GURU NANAK INSTITUTE OF TECHNOLOGY (AUTONOMOUS)

DepartmentofElectronicsandCommunicationEngineering

Course : Database Management Systems

Course Code: 22PC0CS09

AcademicYear-2025-2026

Program	**B.Tech**
Branch	**CSE(AI&ML)**
Year&Semester	**IIYear&IISemester**
Section	**A &B**
PreparedBy	K.KRISHNACHAITHANYA

GURUNANAKINSTITUTEOFTECHNOLOGY
(AUTONOMUS)
Ibrahimpatnam,RRDist-501506 www.gnithyd.ac.in

Course Code	22PC0AD11			
Course Title	Database Management Systems Lab			
Branch	B. Tech. Artificial Intelligence and Data Science			
Year and Semester	II Year II Semester			
Category	Professional Subjects – Core			
Scheme and Credits	L	T	P	Credits
	3	0	0	3
Prerequisites	22ES0AD01 – Data Structures			

Course Objectives:

1. To learn the ER data model, database design and normalization
2. To Learn SQL basics for data definition and data manipulation

Course Outcomes: After the completion of this course, the students would be able to

1. Demonstrate ER Modeling concepts to design the Database
2. Apply integrity constraints on a database
3. Make use of DDL, DML, DCL, TCL commands in creation and manipulation of Database
4. Implementation of database queries using PL/SQL
5. Experiment with triggers to maintain the referential integrity of data

LIST OF EXPERIMENTS:

1. Design ER Model for a given application & Convert ER model to Relational Model.
2. Creating users - roles and Granting privileges.
3. Creating and altering tables for various relations in SQL using Integrity Constraints.
4. Implementing queries in SQL using

4.1 Insertion

4.2 Retrieval (operations like union - intersect - minus - in - exists - group by and having)

4.3 Updation

4.4 Deletion

5. Implementing the concepts of Rollback – commit, checkpoints and Views
6. Implementing joins - sub queries - nested and co related nested queries.
7. Experiment with built in functions in oracle (Numeric, String, Date, Aggregate function set.)

8. Implementing operations on relations using PL/SQL.
9. Implementing functions, stored procedures using PL/SQL
10. Implementing cursors using PL/SQL
11. Implement Exception Handling using PL/SQL
12. Creating triggers using PL/SQ

Case Study – Give options to the students to do case study on their own / faculty may give list of case study in beginning of the semester

1. Inventory control management System
2. College Management System
3. Hospital management System
4. Library management System
5. Payroll management System
6. Health care organization Management System
7. Restaurant Management System
8. Blood Donation Management System
9. Art Gallery Management System
10. Hotel Management System
11. School Management System
12. Salary Management System
13. Wholesale Management System
14. Timetable Management System
15. Website Management

Experiment1: Designing ER Model and Conversion to Relational Model for Library Management System

Tools Used

ER diagramming tool (such as Lucid chart, Draw.io, etc.)

Database management system (e.g., MySQL) for implementing the relational model.

Objective:

To design an Entity-Relationship (ER) model and convert it into a Relational Model for a Bus Management System.

Requirements:

Understanding of Entity-Relationship (ER) modeling concepts.

Knowledge of relational database design principles.

Familiarity with notation standards for ER diagrams.

Experiment:

1. Entity-Relationship (ER) Model:

Entities:

Bus: Represents a bus in the system.

Attributes: BusID (Primary Key), Route, Capacity, Type.

Driver: Represents a driver associated with a bus.

Attributes: DriverID (Primary Key), Name, LicenseNumber.

Passenger: Represents a passenger using the bus service.

Attributes: PassengerID (Primary Key), Name, Age, ContactInfo.

Route: Represents a route followed by buses.

Attributes: RouteID (Primary Key), Source, Destination, Distance.

Ticket: Represents a ticket issued to a passenger for a bus journey.

Attributes: TicketID (Primary Key), PassengerID (Foreign Key), BusID (Foreign Key), RouteID (Foreign Key), Date, Fare.

Relationships:

Bus-Driver: One-to-One relationship between Bus and Driver entities.

Bus-Passenger: One-to-Many relationship between Bus and Passenger entities.

Route-Bus: Many-to-Many relationship between Route and Bus entities.

Ticket-Route: Many-to-One relationship between Ticket and Route entities.

2. Relational Model:

Tables:

Bus table:

Columns: BusID (Primary Key), Route, Capacity, Type, DriverID (Foreign Key).

Driver table:

Columns: DriverID (Primary Key), Name, License Number.

Passenger table:

Columns: PassengerID (Primary Key), Name, Age, ContactInfo.

Route table:

Columns: RouteID (Primary Key), Source, Destination, Distance.

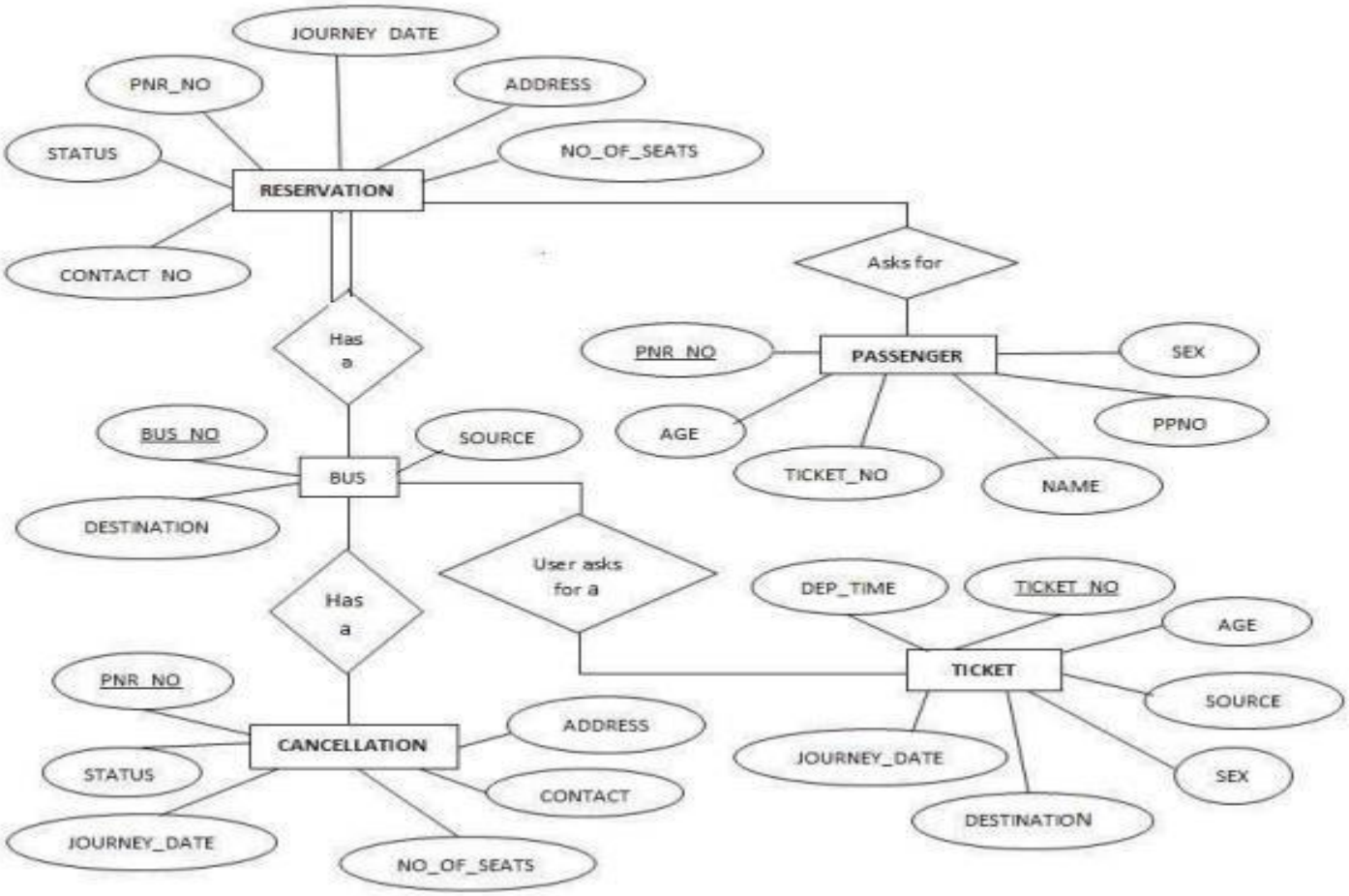
Ticket table:

Columns: TicketID (Primary Key), PassengerID (Foreign Key), BusID (Foreign Key), RouteID (Foreign Key), Date, Fare.

Conclusion:

The Entity-Relationship (ER) model provides a conceptual representation of the Bus Management System, illustrating entities, their attributes, and relationships. The Relational Model converts this conceptual design into a set of normalized tables, facilitating efficient data storage and management within a relational database system.

ER-Diagram: Bus-Management Systems


JOURNEY DATE
PNR_NO
ADDRESS
STATUS
NO_OF_SEATS
RESERVATION
CONTACT NO
Asks for
Has a
PNR_NO
PASSENGER
SEX
BUS_NO
SOURCE
AGE
PPNO
BUS
TICKET_NO
NAME
DESTINATION
User asks for a
Has a
DEP_TIME
TICKET_NO
AGE
PNR_NO
TICKET
SOURCE
ADDRESS
CANCELLATION
JOURNEY_DATE
STATUS
CONTACT
SEX
JOURNEY_DATE
DESTINATION
NO_OF_SEATS

Experiment2. Creating users - roles and Granting privileges

Objective:

To create users, define roles, and grant privileges in a database management system.

Requirements:

Access to a database management system (e.g., MySQL).

