productId
1	Rupesh	2
2	Rohan	4
3	Omkar	3
4	Anniruddha	1
5	Rahul	(NULL)

SELECT * FROM customertable LIMIT 0, 1000

Total: 0.038 sec 5 row(s) Ln 2, Col 28 Connections: 2 Upgrade to SQLyog Professional/Enterprise/Ultimate

Conclusion: Thus, we understood and implemented DCL AND TCL commands to manipulate users and permissions.

Practical no. 9

Title: Write a PL/SQL programs using if then else, for, while, nested loop

Theory:

1. IF-THEN Statement

It is the simplest form of the IF control statement, frequently used in decision-making and changing the control flow of the program execution. The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF. If the condition is TRUE, the statements get executed, and if the condition is FALSE or NULL, then the IF statement does nothing.

Syntax :

```
IF condition THEN  
    S;  
END IF;
```

2. IF-THEN-ELSE Statement

A sequence of IF-THEN statements can be followed by an optional sequence of ELSE statements, which execute when the condition is FALSE.

Syntax :

```
IF condition THEN  
    S1;  
ELSE  
    S2;  
END IF;
```

3. Basic Loop Statement

Basic loop structure encloses sequence of statements in between the LOOP and END LOOP statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax :

```
LOOP  
    Sequence of statements;  
END LOOP;
```

4. FOR LOOP Statement

A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax :

```
FOR counter IN initial_value .. final_value LOOP  
    sequence_of_statements;  
END LOOP;
```

5. WHILE LOOP Statement

A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
WHILE condition LOOP  
    sequence_of_statements  
END LOOP;
```

6. Nested Loops

PL/SQL allows using one loop inside another loop. Following section shows a few examples to illustrate the concept.

Syntax :

```
LOOP  
    Sequence of statements1  
    LOOP  
        Sequence of statements2  
    END LOOP;  
END LOOP;
```

Query and output:

```
DECLARE
    a number(2) := 93;
    c number(2) := 4;
    i number(3) := 2;
    j number(3);
BEGIN
    -- If condition demo
    dbms_output.put_line('If condition demo:');
    IF(a >= 30) THEN
        dbms_output.put_line('A is greater than 30');
    ELSE
        dbms_output.put_line('A is smaller than 30');
    END IF;

    -- For loop demo
    dbms_output.put_line('');
    dbms_output.put_line('For loop demo: ');
    FOR i in 1..5 LOOP
        dbms_output.put_line('i: '||i);
    END LOOP;

    -- While loop demo
    dbms_output.put_line('');
    dbms_output.put_line('While loop demo: ');
    WHILE c>=0 LOOP
        dbms_output.put_line('c: '||c);
        c := c-1;
    END LOOP;

    -- Nested loop demo with example of prime nos
    dbms_output.put_line('');
    dbms_output.put_line('Nested loop demo, Prime nos from 2 to 20: ');
    LOOP
        j := 2;
        LOOP
            exit WHEN ((mod(i,j)=0) or (j=i));
            j := j+1;
        END LOOP;
        IF (j=i) THEN
            dbms_output.put_line(i || ' is prime no.');
        ELSE
            dbms_output.put_line(i || ' is not prime');
        END IF;
        i := i+1;
        exit WHEN i=20;
    END LOOP;
END;
```

Results	Explain	Describe	Saved SQL	History
If condition demo: A is greater than 30 For loop demo: i: 1 i: 2 i: 3 i: 4 i: 5 While loop demo: c: 4 c: 3 c: 2 c: 1 c: 0 Nested loop demo, Prime nos from 2 to 20: 2 is prime no. 3 is prime no. 4 is not prime 5 is prime no. 6 is not prime 7 is prime no. 8 is not prime 9 is not prime 10 is not prime 11 is prime no. 12 is not prime 13 is prime no. 14 is not prime 15 is not prime 16 is not prime 17 is prime no. 18 is not prime 19 is prime no. Statement processed. 0.01 seconds				

Conclusion: Thus, we implemented PL/SQL programs using if then else, for, while, nested loop

Practical no. 10

FS19CO042

Title: Write a PL/SQL code to implement implicit and explicit cursors.

Theory:

1. Cursor :

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.

2. Implicit Cursor :

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, you can refer to the most recent implicit cursor as the SQL cursor, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement.

3. Explicit Cursor

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.

Syntax :

