Vidyavardhini's
College of Engineering & Technology
Vasai Road (W)

Department of Computer Engineering

Laboratory Manual
[Student Copy]

Semester	IV	Class	SE
Course No.	CSL402	Academic Year	2024-25 (Even Sem.)
Course Name	Database Management System Lab		

Vidyavardhini's College of Engineering & Technology

Vision

To be a premier institution of technical education; always aiming at becoming a valuable resource for industry and society.

Mission

- To provide technologically inspiring environment for learning.
- To promote creativity, innovation and professional activities.
- To inculcate ethical and moral values.
- To cater personal, professional and societal needs through quality education.

Department Vision:

To evolve as a center of excellence in the field of Computer Engineering to cater to industrial and societal needs.

Department Mission:

- To provide quality technical education with the aid of modern resources.
- Inculcate creative thinking through innovative ideas and project development.
- To encourage life-long learning, leadership skills, entrepreneurship skills with ethical & moral values.

Program Education Objectives (PEOs):

PEO1: To facilitate learners with a sound foundation in the mathematical, scientific and engineering fundamentals to accomplish professional excellence and succeed in higher studies in Computer Engineering domain
PEO2: To enable learners to use modern tools effectively to solve real-life problems in the field of Computer Engineering.
PEO3: To equip learners with extensive education necessary to understand the impact of computer technology in a global and social context.
PEO4: To inculcate professional and ethical attitude, leadership qualities, commitment to societal responsibilities and prepare the learners for life-long learning to build up a successful career in Computer Engineering.

Program Specific Outcomes (PSOs):

PSO1: Analyze problems and design applications of database, networking, security, web technology, cloud computing, machine learning using mathematical skills, and computational tools.

PSO2: Develop computer-based systems to provide solutions for organizational, societal problems by working in multidisciplinary teams and pursue a career in the IT industry.

Program Outcomes (POs):

Engineering Graduates will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9. Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **PO12. Life-long learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Objectives

1	To Develop Entity Relationship data model.
2	To develop relational Model
3	To formulate SQL queries.
4	To learn procedural interfaces to SQL queries
5	To learn the concepts of transactions and transaction processing
6	To understand how to handle concurrent transactions and able to access data through front end (using JDBC ODBC connectivity)

Course Outcomes

At the end of the course student will be able to:		PO/PSO	Bloom Level
CSL402.1	Design ER and EER diagram for the real life problem with software tool.	Design	Create (Level 6)
CSL402.2	Construct database tables with different DDL and DML statements and apply integrity constraints	Construct, Apply	Apply (Level 3)
CSL402.3	Apply SQL queries ,triggers for given Schema	Apply	Apply (Level 3)
CSL402.4	Apply procedure and functions for given schema	Apply	Apply (Level 3)
CSL402.5	Use transaction and concurrency control techniques to analyze conflicts in multiple transactions.	Use	Apply (Level 3)
CSL402.6	Construct database tables and JDBC/ ODBC connectivity for given application	Construct	Apply(Level 3)

Mapping of Experiments with Course Outcomes

Experiments	Course Outcomes					
	CSL402 .1	CSL402 .2	CSL402 .3	CSL402 .4	CSL402 .5	CSL402 .6
Identify the case study and detail statement of problem. Design an Entity-Relationship (ER) / Extended Entity-Relationship (EER) Model.	3					
Mapping ER/EER to Relational schema model.	3					
Create and populate database using Data Definition Language (DDL) and Apply Integrity Constraints for the specified system		3				
Apply DML commands for the specified system.		3				
Perform Simple queries, string manipulation operations and aggregate functions			3			
Implement SET operators and Datetime functions.			3			
Perform Nested queries and Complex queries			3			
Implement Procedure and functions				3		
Implementation of Views and Triggers			3			
Demonstrate of Database connectivity (course Project)						3

INDEX

Sr. No.	Name of Experiment	D.O.P.	D.O.C.	Page No.	Remark
1	Identify the case study and detail statement of problem. Design an Entity-Relationship (ER) / Extended Entity-Relationship (EER) Model.				
2	Mapping ER/EER to Relational schema model.				
3	Create and populate database using Data Definition Language (DDL) and Apply Integrity Constraints for the specified system				
4	Apply DML commands for the specified system.				
5	Perform Simple queries, string manipulation operations and aggregate functions				
6	Implement SET operators and Datetime functions.				
7	Perform Nested queries and Complex queries				
8	Implement Procedure and functions				
9	Implementation of Views and Triggers				
10	Demonstrate of Database connectivity (Course Project)				

D.O.P: Date of performance

D.O.C : Date of correction

Experiment No.1
Identify the case study and detail statement of problem. Design an Entity-Relationship (ER) / Extended Entity-Relationship (EER) Model.
Date of Performance:17/01/25
Date of Submission:24/01/25

Aim: Identify the case study and detail statement of problem.
Design an Entity-Relationship (ER) / Extended Entity-Relationship (EER) Model.

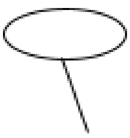
Objective: To show the relationships of entity sets attributes and relationships stored in a database

Theory: Summary of ER, EER Diagram Notation

Strong Entities

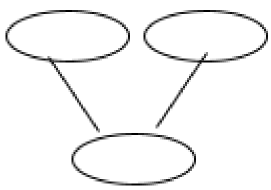
Weak Entities

Attributes

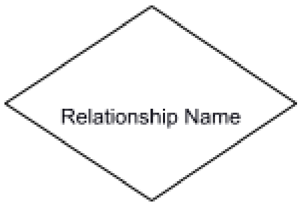


Multi Valued Attributes [Double Ellipse]

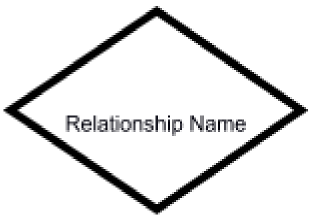
Composite Attributes



Relationships



Identifying Relationships



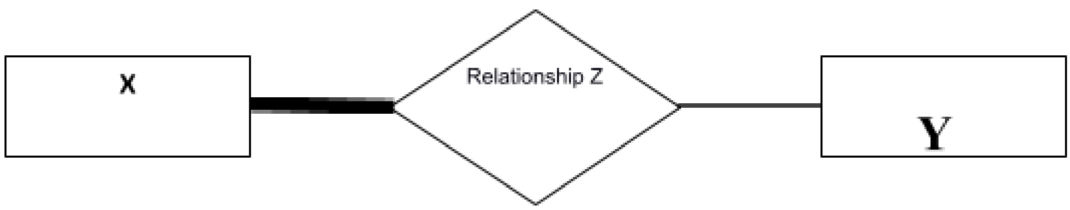
N-ary relationships

More than 2 participating entities

Constraints - Participation

- **Total Participation** - entity X has total participation in Relationship Z, meaning that every instance of X takes part in AT LEAST one relationship. (i.e. there are no members of X that do not participate in the relationship.