Basic understanding of SQL (Structured Query Language).

Familiarity with user management and privilege granting in a database environment.

Experiment:

1. Creating Users:

Open your preferred database management tool (e.g., MySQL Workbench, pgAdmin).

Connect to the database server using appropriate credentials.

Execute the following SQL command to create a new user:

CREATE USER 'username'@'hostname' IDENTIFIED BY 'password';

Replace 'username' with the desired username, 'hostname' with the hostname or IP address from which the user will connect (e.g., 'localhost' for local connections), and 'password' with the desired password.

2. Defining Roles:

Decide on the roles needed in your database environment (e.g., admin, regular user).

Execute the following SQL command to create a new role:

CREATE ROLE rolename;

Replace 'rolename' with the name of the role you want to create.

3. Granting Privileges:

Determine which privileges each role should have (e.g., SELECT, INSERT, UPDATE, DELETE).

Execute the following SQL command to grant privileges to a role or user:

GRANT privilege1, privilege2 ON database.object TO user_or_role;

Replace 'privilege1', 'privilege2' with the specific privileges (e.g., SELECT, INSERT) you want to grant, 'database.object' with the database object (e.g., table, view), and 'user_or_role' with the username or role to which you want to grant privileges.

-- Create UserA and UserB

CREATE USER UserA IDENTIFIED BY password1;

CREATE USER UserB IDENTIFIED BY password2;

-- Grant CREATE SESSION to both users to allow them to log in

GRANT CREATE SESSION TO UserA;

GRANT CREATE SESSION TO UserB;

-- Create a role named Manager

CREATE ROLE Manager;

-- Assuming you have a table in a schema that you wish to grant access to, replace 'your_schema.your_table' with the actual schema and table name

-- Grant SELECT, INSERT, and UPDATE privileges on a specific table to the Manager role

GRANT SELECT, INSERT, UPDATE ON your_schema.your_table TO Manager;

-- Assign the Manager role to UserA

GRANT Manager TO UserA;

-- (Optional) Check the privileges of UserA -- this step requires querying the database's system views or

being logged in as UserA

Example 1:
Suppose we want to create a role called 'staff' with SELECT privilege on a table called 'books' and grant this role to a user named 'john'. We can execute the following SQL commands:

```
CREATE ROLE staff;
GRANT SELECT ON library.books TO staff;
GRANT staff TO 'john'@'localhost';
```

Conclusion:
Creating users, defining roles, and granting privileges are essential tasks in database management. By following these steps, database administrators can control access to database objects and ensure data security and integrity.

EXAMPLE 2:

```
-- Create UserA and UserB
CREATE USER UserA IDENTIFIED BY password1;
CREATE USER UserB IDENTIFIED BY password2;

-- Grant CREATE SESSION privilege to both users
GRANT CREATE SESSION TO UserA;
GRANT CREATE SESSION TO UserB;

-- Create a role named Manager
CREATE ROLE Manager;

-- Grant SELECT, INSERT, UPDATE, and DELETE privileges on the 'employees' table to the Manager role
GRANT SELECT, INSERT, UPDATE, DELETE ON hr.employees TO Manager;

-- Assign the Manager role to UserA
GRANT Manager TO UserA;

-- Create the 'employees' table in the 'hr' schema
CREATE TABLE hr.employees (
employee_id NUMBER PRIMARY KEY,
first_name VARCHAR2(50),
last_name VARCHAR2(50),
email VARCHAR2(100),
hire_date DATE,
job_id VARCHAR2(50),
   salary NUMBER
);

-- Insert sample data into the 'employees' table
```

INSERT INTO hr.employees (employee_id, first_name, last_name, email, hire_date, job_id, salary)
VALUES
(1001, 'John', 'Doe', 'john.doe@example.com', TO_DATE('2022-01-15', 'YYYY-MM-DD'), 'Manager', 50000),
(1002, 'Jane', 'Smith', 'jane.smith@example.com', TO_DATE('2022-02-20', 'YYYY-MM-DD'), 'Engineer', 40000),
(1003, 'Alice', 'Johnson', 'alice.johnson@example.com', TO_DATE('2022-03-25', 'YYYY-MM-DD'), 'Analyst', 35000);

-- Grant SELECT, INSERT, UPDATE, DELETE privileges on the 'employees' table to the 'Manager' role
GRANT SELECT, INSERT, UPDATE, DELETE ON hr.employees TO Manager;

OUTPUT:
SELECT * FROM hr.employees;

Output:

employee_id	first_name	last_name	email	hire_date	job_id	salary
1001	John	Doe	john.doe@example.com	2022-01-15	Manager	50000
1002	Jane	Smith	jane.smith@example.com	2022-02-20	Engineer	40000
1003	Alice	Johnson	alice.johnson@example.com	2022-03-25	Analyst	3500

Experiment3. Creating and altering tables for various relations in SQL using Integrity Constraints

Objective: Learn how to create and alter tables in SQL with integrity constraints to maintain data consistency.

Tasks:

Create a Table for Employees:

Create a table named employees with the following columns:

employee_id (Primary Key)
first_name
last_name
email
hire_date
job_id
salary

Alter Table Structure:

Add a new column named department_id to the employees table to store department information.
Define a foreign key constraint on the department_id column referencing a departments table (if available) or a predefined list of department IDs.

Create a Table for Employees:

```
CREATE TABLE employees (
employee_id INT PRIMARY KEY,
first_nameVARCHAR(50),
last_nameVARCHAR(50),
   email VARCHAR(100) UNIQUE,
hire_date DATE,
job_idVARCHAR(50),
   salary DECIMAL(10, 2) CHECK (salary > 0)
);
```

Add Integrity Constraints:

```
-- Adding a foreign key constraint on job_id
-- Replace 'jobs' with the actual table name or list of job codes if available
ALTER TABLE employees
ADD CONSTRAINT fk_job_id FOREIGN KEY (job_id) REFERENCES jobs(job_id);

-- Adding a unique constraint on email
ALTER TABLE employees
ADD CONSTRAINT unique_email UNIQUE (email);
```

Alter Table Structure:

```
-- Adding a new column department_id
ALTER TABLE employees
ADD department_id INT;
-- Adding a foreign key constraint on department_id
-- Replace 'departments' with the actual table name or list of department IDs if available
ALTER TABLE employees
ADD CONSTRAINT fk_department_id FOREIGN KEY (department_id) REFERENCES
departments(department_id);
```

OUTPUT:

```
-----------------------------------------------
| Column Name   | Data Type          | Constraints                    |
-----------------------------------------------
| employee_id     | INT              | PRIMARY KEY                    |
| first_name      | VARCHAR(50)      | NOT NULL                       |
| last_name       | VARCHAR(50)      | NOT NULL                       |
| email           | VARCHAR(100)     | UNIQUE, NOT NULL               |
| hire_date       | DATE             | NOT NULL                       |
| job_id          | VARCHAR(50)      | FOREIGN KEY REFERENCES jobs(job_id)    |
| salary          | DECIMAL(10, 2)   | CHECK (salary > 0), NOT NULL   |
| department_id   | INT              | FOREIGN KEY REFERENCES
departments(department_id) |


-------------------------------------------------------------
| employee_id | first_name    | last_name    | email          | hire_date | job_id          | salary
| department_id |
-------------------------------------------------------------
| INT             | VARCHAR(50)| VARCHAR(50)| VARCHAR(100)| DATE       | VARCHAR(50)|
DECIMAL(10, 2) | INT         |
| PRIMARY KEY |                 |                | UNIQUE         |            | FOREIGN KEY| CHECK
(salary > 0)| FOREIGN KEY    |
|                 |                 |                |                |            | REFERENCES | NOT NULL
| REFERENCES     |
|                 |                 |                |                |            | jobs(job_id) |
departments(department_id)|
-------------------------------------------------------------
```

Experiment 4. Implementing queries in SQL using

4.1 Insertion

4.2 Retrieval (operations like union - intersect - minus - in - exists - group by and having)

4.3 Updation

4.4 Deletion

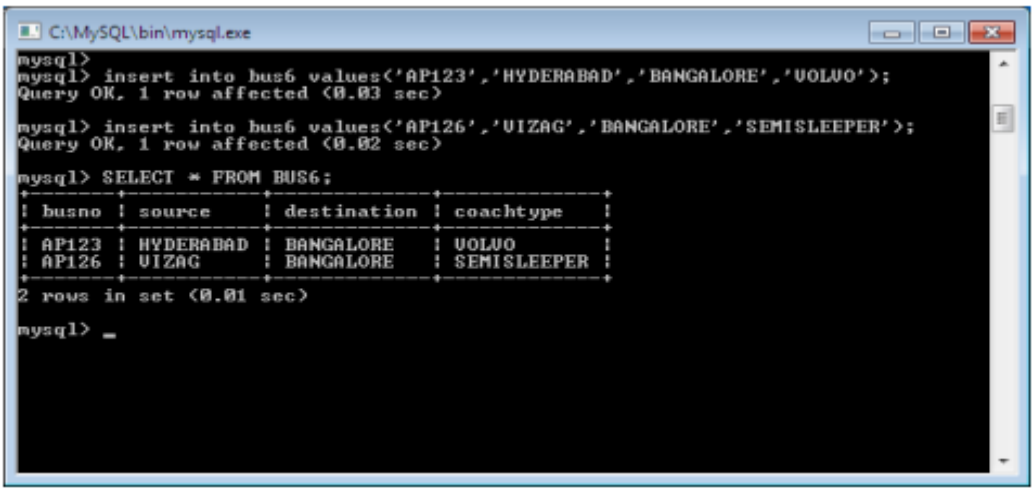
DML Commands:

1. SELECT – retrieve data from the database
2. INSERT - insert data into a table
3. UPDATE - updates existing data within a table
4. DELETE – deletes all records from a table, the space for the records remain