```
CURSOR cursor_name IS select_statement;
```

Output :

1. Implicit Cursor :

CUSTOMERS table :

Results Explain Describe Saved SQL History					
ID	NAME	AGE	ADDRESS	SALARY	
1	Aniruddha	32	Ahmedabad	2000	
2	Rupesh	25	Delhi	1500	
3	Omkar	23	Kota	2000	
4	Devang	25	Mumbai	6500	
5	Arpan	27	Bhopal	8500	
6	Rishabh	22	MP	4500	

The following program will update the table and increase the salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  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 || ' customers selected ');
  END IF;
END;
```

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

Results Explain Describe Saved SQL History				
6 customers selected				

If you check the records in customers table, you will find that the rows have been updated

Customers Table :

Results Explain Describe Saved SQL History					
ID	NAME	AGE	ADDRESS	SALARY	
1	Aniruddha	32	Ahmedabad	2500	
2	Rupesh	25	Delhi	2000	
3	Omkar	23	Kota	2500	
4	Devang	25	Mumbai	7000	
5	Arpan	27	Bhopal	9000	
6	Rishabh	22	MP	5000	

2. Explicit Cursor

Following is a complete example to illustrate the concepts of explicit cursors

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers IS
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers INTO c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
```

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

Results Explain Describe Saved SQL History					
4 Devang Mumbai 1 Aniruddha Ahmedabad 2 Rupesh Delhi 5 Arpan Bhopal 6 Rishabh MP 3 Omkar Kota Statement processed.					

Conclusion: Thus we understood and implemented implicit and explicit cursors.

Practical no. 11

Title: Write a PL/SQL code to create procedure and function.

Theory:

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. We will discuss packages in the chapter 'PL/SQL - Packages'.

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

1. Functions :

These subprograms return a single value; mainly used to compute and return a value.

2. Procedures :

These subprograms do not return a value directly; mainly used to perform an action.

1. Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The procedure contains a header and a body.

Header :

The header contains the name of the procedure and the parameters or variables passed to the procedure.

Body :

The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

Three ways to pass parameters in procedure:

1. IN parameters: The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. OUT parameters: The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. INOUT parameters: The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

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;

```

2. Function

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too. A standalone function is created using the CREATE FUNCTION statement.

The function must contain a return statement.

RETURN clause specifies that data type you 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

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];
```

Output:

1. Procedure :

The screenshot shows the Oracle SQL Developer interface. In the top bar, 'Language' is set to 'PL/SQL'. The main area displays the following PL/SQL code:

```
1  DECLARE  
2      num1 number ;  
3      num2 number ;  
4      maxNumber number ;  
5  PROCEDURE findmax(x IN number, y IN number, z OUT number) IS  
6  BEGIN  
7      IF x > y THEN  
8          z :=x ;  
9      ELSE  
10         z :=y ;  
11     END IF;  
12  END;  
13  BEGIN  
14      num1:=:num1 ;  
15      num2:=:num2 ;  
16      findmax(num1, num2, maxNumber);  
17      dbms_output.put_line(maxNumber);  
18  END;
```

The code is highlighted with syntax coloring. Below the code, the 'Results' tab is selected, showing the output: 'Statement processed.'

The screenshot shows the Oracle SQL Developer interface. In the top-left, there's a code editor with the following PL/SQL procedure:

```

1 DECLARE
2     num1 number ;
3     num2 number ;
4     maxNumber number ;
5 PROCEDURE findmax(x IN number, y IN number, z OUT number) IS
6 BEGIN
7     IF x > y THEN
8         z :=x ;
9     ELSE
10        z :=y ;
11    END IF;
12 END;
13 BEGIN
14     num1:=:num1 ;
15     num2:=:num2 ;
16     findmax(num1, num2, maxNumber);
17     dbms_output.put_line(maxNumber);
18 END;

```

In the top-right, there's a small browser window titled "Enter Bind Variables - Google Chrome" with the URL "apex.oracle.com/pls/apex/f?p=4500:138:16374790941979::". It contains a table with two rows:

Bind Variable	Value
:NUM1	10
:NUM2	20

At the bottom, the "Results" tab is selected.

The screenshot shows the Oracle SQL Developer interface with the same PL/SQL procedure in the code editor. The "Results" tab is selected at the bottom, displaying the output of the procedure:

```

20
Statement processed.

0.01 seconds

```

2. Function :

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```
1 DECLARE
2     num1 number ;
3     num2 number ;
4     maxNumber number ;
5 FUNCTION findmax(x IN number, y IN number)RETURN number IS z number ;
6 BEGIN
7     IF x > y THEN
8         z :=x ;
9     ELSE
10        z :=y ;
11    END IF;
12    RETURN z;
13 END;
14 BEGIN
15     num1:=:num1 ;
16     num2:=:num2 ;
17     maxNumber:= findmax(num1, num2);
18     dbms_output.put_line(maxNumber);
19 END;
```

Results Explain Describe Saved SQL History

20 Statement processed.

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```
1 DECLARE
2     num1 number ;
3     num2 number ;
4     maxNumber number ;
5 FUNCTION findmax(x IN number, y IN number)RETURN number IS z number ;
6 BEGIN
7     IF x > y THEN
8         z :=x ;
9     ELSE
10        z :=y ;
11    END IF;
12    RETURN z;
13 END;
14 BEGIN
15     num1:=:num1 ;
16     num2:=:num2 ;
17     maxNumber:= findmax(num1, num2);
18     dbms_output.put_line(maxNumber);
19 END;
```

Bind Variable Value

Bind Variable	Value
:NUM1	10
:NUM2	20

Enter Bind Variables - Google Chrome apex.oracle.com/pls/apex/f?p=4500:138:16374790941979:: Submit

Results Explain Describe Saved SQL History

The screenshot shows a PL/SQL development environment. At the top, there's a toolbar with 'Language' set to 'PL/SQL', 'Rows' set to '10', and buttons for 'Clear Command' and 'Find Tables'. On the right, there are 'Save' and 'Run' buttons. Below the toolbar is a menu bar with icons for Undo, Redo, Find, and Replace. The main area contains a code editor with the following PL/SQL code:

```
1 DECLARE
2     num1 number ;
3     num2 number ;
4     maxNumber number ;
5     FUNCTION findmax(x IN number, y IN number)RETURN number IS z number ;
6     BEGIN
7         IF x > y THEN
8             z :=x ;
9         ELSE
10            z :=y ;
11        END IF;
12        RETURN z;
13    END;
14    BEGIN
15        num1:=:num1 ;
```

Below the code editor, a navigation bar includes 'Results' (which is selected), 'Explain', 'Describe', 'Saved SQL', and 'History'. The results pane displays the output of the executed statement:

```
20
Statement processed.

0.00 seconds
```

Conclusion: Thus, we defined and used procedures and functions in PL/SQL.

Title: Write a PL/SQL code to create triggers on given database

Theory:

Triggers :

Trigger is invoked by Oracle engine automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition matches. Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers can 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 –

1. Generating some derived column values automatically
2. Enforcing referential integrity
3. Event logging and storing information on table access
4. Auditing
5. Synchronous replication of tables
6. Imposing security authorizations
7. Preventing invalid transactions

Syntax :

```
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;
```

Output :

```
Language PL/SQL Rows 10 Clear Command Find Tables Save Run
C | Q | A: | ⚙️
1 create table account(
2     accNumber number,
3     name varchar(20),
4     balance number
5 );
6
7 create table transactions(
8     accNumber number,
9     name varchar(20),
10    amount number,
11    balance number,
12    transactionDate date
13 );
14
15 create sequence acc_seq;
16
17 create trigger triggerTransaction after update on account for each row
18 begin
19     insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);

```

Results Explain Describe Saved SQL History

Find Go

```
Language PL/SQL Rows 10 Clear Command Find Tables Save Run
C | Q | A: | ⚙️
14
15 create sequence acc_seq;
16
17 create trigger triggerTransaction after update on account for each row
18 begin
19     insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
20 end;
21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update table set balance = balance + 4000 where accNumber = 1;
27 update table set balance = balance + 9000 where accNumber = 2;
28 update table set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;
```

Results Explain Describe Saved SQL History

Find Go

Language PL/SQL ▾ Rows 10 Clear Command Find Tables Save Run

```
1 create table account(
2     accNumber number,
3     name varchar(20),
4     balance number
5 );
6
7 create table transactions(
8     accNumber number,
9     name varchar(20),
10    amount number,
11    balance number,
12    transactionDate date
13 );
14
15 create sequence acc_seq;
```

Results Explain Describe Saved SQL History

Table created.

0.04 seconds

Language PL/SQL ▾ Rows 10 Clear Command Find Tables Save Run