Example: X is Customer, Y is Product, and Z is a 'Purchases' relationship. The figure below indicates the requirement that every customer purchases a product.



- **Partial Participation** - entity Y has partial participation in Relationship Z, meaning that only some instances of Y take part in the relationship.

Example: X is Customer, Y is Product, and Z is a 'Purchases' relationship. The figure below indicates the requirement that not every product is purchases by a customer.

Cardinality:

. 1:N – One Customer buys many products, each product is purchased by only one customer.

1 N

N:1 - Each customer buys at most one product, each product can be purchased by many customers.

N 1

1:1 – Each customer purchases at most one product, each product is purchased by only one customer.

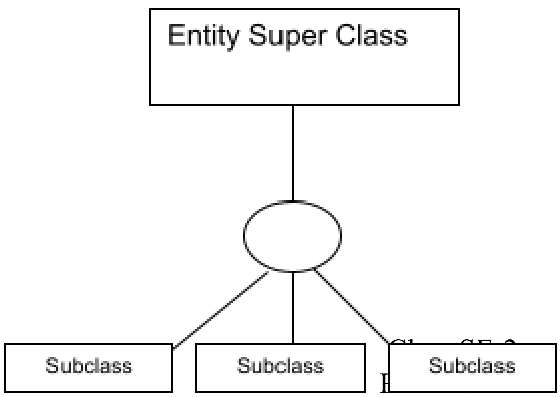
1 1

M:N – Each customer purchases many products, each product is purchased by many customers.

M N

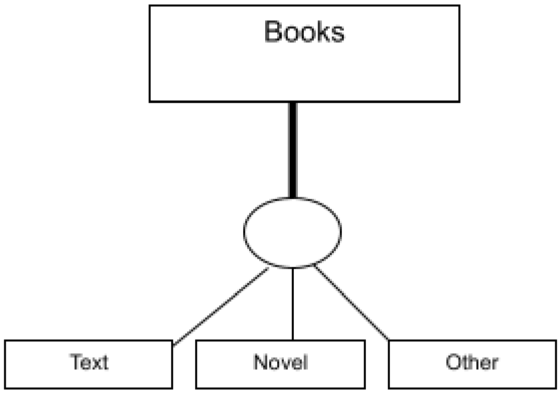
Specialization/Generalization

. Each subclass inherits all relationships and attributes from the super-class.

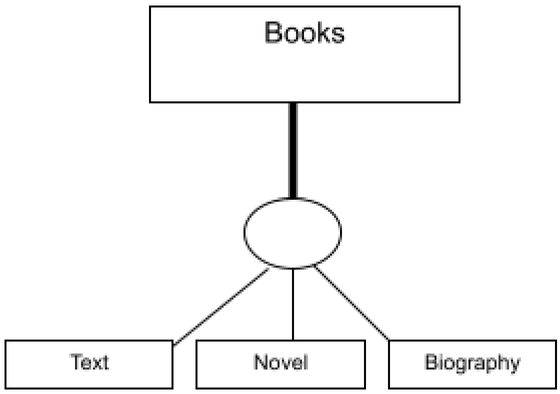


Constraints on Specialization/Generalization

Total Specialization – Every member of the super-class must belong to at least one subclass. For example, any book that is not a text book, or a novel can fit into the "Other" category.

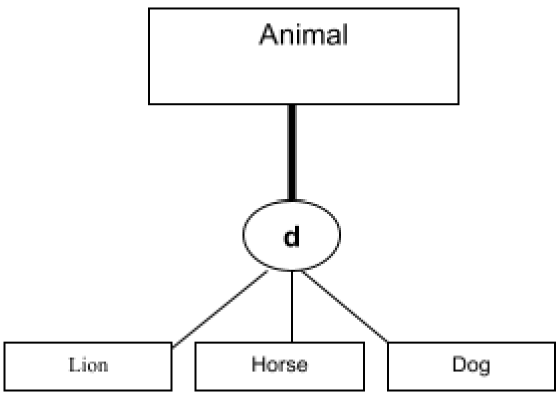


Partial Specialization – each member of the super-class may not belong to one of the subclasses. For example, a book on poetry may be neither a text book, a novel or a biography.

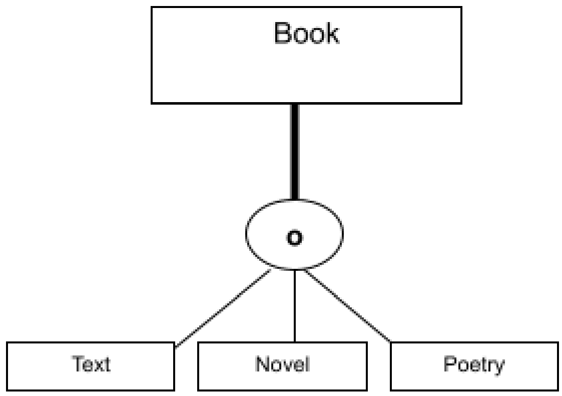


Dis-jointness Constraint

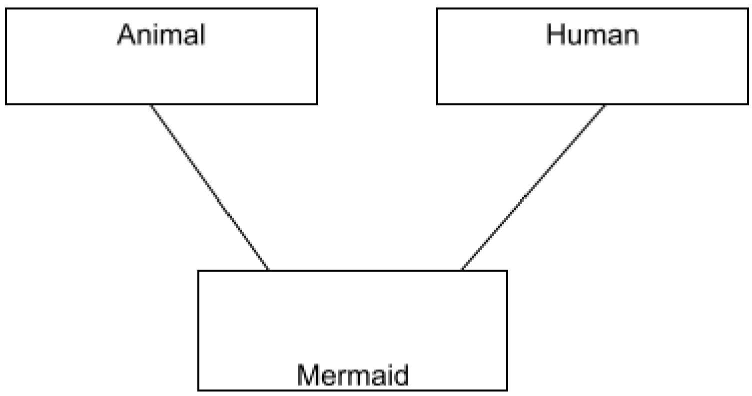
- **Disjoint** – every member of the super-class can belong to at most one of the subclasses. For example, an Animal cannot be a lion and a horse, it must be either a lion, a horse, or a dog.