Commands:

Mysql>Insert into Bus values('AP123', 'Hyderabad', 'Bangalore', 'Volvo');
MySql>Insert into Bus values('AP234', 'Mumbai', 'Hyderabad', 'Semi-sleeper');

Mysql> Select * from Bus;



sql>Insert into Passenger values(82302, 'Smith',23, 'M', 'Hyderabad', '9982682829');

Mysql> Select * from Passenger;

PassportID	Name	Age	Sex	Address	Contact No
8939034	Smith	23	M	Hyderabad	983893023
9820023	John	24	M	Mumbai	983893093
8738939	Kavitha	22	F	Hyderabad	998383673

Mysql> Insert into Passenger_Ticket values('AP123',82302);

//Insert 5 or more records like-wise//

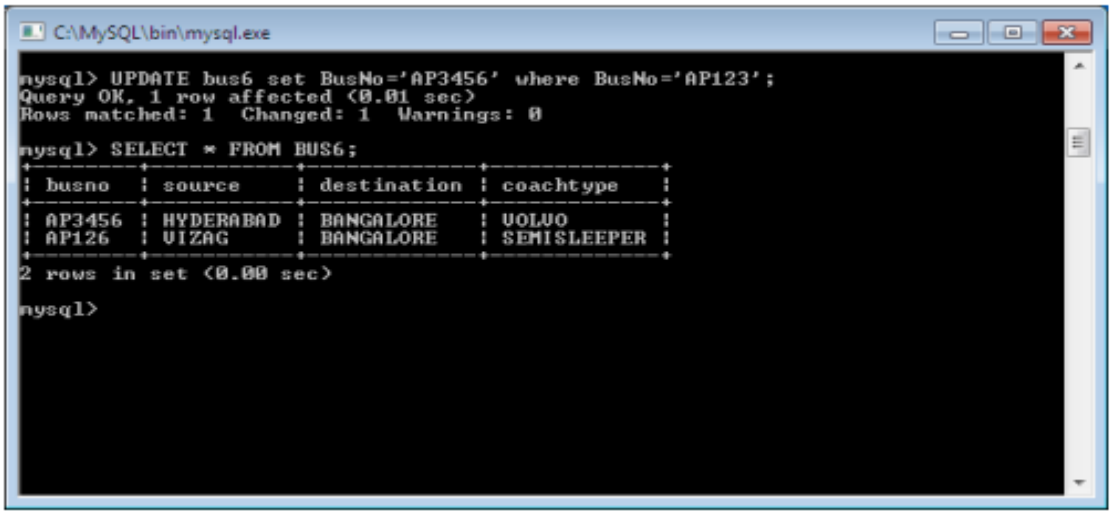
Mysql> Select * from Passenger_Ticket;

PassportID	TicketNo
8738939	453
5443243	332

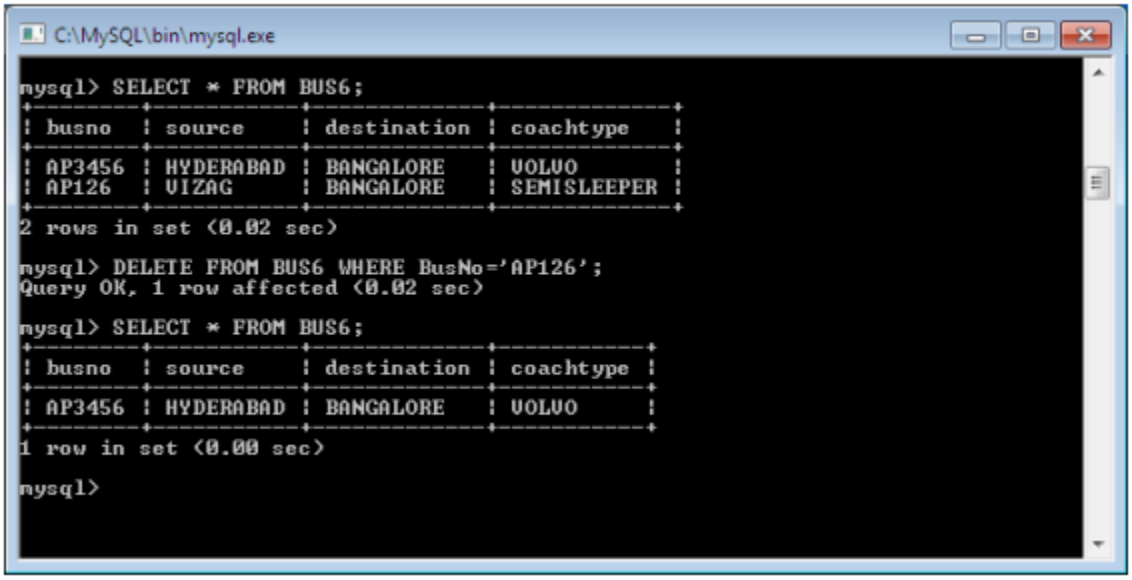
Mysql> Insert into Ticket values(29823, 'AP123',82302, '21-03-2014', '4:00PM',
'98202030334'); //Insert 5 or more records like-wise//

TicketNo	BusNo	PassportID	DOJ	Dept_time	Contact NO
29823	AP123	82302	21-03-2014	4 pm	9832434354
34353	AP234	32243	12-04-2014	5pm	9855645433

Mysql>UPDATE BUS set BusNo='AP3456' where BusNo='AP123';



Mysql> DELETE FROM BUS WHERE BusNo='AP126';



5. Implementing the concepts of Rollback – commit, checkpoints and Views

1. Rollback-Commit:
Objective: To understand the concept of rollback and commit in database transactions.
Experiment Steps:
Create a table (e.g., employees) with columns like id, name, and salary.
Start a transaction.
Insert some records into the table.
Check the records in the table.
Rollback the transaction.
Check the records again to verify that the changes were rolled back.
Start a new transaction.
Insert records again.
Commit the transaction.
Check the records to verify that the changes were committed.

2.Checkpoints:
Objective: To understand the concept of checkpoints in database recovery.
Experiment Steps:
Create a table as mentioned above.
Insert some records into the table.
Take a checkpoint.
Insert more records into the table.
Simulate a crash or system failure.
Restore the database from the checkpoint.
Check the records to verify that only the records up to the checkpoint are present.

3. Views:
Objective: To understand the concept of views in a database and how they can be used to simplify complex queries.
Experiment Steps:
Create a table as mentioned above.
Insert some records into the table.
Create a view that selects specific columns from the table.
Query the view to see the selected columns.
Update some records in the table.
Query the view again to see the updated records.

1. Rollback-Commit:

```
-- Create table
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(255),
    salary DECIMAL(10, 2)
```

```
);

-- Start transaction
START TRANSACTION;

-- Insert records
INSERT INTO employees (id, name, salary) VALUES (1, 'Alice', 50000.00);
INSERT INTO employees (id, name, salary) VALUES (2, 'Bob', 60000.00);

-- Check records
SELECT * FROM employees;

-- Rollback transaction
ROLLBACK;

-- Check records after rollback
SELECT * FROM employees;

-- Start new transaction
START TRANSACTION;

-- Insert records again
INSERT INTO employees (id, name, salary) VALUES (1, 'Alice', 50000.00);
INSERT INTO employees (id, name, salary) VALUES (2, 'Bob', 60000.00);

-- Commit transaction
COMMIT;

-- Check records after commit
SELECT * FROM employees;
```

2. Checkpoints:

```
-- Create table (if not exists)
CREATE TABLE IF NOT EXISTS employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    salary REAL
);

-- Insert records
INSERT INTO employees (id, name, salary) VALUES (1, 'Alice', 50000.00);
INSERT INTO employees (id, name, salary) VALUES (2, 'Bob', 60000.00);

-- Checkpoint (in SQLite, this is not a standard SQL command, it's used for testing)
```

```
PRAGMA wal_checkpoint;

-- Insert more records
INSERT INTO employees (id, name, salary) VALUES (3, 'Charlie', 70000.00);
INSERT INTO employees (id, name, salary) VALUES (4, 'David', 80000.00);

-- Simulate crash

-- Restore from checkpoint (in SQLite, this is not a standard SQL command, it's used for testing)
PRAGMA wal_checkpoint(RESTART);

-- Check records after restore
SELECT * FROM employees;

3. Views:
-- Create a view
CREATE VIEW employee names AS
SELECT id, name FROM employees;

-- Query the view
SELECT * FROM employee _names;

-- Update records in the table
UPDATE employees SET name = 'Alice Smith' WHERE id = 1;

-- Query the view again to see the updated records
SELECT * FROM employee_names;
```

OUTPUT:-

1. Rollback-Commit:

Before rollback (SELECT * FROM employees):

```
id  |  name  |  salary
-----|--------|---------
1   |  Alice |  50000.0
2   |  Bob   |  60000.0
```

After rollback (SELECT * FROM employees):

```
id  |  name  |  salary
-----|--------|---------
1   |  Alice |  50000.0
2   |  Bob   |  60000.0
```

After commit (SELECT * FROM employees):
id	name	salary
1	Alice	50000.0
2	Bob	60000.0

2.Checkpoints:

Before checkpoint (SELECT * FROM employees):
id	name	salary
1	Alice	50000.0
2	Bob	60000.0

After checkpoint and additional inserts (SELECT * FROM employees):
id	name	salary
1	Alice	50000.0
2	Bob	60000.0
3	Charlie	70000.0
4	David	80000.0

After restore from checkpoint (SELECT * FROM employees):
id	name	salary
1	Alice	50000.0
2	Bob	60000.0

3.Views:
View output (SELECT * FROM employee_names):
id	name
1	Alice
2	Bob
3	Charlie
4	David
After updating a record in the table (SELECT * FROM employee_names):
id	name
1	Alice Smith
2	Bob
3	Charlie
4	David

Experiment 6 :- implement joins- sub queries- nested and co related nested queries

1. Joins:
Objective: To understand how different types of joins (e.g., INNER JOIN, LEFT JOIN) work in a database.
Experiment Steps:
Create two tables (e.g., employees and departments) with relevant columns.
Insert some records into both tables.
Perform various types of joins (INNER JOIN, LEFT JOIN, etc.) between the tables.
Observe the results to understand how the joins combine data from the tables.