```
1 create table account(
2     accNumber number,
3     name varchar(20),
4     balance number
5 );
6
7 create table transactions(
8     accNumber number,
9     name varchar(20),
10    amount number,
11    balance number,
12    transactionDate date
13 );
14
15 create sequence acc_seq;
```

Results Explain Describe Saved SQL History

Table created.

0.03 seconds

Language PL/SQL  Rows 10  Clear Command Find Tables Save Run

```
9 |     name varchar(20),
10|     amount number,
11|     balance number,
12|     transactionDate date
13| );
14|
15| create sequence acc_seq;
16|
17| create trigger triggerTransaction after update on account for each row
18| begin
19|     insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
20| end;
21|
22| insert into account values(1, 'Rupesh Raut', 0);
23| insert into account values(2, 'Aniruddha Shriwant', 0);
24| insert into account values(3, 'Omkar Phansopkar', 0);
```

Results Explain Describe Saved SQL History

Sequence created.

0.04 seconds

```
12|     transactionDate date
13| );
14|
15| create sequence acc_seq;
16|
17| create trigger triggerTransaction after update on account for each row
18| begin
19|     insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
20| end;
21|
22| insert into account values(1, 'Rupesh Raut', 0);
23| insert into account values(2, 'Aniruddha Shriwant', 0);
24| insert into account values(3, 'Omkar Phansopkar', 0);
```

Results Explain Describe Saved SQL History

Trigger created.

0.10 seconds

Language PL/SQL  Rows 10  Clear Command Find Tables Save Run

```
15| create sequence acc_seq;
16|
17| create trigger triggerTransaction after update on account for each row
18| begin
19|     insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
20| end;
21|
22| insert into account values(1, 'Rupesh Raut', 0);
23| insert into account values(2, 'Aniruddha Shriwant', 0);
24| insert into account values(3, 'Omkar Phansopkar', 0);
25|
26| update table set balance = balance + 4000 where accNumber = 1;
27| update table set balance = balance + 9000 where accNumber = 2;
28| update table set balance = balance + 10000 where accNumber = 3;
29|
30| select * from account order by accNumber asc;
31| select * from transactions order by transactionDate asc;
```

Results Explain Describe Saved SQL History

1 row(s) inserted.

0.03 seconds

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update table set balance = balance + 4000 where accNumber = 1;
27 update table set balance = balance + 9000 where accNumber = 2;
28 update table set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

ACCTNUMBER	NAME	BALANCE
1	Rupesh Raut	0
2	Aniruddha Shriwant	0
3	Omkar Phansopkar	0

3 rows returned in 0.02 seconds Download

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update table set balance = balance + 4000 where accNumber = 1;
27 update table set balance = balance + 9000 where accNumber = 2;
28 update table set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

no data found

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

18 begin
19 | insert into transactions values(:new.accNumber, :old.name, abs(:new.balance - :old.balance), :new.balance, sysdate);
20 end;
21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update account set balance = balance + 4000 where accNumber = 1;
27 update account set balance = balance + 9000 where accNumber = 2;
28 update account set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

1 row(s) updated.

0.04 seconds

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

20 end;
21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update account set balance = balance + 4000 where accNumber = 1;
27 update account set balance = balance + 9000 where accNumber = 2;
28 update account set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

ACCTNUMBER	NAME	AMOUNT	BALANCE	TRANSACTIONDATE
1	Rupesh Raut	4000	4000	01/30/2021

1 rows returned in 0.01 seconds [Download](#)

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

20 end;
21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update account set balance = balance + 4000 where accNumber = 1;
27 update account set balance = balance + 9000 where accNumber = 2;
28 update account set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

1 row(s) updated.

0.01 seconds

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

20 end;
21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update account set balance = balance + 4000 where accNumber = 1;
27 update account set balance = balance + 9000 where accNumber = 2;
28 update account set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

1 row(s) updated.

0.02 seconds

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update account set balance = balance + 4000 where accNumber = 1;
27 update account set balance = balance + 9000 where accNumber = 2;
28 update account set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

ACCTNUMBER	NAME	BALANCE
1	Rupesh Raut	4000
2	Aniruddha Shriwant	9000
3	Omkar Phansopkar	10000

3 rows returned in 0.01 seconds Download

Language PL/SQL Rows 10 Clear Command Find Tables Save Run