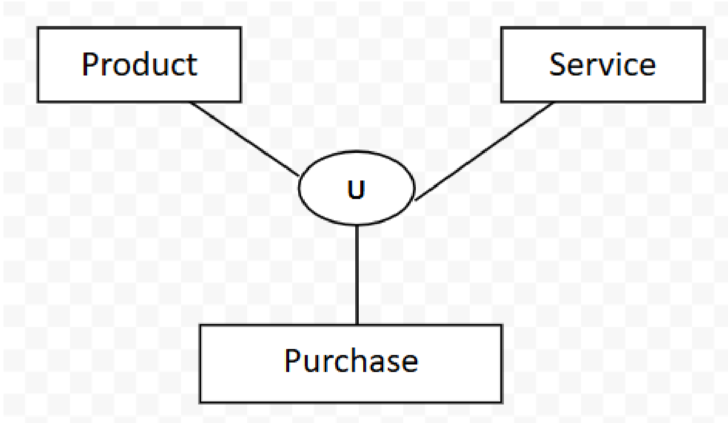
Overlapping – every member of the super-class can belong to more than one of the subclasses. For example, a book can be a text book, but also a poetry book at the same time.



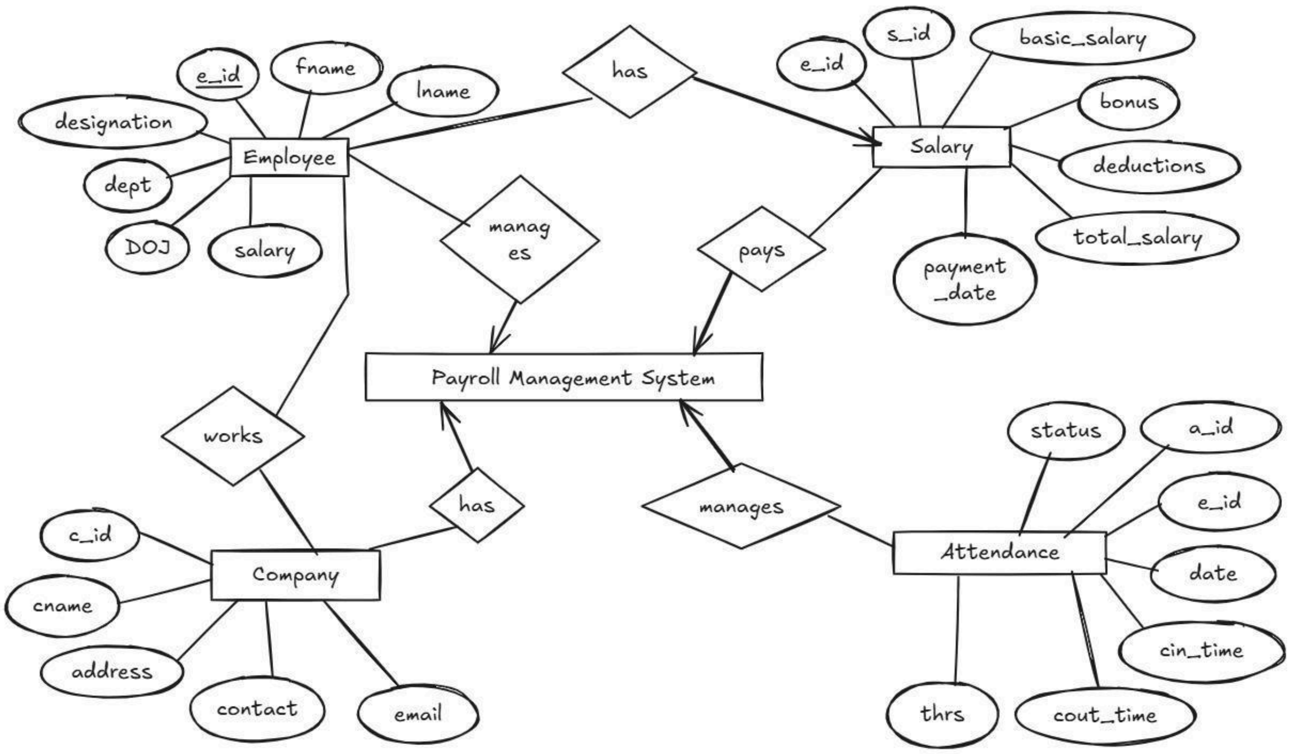
Multiple Inheritance – a subclass participates in more than one subclass/super-class relationship, and inherits attributes and relationships from more than one super-class. For example, the subclass Mermaid participates in two subclass/super-class relationships, it inherits attributes and relationships of Animals, as well as attributes and relationships of Humans.



Union – a subclass/super-class relationship can have more than one super-class, and the subclass inherits from at most one of the super-classes (i.e. the subclass purchase will inherit the relationships and attributes associated with either service or product, but not both). Each super class may have different primary keys, or the same primary key. All members of the super-classes are not members of the super-class. For example, a purchase can be a product, or a service, but not both. And all products and services are not purchase



Implementation:



Conclusion:- In this experiment, I learned to design an ER/EER model, identifying entity sets, attributes, and relationships. I understood the significance of strong and weak entities, multi-valued and composite attributes, and different types of relationships. Exploring cardinality and participation constraints helped me define real-world scenarios effectively. Specialization, generalization, and inheritance concepts provided insights into advanced database modeling.

Experiment No.2	
Mapping ER/EER to Relational schema model.	
Date of Performance:24/01/25	
Date of Submission:31/01/25	

Aim: Mapping ER/EER to Relational schema model.

Objective: objective of design is to generate a formal specification of the database schema

Theory: Mapping Rules

Step 1: Regular Entity Types
Create an *entity relation* for each strong entity type. Include all single-valued attributes. Flatten composite attributes. Keys become secondary keys, except for the one chosen to be the primary key.

Step 2: Weak Entity Types
Also create an entity relation for each weak entity type, similarly including its (flattened) single-valued attributes. In addition, add the primary key of each owner entity type as a foreign key attribute here. Possibly make this foreign key CASCADE.

Step 3: Binary 1:1 Relationship Types
Let the relationship be of the form [S]——<R>——[T].

1. **Foreign key approach**: The primary key of T is added as a foreign key in S. Attributes of R are moved to S (possibly renaming them for clarity). If only one of the entities has total participation it's better to call it S, to avoid null attributes. If neither entity has total participation nulls may be unavoidable. *This is the preferred approach in typical cases.*
2. **Merged relation approach**: Both entity types are stored in the same relational table, "pre-joined". If the relationship is not total both ways, there will be null padding on tuples that represent just one entity type. Any attributes of R are also moved to this table.
3. **Cross-reference approach**: A separate relation represents R; each tuple is a foreign key from S and a foreign key from T. Any attributes of R are also added to this relation. Here foreign keys should CASCADE.