2.Subqueries:
Objective: To understand how to use sub queries to retrieve data based on the result of another query.
Experiment Steps:
Write a query to select some data from one table.
Use this query as a subquery in another query to filter or retrieve additional data.

3. Nested Queries:
Objective: To understand how to use nested queries to perform complex data retrieval operations.
Experiment Steps:
Write a query that contains another query (nested query) within its WHERE clause or FROM clause.
Use the nested query to filter or retrieve data based on certain conditions.

4.Correlated Nested Queries:
Objective: To understand how to use correlated nested queries to perform operations that depend on the outer query.
Experiment Steps:
Write a query that contains a nested query where the inner query references a column from the outer query.
Use this correlated nested query to filter or retrieve data based on the context of the outer query.

1.Joins:

```
-- Create tables
CREATE TABLE employees (
   id INT PRIMARY KEY,
   name VARCHAR(255),
department_id INT
);

CREATE TABLE departments (
   id INT PRIMARY KEY,
   name VARCHAR(255)
);

-- Insert records
```

```
INSERT INTO employees (id, name, department_id) VALUES (1, 'Alice', 1);
INSERT INTO employees (id, name, department_id) VALUES (2, 'Bob', 2);
INSERT INTO employees (id, name, department_id) VALUES (3, 'Charlie', 1);

INSERT INTO departments (id, name) VALUES (1, 'HR');
INSERT INTO departments (id, name) VALUES (2, 'IT');

-- INNER JOIN
SELECT employees.name, departments.name AS department
FROM employees
INNER JOIN departments ON employees.department_id = departments.id;

-- LEFT JOIN
SELECT employees.name, departments.name AS department
FROM employees
LEFT JOIN departments ON employees.department_id = departments.id;
```

2.Subqueries:

```
-- Subquery to get department names
SELECT name
FROM departments
WHERE id IN (
   SELECT department_id
   FROM employees
);

-- Subquery to get employees with salary greater than average
SELECT name
FROM employees
WHERE salary > (
   SELECT AVG(salary)
   FROM employees
);
```

3.Nested Queries:

```
-- Nested query to get employees in HR department
SELECT name
FROM employees
WHERE department_id = (
   SELECT id
   FROM departments
   WHERE name = 'HR'
);
```

```
-- Nested query in FROM clause to get average salary by department
SELECT department_id, AVG(salary) AS avg_salary
FROM (
   SELECT e.department_id, e.salary
   FROM employees e
) AS sub query
GROUP BY department_id;
```

4. Correlated Nested Queries:

```
-- Correlated nested query to get employees with salary greater than department average
SELECT name
FROM employees e
WHERE salary > (
   SELECT AVG(salary)
   FROM employees
   WHERE department_id = e.department_id
);
```

OUTPUT:-

1. Joins:

INNER JOIN:

```
name    | department
--------|-----------
Alice   | HR
Bob     | IT
Charlie | HR
```

LEFT JOIN:

```
name    | department
--------|-----------
Alice   | HR
Bob     | IT
Charlie | HR
```

2. Subqueries:

Subquery to get department names:

```
name
----
HR
IT
```

Subquery to get employees with salary greater than average:

```
name
-----
```

Bob
Charlie

3.Nested Queries:
Nested query to get employees in HR department:
name

Alice
Charlie

Nested query in FROM clause to get average salary by department:
department_id	avg_salary
1	50000.00
2	60000.00

4.Correlated Nested Queries:
Correlated nested query to get employees with salary greater than department average:
name

Bob
David

Experiment-07: Experiment with built in functions in oracle (Numeric ,String,Date,Aggregation operators)

AIM:PerformingthequeryingusingAggregatefunctions(COUNT,SUM,AVG,andMAXandMIN),GROUP BY,HAVINGandCreationanddroppingofViews.

RECOMMENDEDHARDWARE/SOFTWAREREQUIREMENTS:

- **Hardware Requirements: Intel Based desktop PC with minimum of 166 MHZ or fasterprocessorwithatleast1GBRAMand500MBfree diskspace.**
- **MySQL5.6.1**

PRE-REQUISITES: Student must know about the RDBMS-SQL

Aggregate Functions:

a) AVG: Retrieve average value of a column.

b) SUM: Retrieve the sum of all unique values in a column

c) COUNT: Retrieve the count of a column

d) MAX: Retrieve the maximum value of a column

e) MIN:Retrieve the minimum value of a column

CLAUSES:

a)ORDERBY*:*

The ORDERBY clause sorts the results of a query in ascending or descending order.

b)GROUPBY*:*

Sometimes we want to apply aggregate functions to groups of rows

c)HAVING*:*

HAVING Is like *a* WHERE *clause except that it applies to the results of a GROUP BY query.*

3. VIEW:

A VIEW is a table whose rows are not explicitly stored in the database but are computed as needed from'view definition'.

AVIEW is a computed table by taking reference from base tables.

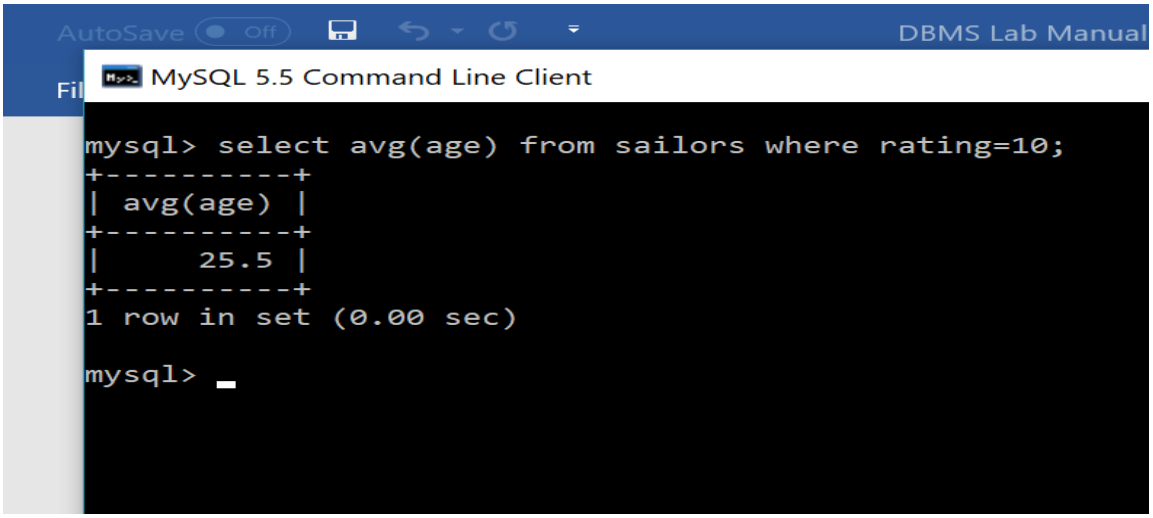
QUERIES:

1. AGGREGATEFUNCTIONS

a) AVG:

MySQL>select avg(age)from sailors where

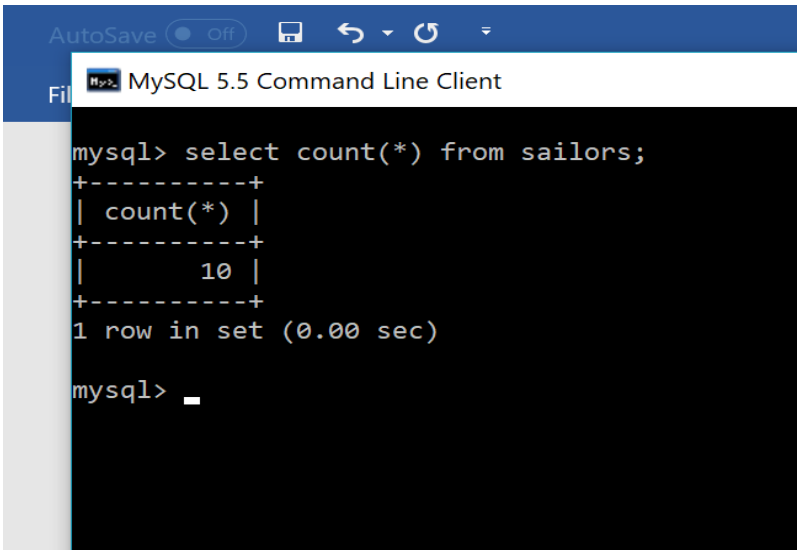
rating=10;OUTPUT:



b) COUNT:

MySQL>select count(*)from sailors;

OUTPUT:



c) MAX:

MySQL>select max(age)from sailors;

OUTPUT:

MySQL 5.5 Command Line Client

```
mysql> SELECT MAX(AGE) FROM SAILORS;
+----------+
| MAX(AGE) |
+----------+
|     63.5 |
+----------+
1 row in set (0.00 sec)

mysql>
```

d) MIN:

MySQL>select min(age)from sailors;

OUTPUT:

MySQL 5.5 Command Line Client

```
mysql> select min(age) from sailors;
+----------+
| min(age) |
+----------+
|       16 |
+----------+
1 row in set (0.00 sec)

mysql>
```

e) SUM:

MySQL>select sum(age)from sailors;

OUTPUT:

MySQL 5.5 Command Line Client

```
mysql> select sum(age) from sailors;
+----------+
| sum(age) |
+----------+
|      369 |
+----------+
1 row in set (0.00 sec)

mysql>
```

2. CLAUSES:

a) ORDERBY:

MySQL>selects name,rating from sailors ORDERBY age;

OUTPUT:

MySQL 5.5 Command Line Client

```
mysql> select sname,rating from sailors ORDER BY age;
+---------+--------+
| sname   | rating |
+---------+--------+
| zorba   |     10 |
| art     |      3 |
| andy    |      8 |
| brutus  |      1 |
| horatio |      7 |
| rusty   |     10 |
| horatio |      9 |
| dustin  |      7 |
| lubber  |      8 |
| bob     |      3 |
+---------+--------+
10 rows in set (0.00 sec)

mysql>
```

b) GROUPBY:

MySQL>selects name,avg(rating)as average from sailors GROUPBYs name;

OUTPUT:

MySQL 5.5 Command Line Client

```
mysql> select sname,avg(rating) as average from sailors GROUP BY  sname;
+---------+---------+
| sname   | average |
+---------+---------+
| andy    |  8.0000 |
| art     |  3.0000 |
| bob     |  3.0000 |
| brutus  |  1.0000 |
| dustin  |  7.0000 |
| horatio |  8.0000 |
| lubber  |  8.0000 |
| rusty   | 10.0000 |
| zorba   | 10.0000 |
+---------+---------+
9 rows in set (0.05 sec)

mysql>
```

c) HAVINGCLAUSE:

MySQL>selects name,avg(rating)as average from sailors

GROUPBYs name HAVING avg(rating)>8;

OUTPUT:

MySQL 5.5 Command Line Client

```
mysql> select sname,avg(rating) as average from sailors GROUP BY  sname HAVING avg(rating)>8;
+-------+---------+
| sname | average |
+-------+---------+
| rusty | 10.0000 |
| zorba | 10.0000 |
+-------+---------+
2 rows in set (0.00 sec)

mysql>
```

3. CREATIONOFVIEW:

MySQL>create views sailors(sname,sid)as selectS.sname,S.sid from sailors SwhereS.age=35;

MySQL>select*from ssailors;

OUTPUT:

MySQL 5.5 Command Line Client

```
mysql> create view ssailors(sname,sid) as select S.sname,S.sid from sailors S where S.age=35;
Query OK, 0 rows affected (0.11 sec)

mysql> select * from ssailors;
+---------+-----+
| sname   | sid |
+---------+-----+
| rusty   |  58 |
| horatio |  64 |
| horatio |  74 |
+---------+-----+
3 rows in set (0.00 sec)

mysql>
```

4. DROPTHEVIEW:

MySQL>drop views sailors;

MySQL>descssailors;

MySQL 5.5 Command Line Client

```
mysql> drop view ssailors;
Query OK, 0 rows affected (0.00 sec)

mysql> desc ssailors;
ERROR 1146 (42S02): Table 'sunil.ssailors' doesn't exist
mysql>
```

RESULT:

Studentsareabletoperformthequeryingbyusingtheabovecommands<u>VIVA-VOICE:</u>

1. Differentiate between SUM and COUNT?
2. Infer the output when we use MIN?
3. Specify the output when we use MAX?

4. Distinguish DROP from DELETE?
5. Infer the output when we use AVG?

Experiment-08: Implementing Operators on Relations using PL/SQL

AIM: Performing querying using *ANY*, ALL, IN, Exists, NOTEXISTS, UNION, INTERSECT, Constraints etc.

RECOMMENDEDHARDWARE/SOFTWAREREQUIREMENTS:

- **Hardware Requirements:Intel Based desktop PC with minimum of166MHZorfasterprocessorwithatleast64MBRAMand100MBfree disk space.**
- **MySQL5.6.1**

PRE-REQUISITES: Student must know about the RDBMS-Basic forms of SQL

OPERATORS:

1. UNION*:*

UNION is used to combine results of two or more SELECT statements. it eliminates the duplicate rows from result set.

2. INTERSECT:

Intersectoperationisusedtocombinetwoselectstatements,butitonlyreturnstherecordswhichareCOMMON frombothSELECT statements

3. MINUS/EXCEPT:

TheminusoperationcombinesresultsoftwoSELECTstatementsandreturnonlythoseinthefinalresult,

whichbelongs to the first set of the result

4. IN:

Thein-operator helps to connect inner query to outer query and also

allows to test whether a value is in a given set of elements.

5. NOT-IN:

The not-in operator is used as opposite of IN operator

6. EXISTS:

The exists operator is used to search for the presence of a row in a specified table that meets a certain criterion. It allows us to test whether a set is non empty or not.

NOTEXISTS:

It is used opposite to EXISTS OPERATOR.

7. ANY:

It compares a value to any applicable value in the list as per the condition

8. ALL:

ALL operator is used to select all tuples of SELECT statements.

QUERIES:

UNION:

MySQL>DisplayuniquesidofallreservationM

ysql>Selectdistictsidfromreserves;

->UNION

->Select*from boats;

Select MySQL 5.5 Command Line Client

```
mysql> select * from reserves
    -> UNION
    -> select * from boats;
+------+-----------+------------+
| sid  | bid       | day        |
+------+-----------+------------+
|   22 | 101       | 1998-10-10 |
|   22 | 102       | 1998-10-10 |
|   22 | 103       | 1998-08-10 |
|   22 | 104       | 1998-07-10 |
|   31 | 102       | 1998-10-11 |
|   31 | 103       | 1998-06-11 |
|   31 | 104       | 1998-12-11 |
|   64 | 101       | 1998-05-09 |
|   64 | 102       | 1998-08-09 |
|   74 | 103       | 1998-08-09 |
|  101 | interlake | blue       |
|  102 | interlake | red        |
|  103 | interlake | green      |
|  104 | marine    | red        |
+------+-----------+------------+
14 rows in set (0.00 sec)

mysql> _
```

INTERSECT:

MySQL>Select sid from sailors

->INTERSECT

->Selects id from reserves;

OUTPUT:

Sid

22

31

64

74

EXCEPT:

MySQL>selects id from sailors

>EXCEPT

>Selects id from reserves

;OUTPUT:

Sid

29

32

58

71

85

95

4.IN-OPERATOR:

MySQL>selects.sname from sailors

MySQL 5.5 Command Line Client

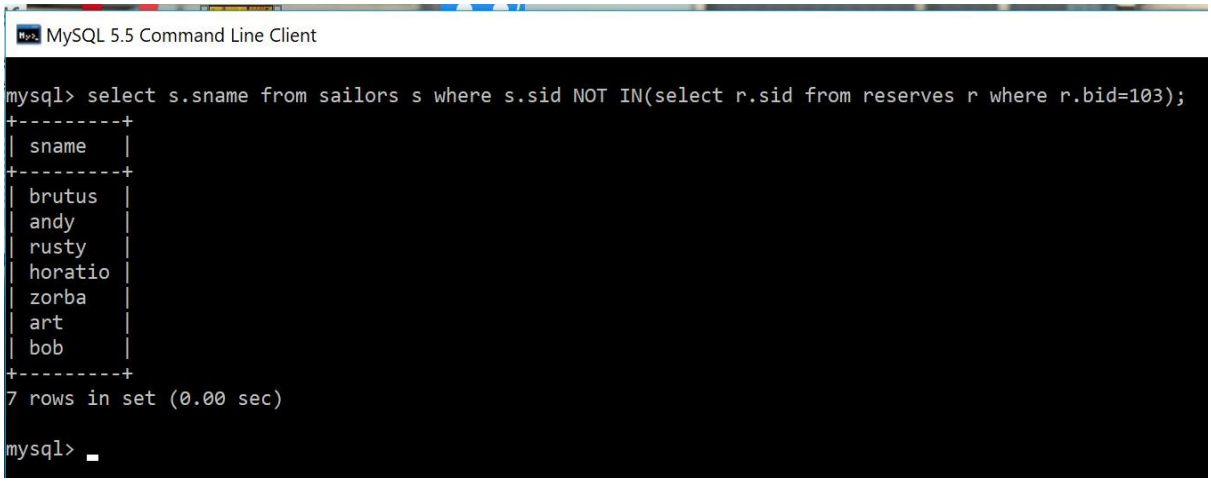
```
mysql> select s.sname from sailors s where s.sid IN(select r.sid from reserves r where r.bid=103);
+---------+
| sname   |
+---------+
| dustin  |
| lubber  |
| horatio |
+---------+
3 rows in set (0.00 sec)

mysql>
```

wheres.sidIN(selectr.sidfromreservesrwherer.bid=103);

5.NOT-IN:

MySQL>selects. sname fromsailorsswheres.sid NOTIN(selectr.sid fromreservesrwherer.bid=103);