```

21
22 insert into account values(1, 'Rupesh Raut', 0);
23 insert into account values(2, 'Aniruddha Shriwant', 0);
24 insert into account values(3, 'Omkar Phansopkar', 0);
25
26 update account set balance = balance + 4000 where accNumber = 1;
27 update account set balance = balance + 9000 where accNumber = 2;
28 update account set balance = balance + 10000 where accNumber = 3;
29
30 select * from account order by accNumber asc;
31 select * from transactions order by transactionDate asc;

```

Results Explain Describe Saved SQL History

ACCTNUMBER	NAME	AMOUNT	BALANCE	TRANSACTIONDATE
1	Rupesh Raut	4000	4000	01/30/2021
2	Aniruddha Shriwant	9000	9000	01/30/2021
3	Omkar Phansopkar	10000	10000	01/30/2021

3 rows returned in 0.00 seconds Download

Conclusion: Thus, we created triggers on database.

Experiment 13

Title: Case Study on ER Model and EER Model.

Theory:

- Purpose of ER diagram

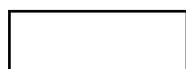
ENTITY RELATIONAL (ER) MODEL is a high-level conceptual data model diagram for database. ER modeling helps to analyze data requirements systematically to produce a well-designed database.

The Entity-Relation model represents real-world entities and the relationship between them. It is considered best practice to complete ER modeling before implementing your database.

ER model can be converted to any other data model like relational or network model for actual database implementation.

- Symbols used to represent components of ER diagram.

Entity



Weak Entity



Simple

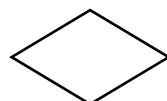


Attribute

Multivalued



Relationship



Weak

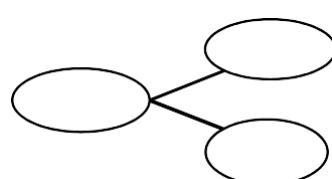


Relationship