Approach	Join cost	Null-storage cost
Foreign key	1	low to moderate
Merged relation	0	very high, unless both are total
Cross-reference	2	None

Step 4: Binary 1:N Relationship Types

Let the relationship be of the form [S]——^{N}<R>^{1}——[T]. The primary key of T is added as a foreign key in S. Attributes of R are moved to S. This is the foreign key approach. The merged relation approach is not possible for 1:N relationships. (Why?) The cross-reference approach might be used if the join cost is worth avoid null storage.

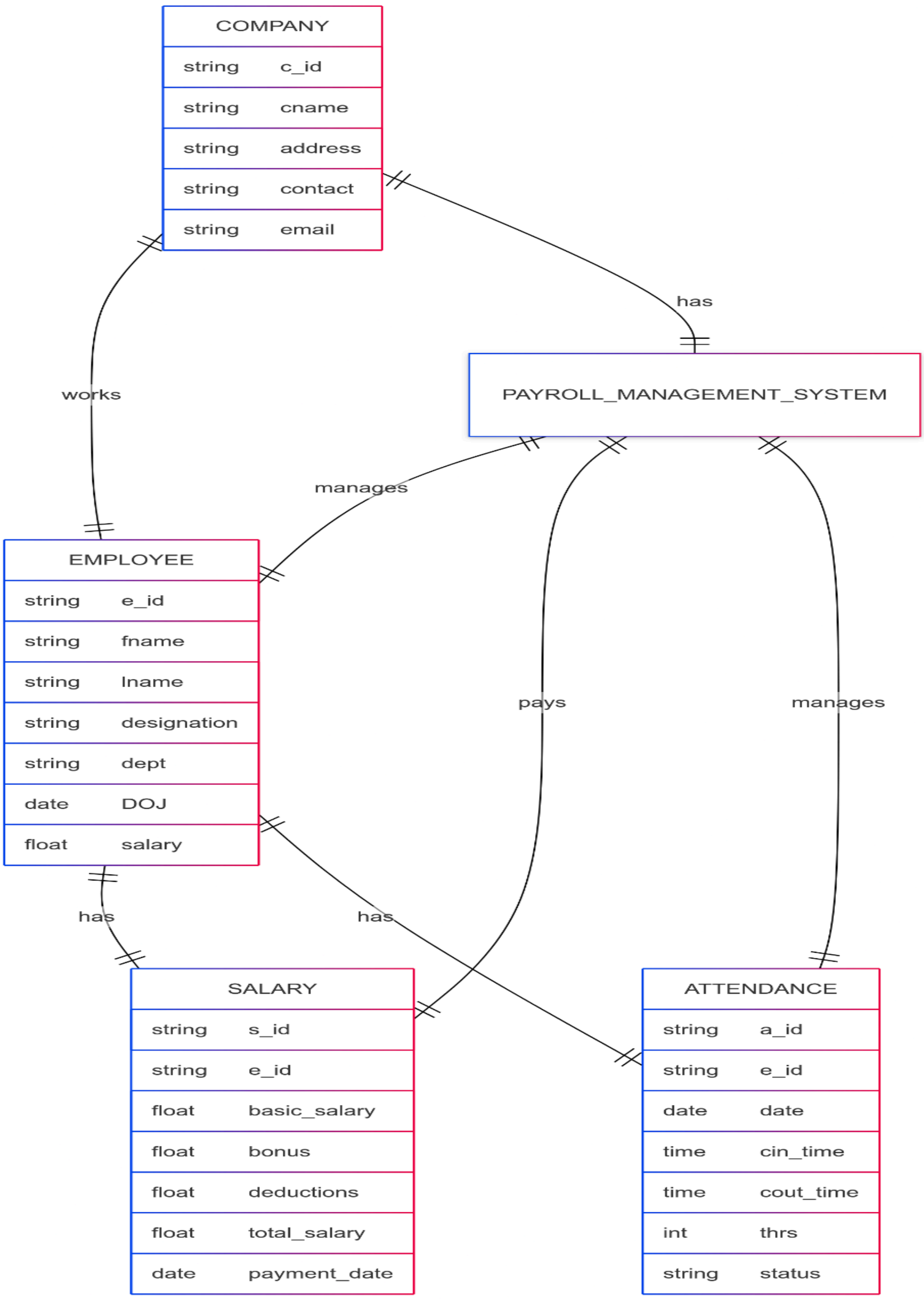
Step 5: Binary M:N Relationship Types
Here the cross-reference approach (also called a *relationship relation*) is the only possible way.

Step 6: Multivalued Attributes
Let an entity S have multivalued attribute A. Create a new relation R representing the attribute, with a foreign key into S added. The primary key of R is the combination of the foreign key and A. Once again this relation is dependent on an "owner relation" so its foreign key should CASCADE.

Step 7: Higher-Arity Relationship Types

Here again, use the cross-reference approach. For each n-ary relationship create a relation to represent it. Add a foreign key into each participating entity type. Also add any attributes of the relationship. The primary key of this relation is the combination of all foreign keys into participating entity types *that do not have a max cardinality of 1.*

Implementation:



Conclusion: The experiment successfully mapped ER/EER diagrams to relational schema models by applying structured mapping rules. This ensured a formal specification of the database schema, effectively capturing entity types, attributes, and relationships while minimizing redundancy and accommodating various participation constraints.

Experiment No.3
Create and populate database using Data Definition Language (DDL) and Apply Integrity Constraints for the specified system
Date of Performance:31/01/25
Date of Submission:07/02/25

Aim: Create and populate database using Data Definition Language (DDL) and apply Integrity Constraints for the specified system

Objective: DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Integrity constraints are used to ensure accuracy and consistency of data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity

Theory: DDL Commands

Create a table
Display the table description
Rename the table
Alter the table
Drop the table
Integrity constraints are:

1. PRIMARY KEY CONSTRAINTS
2. FOREIGN KEY CONSTRAINTS
3. NULL CONSTRAINTS
4. NOT NULL CONSTRAINTS
5. CHECK CONSTRAINTS
6. DEFAULT CONSTRAINTS

Implementation:

1. **Create Database , Table and Display Table description**

create database lib_DB;

-- drop database lib;

use lib_DB;

CREATE TABLE student(

pid int primary key,

s_name varchar(55) NOT NULL,

s_contact varchar(12),

s_dept varchar(30) DEFAULT "COMPS",

s_age int CHECK (s_age > 18)

);

desc student;

Field	Type	Null	Key	Default	Extra
pid	int	NO	PRI	NULL	
s_name	varchar(55)	NO		NULL	
s_contact	varchar(12)	YES		NULL	
s_dept	varchar(30)	YES		COMPS	
s_age	int	YES		NULL	