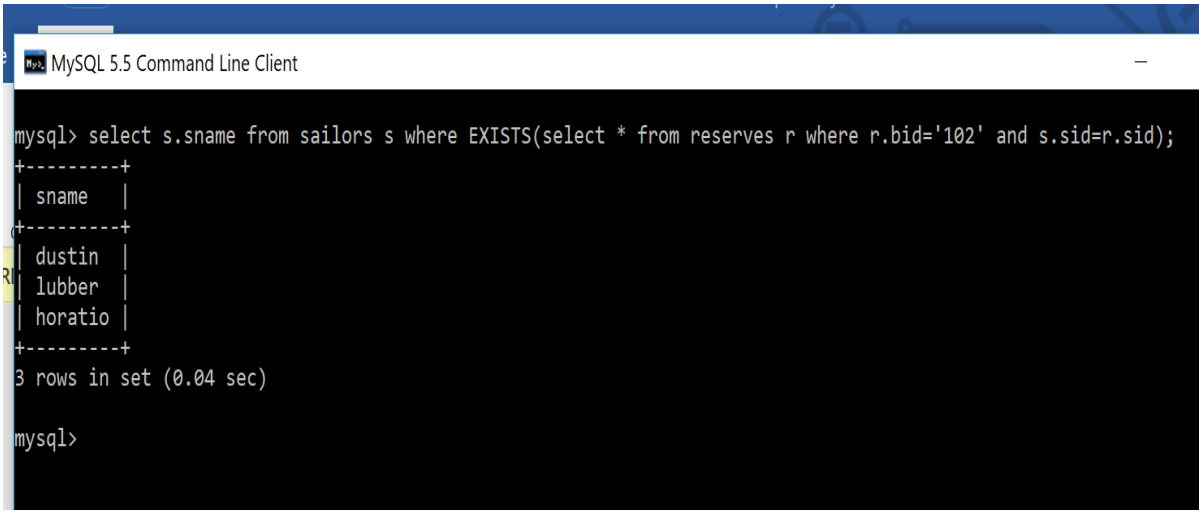
OUTPUT:



6. EXISTS:

MySQL>selects. sname from sailors where EXISTS(select*from reserves wherer.bid=102AND s.sid=r.sid);

OUTPUT:



7. NOTEXISTS:

MySQL>selects.sname from sailors swhere NOT

EXISTS(select*from reservesrwherer.bid=102ands.sid=r.sid);

OUTPUT:

8.ANY:

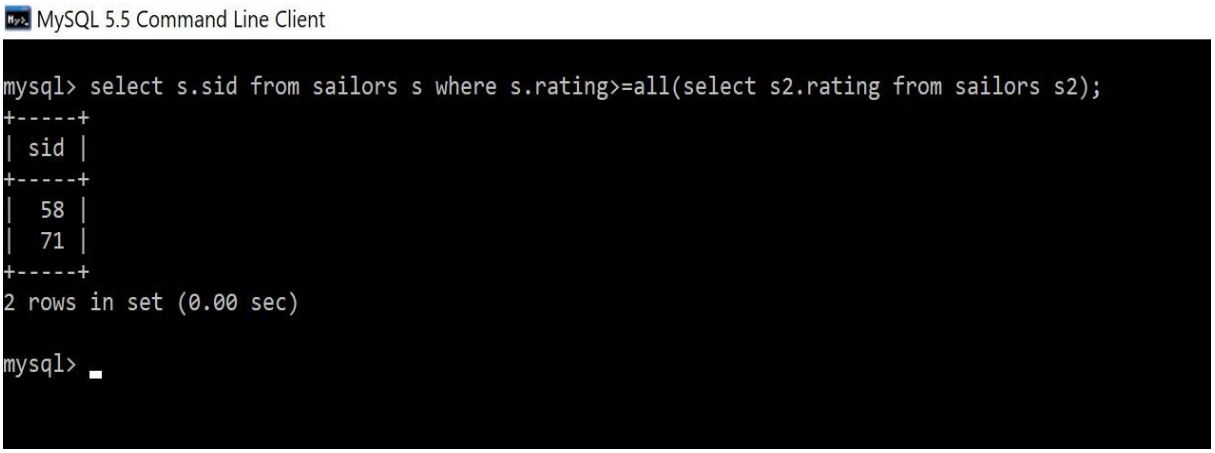
Selects.sid from sailors swheres.rating>ANY(selects2.ratingfromsailorss2where s2.sname='horatio');

OUTPUT:

9.ALL:

MySQL>selects.sid from sailors swheres.rating>=ALL(selects2.rating fromsailorss2);

OUTPUT:



RESULT:TheStudentisableexecutetheQueriesbyusingtheaboveoperators.VIVA-

VOICE:

1. Specify the result of String functions?
2. Specify the result of Date functions?
3. Infer the result of conversion function?
4. Define Concatenation?
5. Differentiate between LTRIM and RTRIM?

Experiment-09: Implemeting functions, stored procedures using PL/SQL.

AIM: Creating and Executing Stored procedures.

RECOMMENDEDHARDWARE/SOFTWAREREQUIREMENTS:

- **Hardware Requirements:Intel Based desktop PC with**

minimum of 166MHZorfaster processor with atleast 64MBRAM and100 MBfree disk space.

- **MySQL5.6.1**

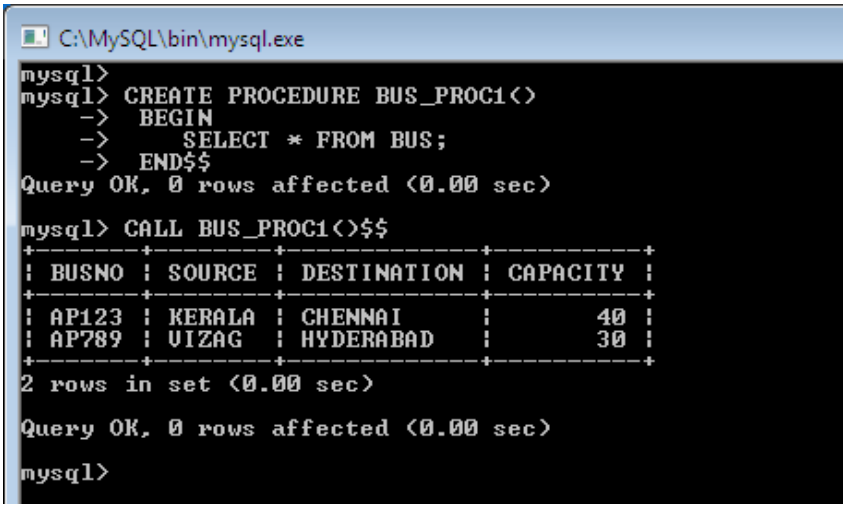
PRE-REQUISITES: Student must know about the Relational Database SQL-Procedures PROCEDURE: A stored procedure isasetofStructuredQueryLanguage(SQL)statementswithanassigned name,whicharestoredinarelationaldatabasemanagementsystemasagroup,soitcanbe reused and shared by multiple programs.

Ex1:ExecutingaSimpleprocedurecalledBUS_PROC1(),When we execute it will display all the datafrom“bus”table.

```
Delimiter$$
CREATEPROCEDUREBUS
 _PROC1()BEGIN
   SELECT*FROMBUS;
   SELECT
*FROMBUS_AUDIT1;
DESCBUS;
 END$$
```

OUTPUT:

```
CALLBUS_PROC1 ()$$
```

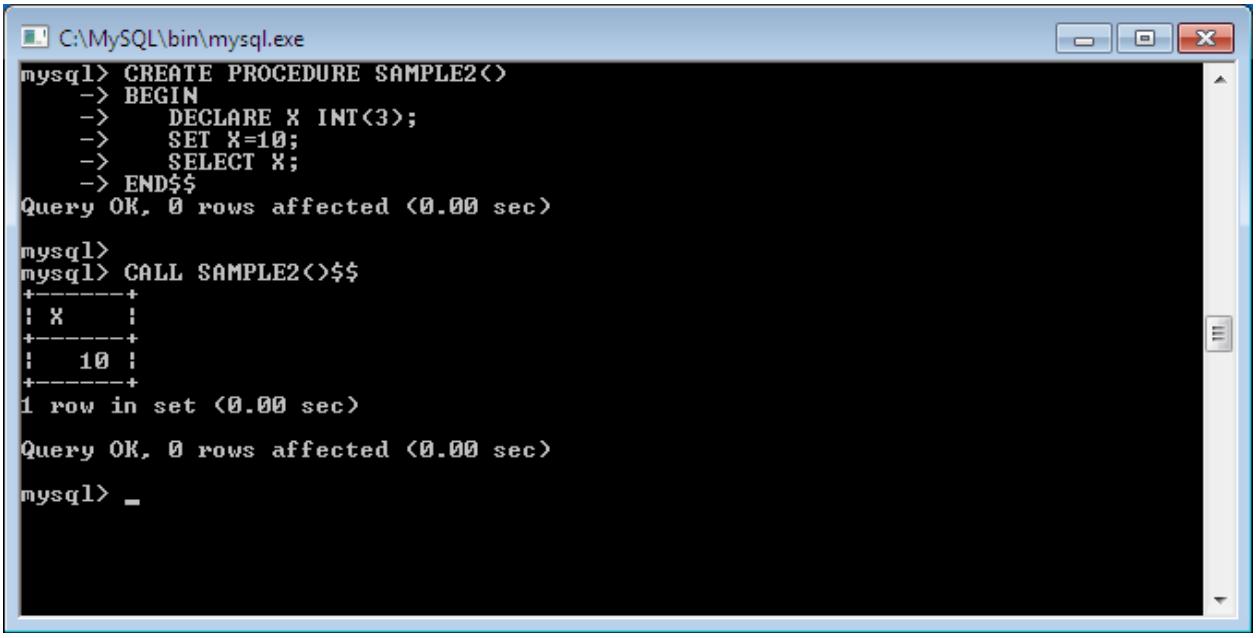


Ex2: Executing the procedure to show the declaration of local variables in a stored procedure.

Local variables are declared within stored procedures and are only valid between Begin and END. Block where they are declared. Local variables can have any SQL data type.

OUTPUT:

```
CREATE PROCEDURE
SAMPLE2()BEGIN
DECLARE X
INT(3);SETX=
10;
SELECTX;
END$$

CALL SAMPLE2()$$
```



Ex3:ExecutingtheProcedureparameter- OUT

The following example shows a simple stored procedure that uses an OUT parameter. CREATE PROCEDURE SIMPLE_PROC(OUTPARAMINT)

BEGIN

SELECT COUNT (*)INTO PARAM FROM BUS;

END$$

In the body of the procedure, the parameter will get the count value from the table bus. After calling the procedure the work OUT tells the DBMS that the values goes outfromtheprocedure.Hereparam1isthenameoftheoutput parameter and we have passed its value to a session variable

named @a,in the call statement.