Connecting line



Composite



attribute

Derived
Attribute



Key/single
valued attribute

Attribute

- Purpose of EER diagram.

Enhanced entity-relationship (EER) diagrams are basically an expanded upon version of ER diagrams. EER models are helpful tools for designing databases with high-level models.

With their enhanced features, you can plan databases more thoroughly by delving into the properties and constraints with more precision.

EER mainly used to display following relationship concepts:

1. Super and sub class
2. Specialization and Generalization
3. Aggregation

- Generalization and Specialization concepts with example

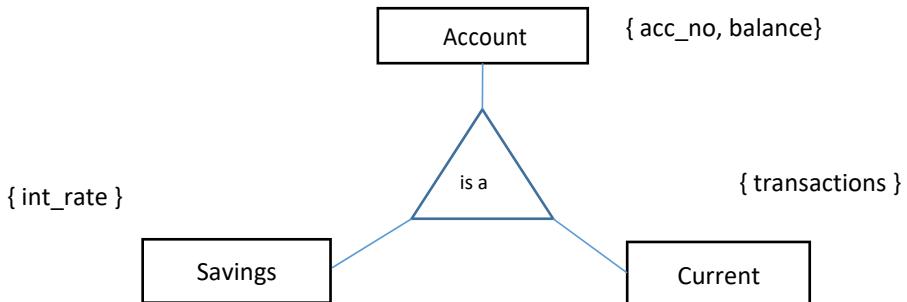
Generalization: In generalization, two or more lower level entities are combined together to form one higher level entity. I.e. Subclasses are combine together to form a super class
It is a bottom to up approach.

One higher level entity can also be combined with other lower level entities to form another higher level entity.

Common properties among lower level entities to form higher level entity.

For example,

Savings and current accounts are both accounts with added special properties.

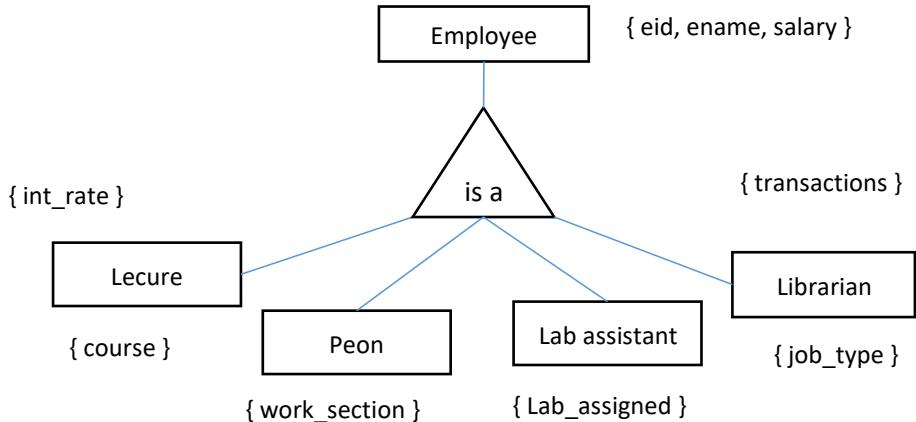


Specialization: In specialization, higher level entity is divided into two or more lower level entities. I.e. Super class is divided to form two or more subclasses.

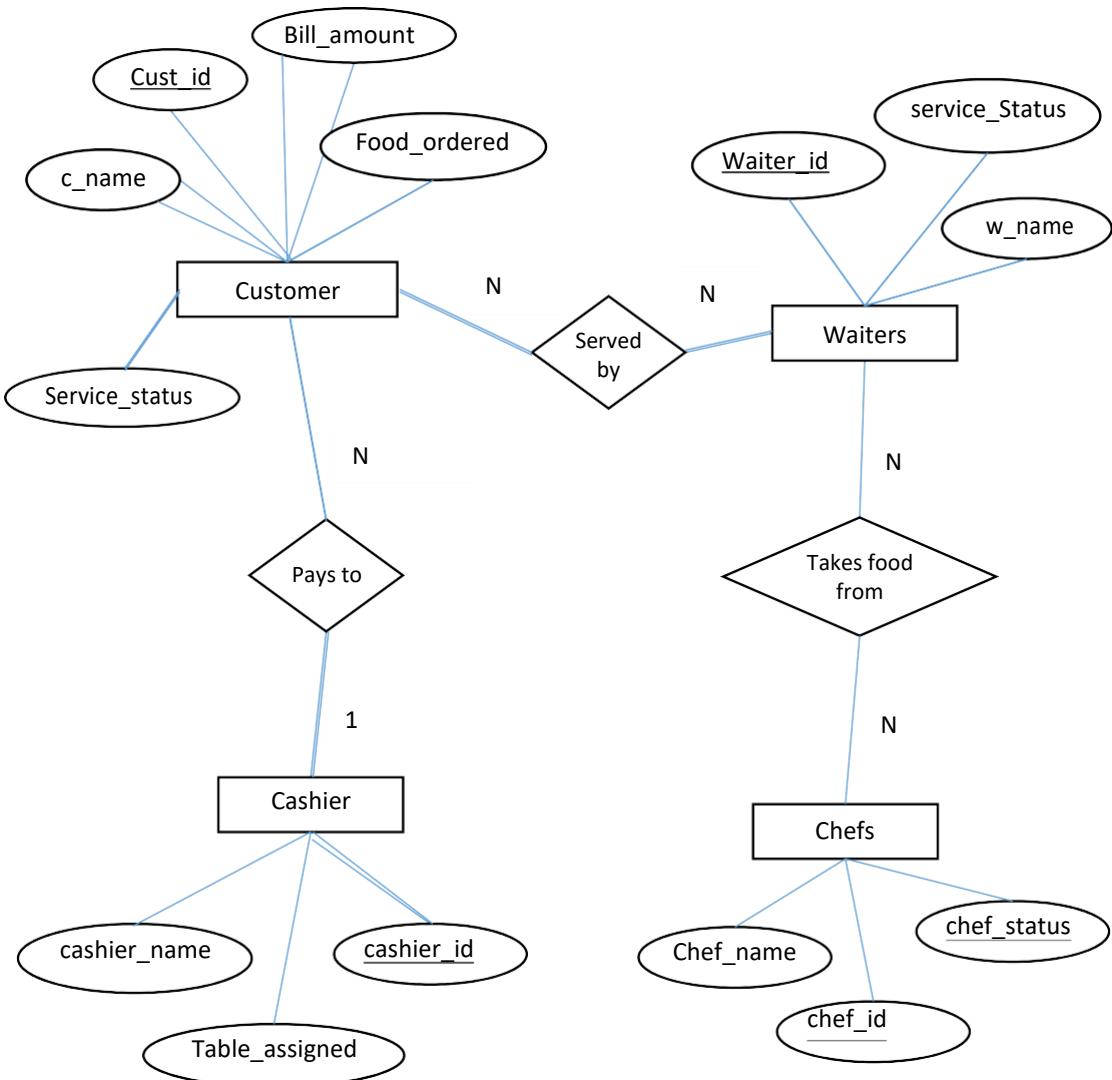
It is a top-down approach.

This division is done on the basis of specialized properties in entity.

For example,



ER diagram of restaurant management system:



Conclusion: In this exp, we studied ER and EER models and made few exemplary diagrams.