2. **Create Table Lib_Infra**

CREATE TABLE Lib_Infra(

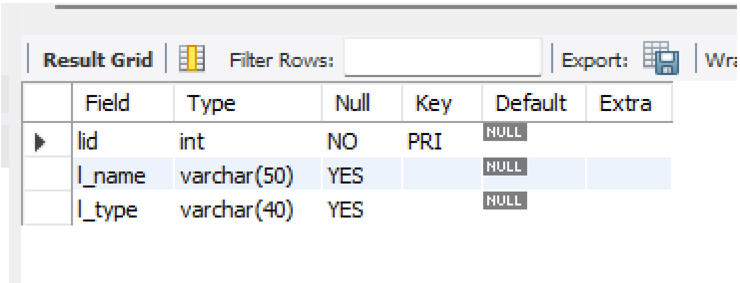
lid int PRIMARY KEY,

l_name varchar(50),

l_type varchar(40)

);

desc Lib_Infra;

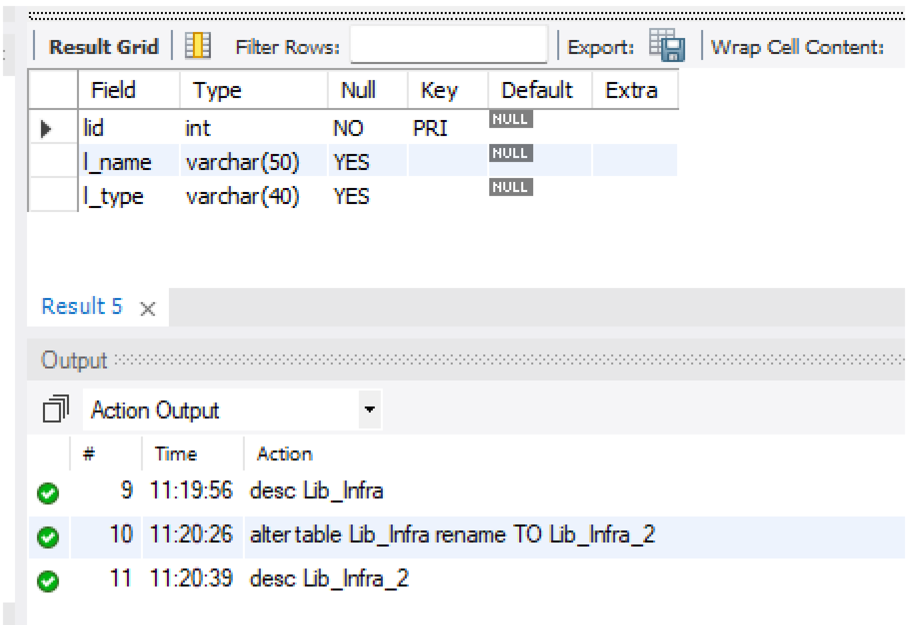


Field	Type	Null	Key	Default	Extra
lid	int	NO	PRI	NULL	
l_name	varchar(50)	YES		NULL	
l_type	varchar(40)	YES		NULL	

3. Rename Table

alter table Lib_Infra rename TO Lib_Infra_2;

desc Lib_Infra_2;



Field	Type	Null	Key	Default	Extra
lid	int	NO	PRI	NULL	
l_name	varchar(50)	YES		NULL	
l_type	varchar(40)	YES		NULL	

#	Time	Action
9	11:19:56	desc Lib_Infra
10	11:20:26	alter table Lib_Infra rename TO Lib_Infra_2
11	11:20:39	desc Lib_Infra_2

4. Alter table & Adding Constraint

```
alter table Lib_Infra_2 ADD COLUMN pid int;

ALTER TABLE Lib_Infra_2

ADD CONSTRAINT si_fk

FOREIGN KEY (pid)

REFERENCES student(pid);

Desc Lib_Infra_2;
```

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

Field	Type	Null	Key	Default	Extra
lid	int	NO	PRI	NULL	
l_name	varchar(50)	YES		NULL	
l_type	varchar(40)	YES		NULL	
pid	int	YES	MUL	NULL	

Result 6 ×

Conclusion: In this experiment, I learned to use DDL commands to create, modify, and manage database schemas efficiently. The implementation of integrity constraints such as PRIMARY KEY, FOREIGN KEY, NOT NULL, CHECK, and DEFAULT ensured data accuracy and consistency. Renaming and altering tables helped in modifying schema structures without data loss. Referential integrity was enforced using FOREIGN KEY constraints to maintain relationships between tables.

Experiment No.4
Apply DML Commands for your specified System.
Date of Performance:07/02/25
Date of Submission:14/02/25

Aim:- Apply DML Commands for your the specified System

Objective: The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

Theory:

DML:-DATA MANIPULATION LANGUAGE

Commands used in DML are

Insert Values
Retrieve all attributes
Update table
Delete table

Implementation:

1. **Insert Queries:**

```
CREATE DATABASE EMPLOYEES;
USE EMPLOYEES;

CREATE TABLE EMPLOYEES (
ID INT PRIMARY KEY,
NAME VARCHAR(70) NOT NULL,
DEPARTMENT VARCHAR(30) NOT NULL,
NOMINEE VARCHAR(50));

INSERT INTO EMPLOYEES (ID, NAME, DEPARTMENT)
VALUES
(1, 'JOHN DOE', 'HR');
INSERT INTO EMPLOYEES
VALUES
(2, "JASON SALDANHA", "IT", "KYLE"),
(3, "NISCA SHARMA", "FINANCE", "KAVYESH"),
(4, "JENNIFER D'SOUZA", "IT", "ALDRIDGE"),
(5, "MISHA K.", "HR","JOE");
```

```
1 •  SELECT * FROM employees;
```

Result Grid | Filter Rows: | Edit:

Id	Name	Department	Nominee
1	John Doe	HR	NULL
2	Jason Saldanha	IT	Kyle
3	Nisca Sharma	Finance	Kavyesh
4	Jennifer D'Souza	IT	Aldridge
5	Misha K.	HR	Joe
NULL	NULL	NULL	NULL

2. **Update Query:**

UPDATE employees SET Department = 'Finance' WHERE Id = 1;

```
1 •  UPDATE employees SET Department = 'Finance' WHERE Id = 1;
2 •  SELECT * FROM employees;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell

Id	Name	Department	Nominee
1	John Doe	Finance	NULL
2	Jason Saldanha	IT	Kyle
3	Nisca Sharma	Finance	Kavyesh
4	Jennifer D'Souza	IT	Aldridge
5	Misha K.	HR	Joe
NULL	NULL	NULL	NULL

3. **Delete Query:-**

DELETE FROM employees WHERE id = 1;