OUTPUT:

CALLSIMPLE_PROC(@a)$$

Queryok,1rowaffected(0.

22sec)SELECT@a$$

```
mysql> create procedure simple(out param1 int)
    -> begin
    -> select count(*) into param1 from bus;
    -> end $$
Query OK, 0 rows affected (0.29 sec)

mysql> call simple(@a)$$
Query OK, 1 row affected (0.22 sec)

mysql> select @a$$
+------+
| @a   |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

RESULT:The Student is able to work on Stored Procedures.VIVA VOICE:

1. Define stored procedure?
2. When would you use stored procedure or functions?
3. State external procedures?
4. Recall input parameter and how it is different from OUT parameter?
5. Show how to use Stored Procedures

Experiment-10:Implementing Cursors using PL/SQL.

EXPERIMENT10:CURSORS

AIM: To declare MySQL cursor in stored procedure to iterate

through a result set returned by a SELECT statement.

RECOMMENDEDHARDWARE/SOFTWAREREQUIREMENTS:

- **Hardware Requirements: Intel Based desktop PC with minimum of 166 MHZ or fasterprocessorwithatleast64MBRAMand100MBfree diskspace.**
- **MySQL5.6.1**

PRE-REQUISITES: Student must know about the RelationalSQL-Cursors

CURSOR: To handle a result set inside a stored procedure, you use a cursor.Acursor allow you to iterate a set of rows returned by a query and process each row accordingly.

MySQL cursor is read-only, on-scrollable and as ensitive.

- **Read-only: you cannot update data in the underlying table through the cursor.**
- **Non-scrollable**:you can only fetch rows in the order determined by **The SELECT statement. You can not fetch rows in the reversed order. In addition, you cannot skip rows or jump to a specific row in the result set.**
- **A sensitive: there are two kinds of cursors: a sensitive cursor and insensitive cursor. An a sensitive cursor points to the actual data, whereas an insensitive cursor uses a temporary copy of the data. An a sensitive cursor performs faster than an insensitive cursor because it does not have to make a temporary**

copy of data. However, any change that made to the data from other connections will affect the data that is being used by an a sensitive cursor, therefore, it is safer if you do not update the data that is being used by an a sensitive cursor.MySQLcursoris a sensitive.

Working with MySQLcursor:

Step:1:Declare a cursor by using the DECLARE statement:

```
1 DECLAREcursor_nameCURSORFORSELECT_statement;
```

Thecursordeclarationmustbeafteranyvariabledeclaration.Ifyoud eclareacursorbeforevariables declaration, MySQL will issue an error. A cursor must always be associated with a SELECT statement.

Step:2:Openthecursorbyusing theOPEN statement.The OPEN statementinitializestheresultset for the cursor, therefore, you must call the OPEN statement before fetching rows from theresultset.

```
1 OPENcursor_name;
```

Step:3:FETCHstatementtoretrievethenextrowpointedbythecursorandmovethecursortothe nextrowintheresultset.

```
1 FETCHcursor_nameINTOvariableslist;
```

After that, you can check to see if there is any row available before fetching it. Declare a NOTFOUNDhandlertohandlethe situationwhenthe cursorcouldnotfindanyrow.

```
1 DECLARE CONTINUE HANDLERFORNOT FOUNDSETfinished=1;
```

Step:4:CLOSEstatementtodeactivatethecursorandreleasethememory associatedwithitasfollows:

```
1 CLOSEcursor_name;
```

Example:Developing a stored procedure that builds an email list of all employees in the employees table in the sample database.

```
DELIMITER$$
CREATEPROCEDUREbuild_email_list(INOUTemail_
listvarchar(4000))BEGIN
 DECLARE v_finished INTEGER
```

```
DEFAULT
0;DECLAREv_emailvarchar(100)
DEFAULT"";
-- declare cursor for employee email
DEClAREemail_cursor CURSOR FOR
SELECT email FROM employee;
--declare NOT FOUND handler
DECLARECONTINUEHANDLER
    FORNOT
FOUNDSETv_finished=1;OPE
N email_cursor;
get_email:LOOP
FETCHemail_cursor INTO v_email;
```

IFv_finished=1T

HENLEAVEget_

email;

END IF;

--buildemaillist

SET email_list =

CONCAT(v_email,";",email_list);E

ND LOOPget_email;

CLOSE

email_cursor;E

ND$$

```
mysql> create procedure build_email_list (INOUT email_list varchar(4000))
    -> begin
    -> declare V_finished integer default 0;
    -> declare V_email varchar(100) default "";
    -> declare email_cursor cursor for select email from employees;
    -> declare continue handler for not found set V_finished=1;
    -> open email_cursor;
    -> get_email:loop
    -> fetch email_cursor into V_email;
    -> if V_finished=1 then
    -> leave get_email;
    -> end if;
    -> set email_list=concat(V_email,";",email_list);
    -> end loop get_email;
    -> close email_cursor;
    -> end $$
Query OK, 0 rows affected (0.47 sec)
```

Youcantestthe

build_email_liststoredprocedureusingthefollowingscript

:SET@email_list=""$$

```
mysql> set @email_list=" ";
    -> $$
Query OK, 0 rows affected (0.00 sec)
```

CALLbuild_email_list(@email_list)$$

```
mysql> call build_email_list(@email_list)$$
Query OK, 0 rows affected, 1 warning (0.20 sec)
```

SELECT@email_list$$

```
mysql> select @email_list$$
+-------------------------------------------------------+
| @email_list                                           |
+-------------------------------------------------------+
| pankaj@soft.com;havi@soft.com;kar@soft.com;a@soft.com;  |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

RESULT: The Student is able to work on Cursors.

VIVAVOICE:

1. Define a cursor?
2. List the types of cursor?
3. State the use of parameterized cursor?
4. State the use of cursor variable?
5. Define normal cursor?

Experiment-11: Implement exceptions handling using PL/SQL

```
-- Create a table to store student information
CREATE TABLE students (
   id INT PRIMARY KEY,
   name VARCHAR(50),
   age INT
);

-- Procedure to insert a student record and handle exceptions
CREATE OR REPLACE PROCEDURE Insert Student (
```

```
student_id IN INT,
student_name IN VARCHAR2,
student_age IN INT
)
IS
BEGIN
   -- Check if student_id is already present in the table
   SELECT COUNT(*)
   INTO   id_count
   FROM   students
WHERE  id = student_id;

   IF id_count> 0 THEN
      -- Raise an exception if student_id already exists
      RAISE_APPLICATION_ERROR(-20001, 'Student ID already exists');
   END IF;

   -- Insert student record into the table
   INSERT INTO students (id, name, age)
   VALUES (student_id, student_name, student_age);

   -- Commit the transaction
   COMMIT;

   DBMS_OUTPUT.PUT_LINE('Student record inserted successfully');
EXCEPTION
   WHEN OTHERS THEN
      -- Rollback the transaction in case of exception
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/

-- Enable output
SET SERVEROUTPUT ON;

-- Test the procedure
BEGIN
InsertStudent(1, 'Alice', 20);   -- This will execute successfully
InsertStudent(1, 'Bob', 25);     -- This will raise an exception
END;
/
```

OUTPUT:-
Student record inserted successfully
Error: ORA-20001: Student ID already exists

Experiment-12:Creating Triggers using PL/SQL.

AIM:To Implement the concept of triggers-Insert,Update,Delete

RECOMMENDEDHARDWARE/SOFTWAREREQUIREMENTS:

- **Hardware Requirements :Intel Based desktop PC with minimum of 166MHZ or faster processor with at least 64MBRAM and 100MB free disks pace.**
- **MySQL5.6.1**

PRE-REQUISITES: Student must know about the Relational Database SQL-Triggers.

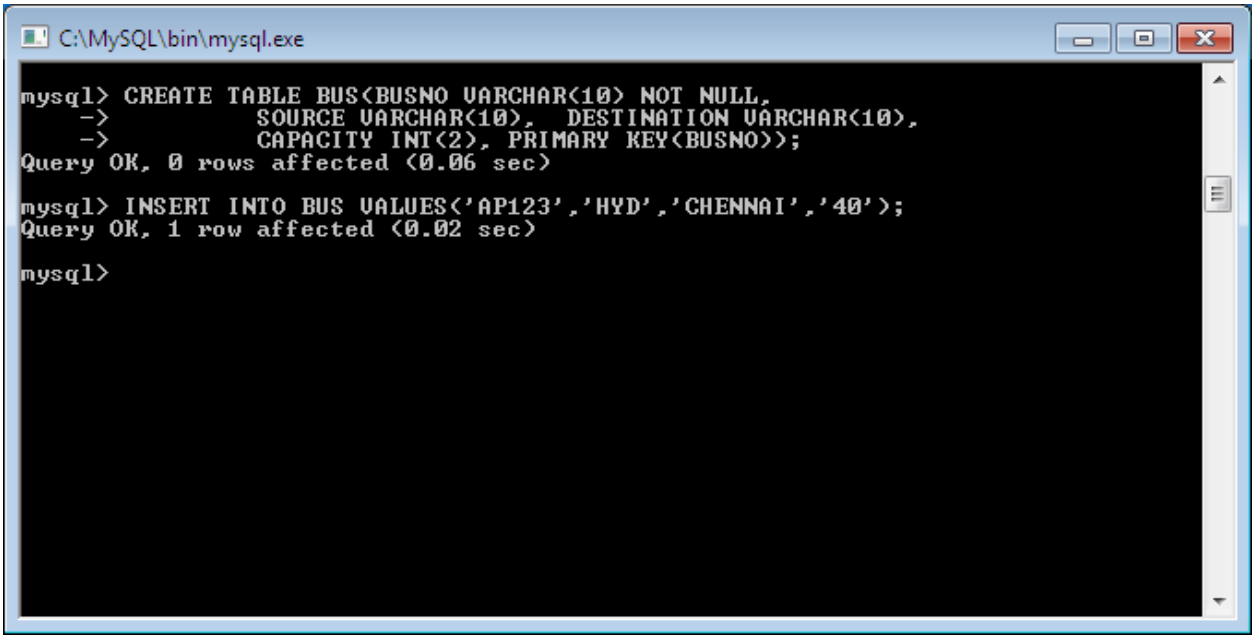
1. **Create a table with the schema Bus(busno,source, destination,capacity)**

```
MySQL>CREATETABLEBUS(B
USNOVARCHAR(10)NOTNU
LL,SOURCE
VARCHAR(10),DESTINATIO
N VARCHAR(10),CAPACITY
INT(2),PRIMARYKEY(BUSN
O));
```

2. **Insertvalues**

```
MySQL>INSERTINTOBUSVALUES('AP123','HYD','CH
ENNAI','40');
```

//Atleast 3values//



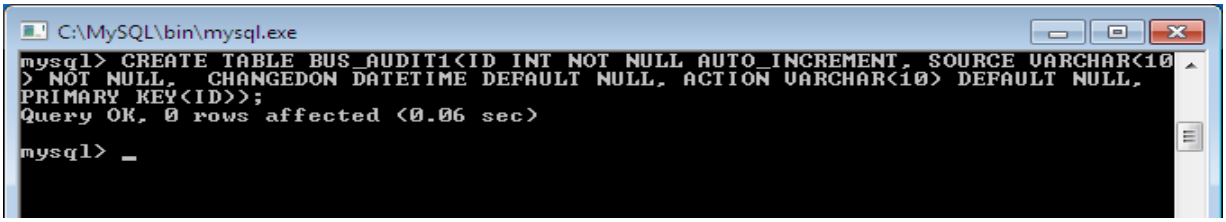
3. **Create an Audit table for the bus to track the actions on the table using Triggers concept.(Schema:Bus_Audit1(ID, Source, Changed on, Action))**

CREATETABLEBUS_AUDIT1 (IDINTNOTNULLAUTO_INCREMENT, SOURCE

VARCHAR(10)NOTNULL,CHANGEDONDATETIMEDEFA

ULTNULL,ACTIONVARCHAR(10)DEFAULTNULL,PRIMA

RYKEY(ID));



4. Creating UPDATET rigger:

DELIMITER$$

CREATE TRIGGER

BEFORE_BUS_UPDATE

BEFOREUPDATEON BUS

FOREACH

ROWBEGI

N

INSERTINTOBUS_AUDIT1

SET

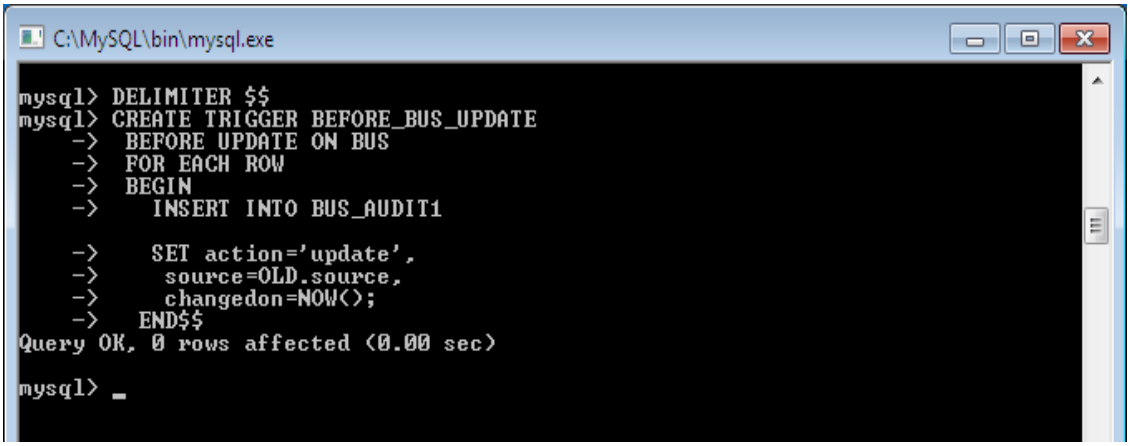
action='UPDA

TE',source=O

LD.source,cha

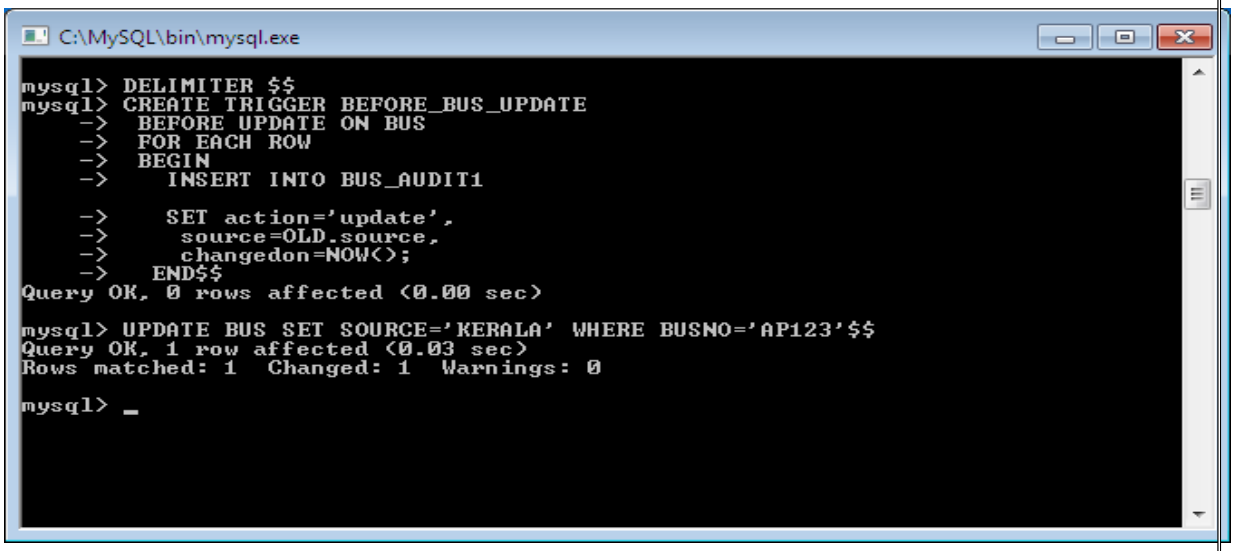
ngedon=NOW(

);END$$



Perform an UPDATE operation on the bus table:

MySQL>UPDATEBUSSETSOURCE=‘KERALA’WHEREBUSNO=‘AP123’$$



5. Creating INSERT Trigger:

CREATE TRIGGER

BEFORE_BUS_INSERTBEFO

REINSERTONBUS

FOREACH

ROWBEGI

N

INSERTINTOBUS_AUDIT1

SET

action='Insert',

source=NEW.

source,

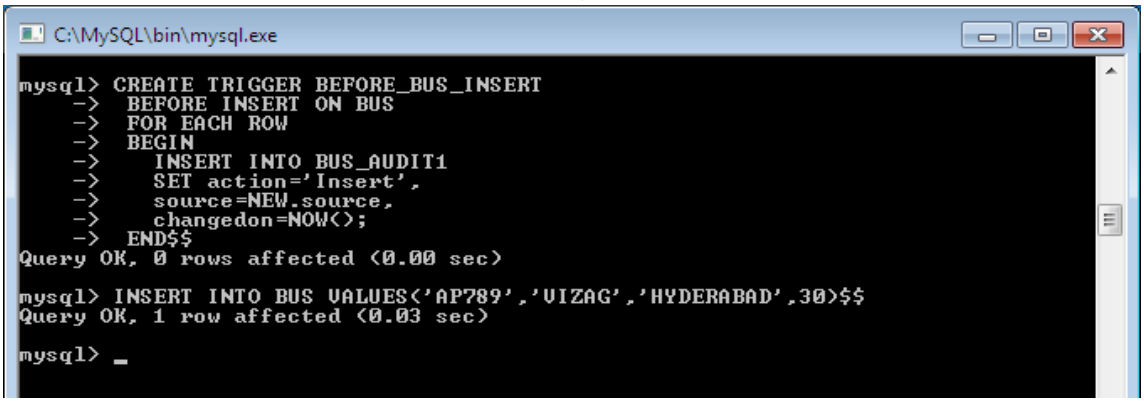
Changed

on=NOW();EN

D$$

Perform an INSERT operation on bus:

INSERT IN TO BUS VALUES('AP789','VIZAG','HYDERABAD',30)$$



6. Create DELETE Trigger

CREATE TRIGGER BEFORE_BUS_DELETEBEFOREDELETEON BUS

FOREACH ROWBEGN Insertintobus_ audit1SET action='delete', source=old.sour ce, Changed on=NOW ();END$$

Perform DELETE operation on bus:

DELETEFROMBUSWHERESOURCE='HYDERABAD'$$

7. OUTPUT:

Select*frombus_audit1$$

S.No	Source	Changedon	Action
1	Bangalore	2014:03:2312:51:00	Insert
2	Kerela	2014:03:25:12:56:00	Update
3	Hyderabad	2014:04:26:12:59:02	Delete

RESULT: The Student is able to work on Triggers to create an active database.VIVA-VOICE:

1. Define TRIGGER?
2. List the types of triggers?

3. List the trigger timings?
4. Is it possible to create a trigger on views?
5. Outline row and statement trigger?

Experiment-1

1.