```
1 • DELETE FROM employees WHERE id = 1;
2 • SELECT * FROM employees;
```

Result Grid | Filter Rows: | Edit: | Export/Ir

Id	Name	Department	Nominee
2	Jason Saldanha	IT	Kyle
3	Nisca Sharma	Finance	Kavyesh
4	Jennifer D'Souza	IT	Aldridge
5	Misha K.	HR	Joe
NULL	NULL	NULL	NULL

Conclusion: The experiment successfully utilized DML commands to manipulate data within the database, demonstrating the ability to insert, retrieve, update, and delete data effectively, thereby achieving efficient data management in the specified system

Experiment No.5
Perform Simple queries, string manipulation operations and aggregate functions
Date of Performance:14/02/25
Date of Submission:21/02/25

Aim:- Perform Simple queries and aggregate functions.

Objective: Queries are a way of searching for and compiling data from one or more tables .aggregate functions are used to find Average, Maximum and minimum values, count values from given database

Theory:

Student (sid,sname,city,age, Marks)

Department(did, dname, sid)

Q1. Create a table student with given attributes.

Q2. Create a table department with given attributes.

Q3. Insert values into the respective tables & display them.

Q4. Update any row from student relation

Q5. Delete any row from the department table.

Q6. Give the minimum age of the student relation.

Q7. Find out the avg of marks of the student relation.

Q8. Give the total count of tuples in department relation group by did.

Implementation:

```
CREATE DATABASE students;
USE students;

-- Q1: Create a table student with given attributes
CREATE TABLE SE2_Students (
   sid INT PRIMARY KEY,
   sname VARCHAR(255),
   city VARCHAR(255),
   age INT,
   Marks DECIMAL(5,2)
);

-- Display table description
DESCRIBE SE2_Students;

-- Q2: Create a table department with given attributes
CREATE TABLE Dept_SE2 (
   did INT PRIMARY KEY,
   dname VARCHAR(50),
   sid INT,
   FOREIGN KEY (sid) REFERENCES SE2_Students(sid)
);

-- Display table description
DESCRIBE Dept_SE2;

-- Q3: Insert values into the respective tables & display them
INSERT INTO SE2_Students (sid, sname, city, age, Marks)
VALUES
(1, 'Aman Verma', 'Lucknow', 20, 85.5),
(2, 'Karan Patel', 'Indore', 22, 78.0),
(3, 'Ravi Kumar', 'Surat', 21, 80.2),
(4, 'Aakash Sharma', 'Delhi', 19, 88.7),
(5, 'Vikram Singh', 'Chennai', 20, 88.7);

SELECT * FROM SE2_Students;

INSERT INTO Dept_SE2 (did, dname, sid)
VALUES
(10, 'Computer', 1),
(11, 'IT', 2),
(12, 'ExTC', 3),
(13, 'CSCDS', 4),
(14, 'Civil', 5);
```

```
SELECT * FROM Dept_SE2;

-- Q4: Update any row from student relation
UPDATE SE2_Students
SET city = 'Pune', age = 21
WHERE sid = 1;

SELECT * FROM SE2_Students;

-- Q5: Delete any row from the department table
DELETE FROM Dept_SE2
WHERE did = 13;

SELECT * FROM Dept_SE2;

-- Q6: Give the minimum age of the student relation
SELECT MIN(age) AS min_age FROM SE2_Students;

-- Q7: Find out the average marks of the student relation
SELECT AVG(Marks) AS Average_Marks FROM SE2_Students;

-- Q8: Give the total count of tuples in department relation grouped by did
SELECT did, COUNT(*) AS tuple_count
FROM Dept_SE2
GROUP BY did;
```

Output:-

Q1 .

Field	Type	Null	Key	Default	Extra
sid	int	NO	PRI	NULL	
sname	varchar(255)	YES		NULL	
city	varchar(255)	YES		NULL	
age	int	YES		NULL	
Marks	decimal(5,2)	YES		NULL	

Q2.

	Field	Type	Null	Key	Default	Extra
▶	did	int	NO	PRI	NULL	
	dname	varchar(50)	YES		NULL	
	sid	int	YES	MUL	NULL	

Q3.

	sid	sname	city	age	Marks
▶	1	Aman Verma	Lucknow	20	85.50
	2	Karan Patel	Indore	22	78.00
	3	Ravi Kumar	Surat	21	80.20
	4	Aakash Sharma	Delhi	19	88.70
	5	Vikram Singh	Chennai	20	88.70
*	NULL	NULL	NULL	NULL	NULL

	did	dname	sid
▶	10	Computer	1
	11	IT	2
	12	ExTC	3
	13	CSCDS	4
	14	Civil	5
*	NULL	NULL	NULL

Q4.

	sid	sname	city	age	Marks
▶	1	Aman Verma	Pune	21	85.50
	2	Karan Patel	Indore	22	78.00
	3	Ravi Kumar	Surat	21	80.20
	4	Aakash Sharma	Delhi	19	88.70
	5	Vikram Singh	Chennai	20	88.70
*	NULL	NULL	NULL	NULL	NULL

Q5.

	did	dname	sid
▶	10	Computer	1
	11	IT	2
	12	ExTC	3
	14	Civil	5
*	NULL	NULL	NULL

Q6.

	min_age
▶	19

Q7.

	Average_Marks
▶	84.220000

Q8.

	did	tuple_count
▶	10	1
	11	1
	12	1
	14	1

Conclusion: The experiment successfully demonstrated the creation and management of database tables and effectively applied SQL queries to manipulate and retrieve data. Using aggregate functions like MIN, AVG, and COUNT, key insights were extracted from the data, showcasing the ability to handle relational databases efficiently and achieve the stated objectives.

Experiment No.6
Implement SET operators and Datetime functions.
Date of Performance:21/02/25
Date of Submission:07/03/25

Aim: **Implement SET operators and Datetime functions**

Objective: SET operators in SQL are used to combine results from two queries, such as UNION, INTERSECT, and MINUS, while Datetime functions are used to manipulate and extract parts of date and time values, like NOW(), DATEADD(), and DATEDIFF()

Theory:
SET OPERATORS:

1. UNION / UNION ALL:
- UNION: Returns result from both queries after eliminating duplications.

e.g.:
```
SELECT employee_id, job_id
FROM employees
UNION
SELECT employee_id, job_id
FROM job_history;
```

- UNION ALL: returns results from both queries, including all duplications.

e.g.:
```
SELECT employee_id, job_id, department_id
FROM employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM job_history
ORDER BY employee_id;
```

2. INTERSECT:

e.g.:
```
SELECT employee_id, job_id
FROM employees
INTERSECT
SELECT employee_id, job_id
FROM job_history;
```

3. MINUS:

e.g.:
```
SELECT employee_id, job_id
FROM employees
MINUS
SELECT employee_id, job_id
FROM job_history;
```

Datetime functions:

1. CURDATE()
Returns the current date (without time).
Example:
SELECT CURDATE();

2. CURTIME()
Returns the current time (without the date).
SELECT CURTIME();

3. NOW()
Returns the current date and time.
SELECT NOW();

4. DATE()
Extracts the date part of a datetime value (removes the time).
Example:
SELECT DATE(NOW());

5. TIME()
Extracts the time part of a datetime value (removes the date).
Example:
SELECT TIME(NOW());

6. YEAR()
Extracts the year from a date or datetime value.
Example:
SELECT YEAR(NOW());

7. MONTH()
Extracts the month from a date or datetime value.
Example:
SELECT MONTH(NOW());

8. DAY()
Extracts the day of the month from a date or datetime value.
Example:
SELECT DAY(NOW());

9. DATE_ADD()
Adds a specified time interval to a date or datetime.
Example:
SELECT DATE_ADD(NOW(), INTERVAL 5 DAY);

10. DATE_SUB()
Subtracts a specified time interval from a date or datetime.
Example:
SELECT DATE_SUB(NOW(), INTERVAL 7 DAY);

11. DATEDIFF()
Returns the difference in days between two dates.
Example:
SELECT DATEDIFF('2025-02-01', '2025-01-27');

12. TIMEDIFF()
Returns the difference in time between two time values.
Example:
SELECT TIMEDIFF('15:00:00', '14:30:00');

13. STR_TO_DATE()
Converts a string into a date, based on a specified format.
Example:
SELECT STR_TO_DATE('2025-01-27', '%Y-%m-%d');

14. DATE_FORMAT()
Formats a date or datetime value according to a specified format.
Example:
SELECT DATE_FORMAT(NOW(), '%Y-%m-%d %H:%i:%s');

15. UNIX_TIMESTAMP()
Returns the current date and time as a Unix timestamp (seconds since 1970-01-01).
Example:
SELECT UNIX_TIMESTAMP();

16. FROM_UNIXTIME()
Converts a Unix timestamp to a datetime value.
Example:
SELECT FROM_UNIXTIME(1706359800);

Implementation:

```
CREATE DATABASE set_ops;
USE set_ops;

CREATE TABLE Student (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50),
    department_id INT,
    admission_date DATE
);

CREATE TABLE Department (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50)
);

INSERT INTO Student (student_id, student_name, department_id, admission_date) VALUES
(1, 'Alice', 101, '2023-09-10'),
(2, 'Bob', 102, '2022-08-15'),
(3, 'Charlie', 103, '2021-07-20'),
(4, 'David', 101, '2023-06-05'),
(5, 'Eve', 104, '2024-01-15'),
(6, 'Frank', 105, CURDATE());

INSERT INTO Department (department_id, department_name) VALUES
(101, 'Computer Science'),
(102, 'Mechanical Engineering'),
(104, 'Electrical Engineering'),
(105, 'Civil Engineering');
```

```
SELECT department_id FROM Student
UNION
SELECT department_id FROM Department;

SELECT department_id FROM Student
UNION ALL
SELECT department_id FROM Department;

SELECT s.department_id
FROM Student AS s
WHERE s.department_id IN (
   SELECT d.department_id
   FROM Department AS d
);

SELECT department_id
FROM Student
WHERE department_id NOT IN (
   SELECT department_id
   FROM Department
);

SELECT CURRENT_DATE;

SELECT * FROM Student
WHERE admission_date >= NOW() - INTERVAL 1 YEAR;

SELECT student_name, DATE_FORMAT(admission_date, '%Y-%m-%d')
FROM Student;

SELECT student_name, EXTRACT(YEAR FROM admission_date) AS admission_year,
```

EXTRACT(MONTH FROM admission_date) AS admission_month

FROM Student;

Output:

1. SELECT department_id FROM Student

   UNION

   SELECT department_id FROM Department;

department_id
101
102
103
104
105

2. SELECT department_id FROM Student

UNION ALL

SELECT department_id FROM Department;

department_id
101
102
103
101
104
105
101
102
104
105

3. SELECT s.department_id

```
FROM Student AS s
WHERE s.department_id IN (
    SELECT d.department_id
    FROM Department AS d
);
```

department_id
101
102
101
104
105

4. SELECT department_id

```
FROM Student
WHERE department_id NOT IN (
    SELECT department_id
    FROM Department
);
```

department_id
103

5. SELECT CURRENT_DATE;

CURRENT_DATE
2025-03-27

6. SELECT * FROM Student

WHERE admission_date >= NOW() - INTERVAL 1 YEAR;.

	student_id	student_name	department_id	admission_date
▶	6	Frank	105	2025-03-27
*	NULL	NULL	NULL	NULL

7. SELECT student_name, DATE_FORMAT(admission_date, '%Y-%m-%d') FROM Student;

	student_name	DATE_FORMAT(admission_date, '%Y-%m-%d')
▶	Alice	2023-09-10
	Bob	2022-08-15
	Charlie	2021-07-20
	David	2023-06-05
	Eve	2024-01-15
	Frank	2025-03-27

8. SELECT student_name, EXTRACT(YEAR FROM admission_date) AS admission_year,

EXTRACT(MONTH FROM admission_date) AS admission_month
FROM Student;

	student_name	admission_year	admission_month
▶	Alice	2023	9
	Bob	2022	8
	Charlie	2021	7
	David	2023	6
	Eve	2024	1
	Frank	2025	3

Conclusion: The experiment successfully demonstrated the implementation of SQL SET operators and Datetime functions, showcasing their ability to manipulate and combine query results as well as extract and format date and time values. These functionalities enhance data retrieval and analysis capabilities, ensuring efficient handling of relational databases.

Experiment No.7
Nested queries and Complex queries
Date of Performance:07/03/25
Date of Submission:21/03/25

Aim: Nested queries and Complex queries

Objective: In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query

Theory:

Sample table: Salesman

salesman_id name city commission

Sample table: Orders

ord_no purch_amt ord_date customer_id salesman_id

Questions

1. Write a query to display all the orders from the orders table issued by the salesman 'Paul Adam'.

2. Write a query to display all the orders for the salesman who belongs to the city London.

3. Write a query to find all the orders issued against the salesman who may works for customer whose id is 3007

4. Write a query to display all the orders which values are greater than the average order value for 10th October 2012

5. Write a query to find all orders attributed to a salesman in New york.

6. Write a query to display the commission of all the salesmen servicing customers in Paris

Implementation:

CREATE DATABASE SalesDB;
USE SalesDB;

CREATE TABLE Salesman (

```
    salesman_id INT PRIMARY KEY,
    name VARCHAR(255),
    city VARCHAR(255),
    commission DECIMAL(10, 2)
);

CREATE TABLE Orders (
    ord_no INT PRIMARY KEY,
    purch_amt DECIMAL(10, 2),
    ord_date DATE,
    customer_id INT,
    salesman_id INT
);

CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(255),
    customer_city VARCHAR(255),
    ord_no INT
);

INSERT INTO Salesman (salesman_id, name, city, commission) VALUES
(1, 'Paul Adam', 'London', 2500.50),
(2, 'John Doe', 'New York', 1800.75),
(3, 'Jane Smith', 'London', 2000.00),
(4, 'Chris Green', 'Paris', 2200.00),
(5, 'Alice Brown', 'New York', 1900.25),
(6, 'David White', 'New York', 2100.00);

INSERT INTO Orders (ord_no, purch_amt, ord_date, customer_id, salesman_id) VALUES
(101, 500.00, '2012-10-10', 1001, 1),
(102, 800.00, '2012-10-10', 1002, 4),
(103, 1200.00, '2012-10-11', 1003, 2),
(104, 450.00, '2012-10-10', 1004, 1),
(105, 700.00, '2012-10-12', 1005, 4),
(106, 1000.00, '2012-10-10', 1006, 5),
(107, 1200.00, '2012-10-11', 1007, 6),
(108, 950.00, '2012-10-10', 3007, 1),
(109, 550.00, '2012-10-11', 1008, 3);
```

```
INSERT INTO Customers (customer_id, customer_name, customer_city, ord_no) VALUES
(1001, 'Customer A', 'London', 101),
(1002, 'Customer B', 'Paris', 102),
(1003, 'Customer C', 'New York', 103),
(1004, 'Customer D', 'London', 104),
(1005, 'Customer E', 'Paris', 105),
(1006, 'Customer F', 'New York', 106),
(1007, 'Customer G', 'Paris', 108),
(1008, 'Customer H', 'Paris', 109),
(3007, 'Customer X', 'New York', 108);

ALTER TABLE Customers
ADD CONSTRAINT fk_ord_no FOREIGN KEY (ord_no) REFERENCES Orders(ord_no);

SELECT o.*
FROM Orders o
JOIN Salesman s ON o.salesman_id = s.salesman_id
WHERE s.name = 'Paul Adam';

SELECT o.*
FROM Orders o
JOIN Salesman s ON o.salesman_id = s.salesman_id
WHERE s.city = 'London';

SELECT o.*
FROM Orders o
JOIN Salesman s ON o.salesman_id = s.salesman_id
JOIN Customers c ON o.customer_id = c.customer_id
WHERE c.customer_id = 3007;

SELECT *
FROM Orders
WHERE purch_amt > (
   SELECT AVG(purch_amt)
   FROM Orders
   WHERE ord_date = '2012-10-10'
);

SELECT o.*
FROM Orders o
```

JOIN Salesman s ON o.salesman_id = s.salesman_id
WHERE s.city = 'New York';

SELECT DISTINCT s.salesman_id, s.name, s.commission
FROM Salesman s
JOIN Orders o ON s.salesman_id = o.salesman_id
JOIN Customers c ON o.customer_id = c.customer_id
WHERE c.customer_city = 'Paris';

Output:

1. SELECT o.*
   FROM Orders o
   JOIN Salesman s ON o.salesman_id = s.salesman_id
   WHERE s.name = 'Paul Adam';

	ord_no	purch_amt	ord_date	customer_id	salesman_id
▶	101	500.00	2012-10-10	1001	1
	104	450.00	2012-10-10	1004	1
	108	950.00	2012-10-10	3007	1

2. SELECT o.*

   FROM Orders o
   JOIN Salesman s ON o.salesman_id = s.salesman_id
   WHERE s.city = 'London';

	ord_no	purch_amt	ord_date	customer_id	salesman_id
▶	101	500.00	2012-10-10	1001	1
	104	450.00	2012-10-10	1004	1
	108	950.00	2012-10-10	3007	1
	109	550.00	2012-10-11	1008	3

3. SELECT o.*
   FROM Orders o
   JOIN Salesman s ON o.salesman_id = s.salesman_id
   JOIN Customers c ON o.customer_id = c.customer_id

WHERE c.customer_id = 3007;

	ord_no	purch_amt	ord_date	customer_id	salesman_id
▶	108	950.00	2012-10-10	3007	1

4. SELECT *
FROM Orders
WHERE purch_amt > (
SELECT AVG(purch_amt)
FROM Orders
WHERE ord_date = '2012-10-10'
);

	ord_no	purch_amt	ord_date	customer_id	salesman_id
▶	102	800.00	2012-10-10	1002	4
	103	1200.00	2012-10-11	1003	2
	106	1000.00	2012-10-10	1006	5
	107	1200.00	2012-10-11	1007	6
	108	950.00	2012-10-10	3007	1
*	NULL	NULL	NULL	NULL	NULL

5. SELECT o.*
FROM Orders o
JOIN Salesman s ON o.salesman_id = s.salesman_id
WHERE s.city = 'New York';

	ord_no	purch_amt	ord_date	customer_id	salesman_id
▶	103	1200.00	2012-10-11	1003	2
	106	1000.00	2012-10-10	1006	5
	107	1200.00	2012-10-11	1007	6

6. SELECT DISTINCT s.salesman_id, s.name, s.commission
FROM Salesman s
JOIN Orders o ON s.salesman_id = o.salesman_id
JOIN Customers c ON o.customer_id = c.customer_id
WHERE c.customer_city = 'Paris';

salesman_id	name	commission
4	Chris Green	2200.00
6	David White	2100.00
3	Jane Smith	2000.00

Conclusion: The experiment successfully demonstrated the use of nested and complex queries in SQL to retrieve meaningful data from relational databases. By leveraging inner and outer queries, critical insights such as orders based on specific criteria, commission details, and comparison of order values were efficiently extracted. This highlights the versatility and power of SQL in handling intricate database operations.

Experiment No.8
Procedures and Functions
Date of Performance:21/3/25
Date of Submission:27/3/25

Aim: To implement Functions and procedure.

Objective: The function must return a value but in Stored Procedure it is optional. Even a procedure can return zero or n values. Functions can have only input parameters for it whereas Procedures can have input or output parameters

Theory:

Procedure:

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

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

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.

The AS keyword is used instead of the IS keyword for creating a standalone procedure

Creating a Function

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

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
```

{IS | AS}
BEGIN
< function_body >
END [function_name];

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the 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.

Implementation:

```
mysql> CREATE DATABASE SE2;
Query OK, 1 row affected (0.01 sec)

mysql>
mysql> USE SE2;
Database changed
mysql>
mysql> CREATE TABLE Employees_SE2 (
    ->     EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
    ->     EmployeeName VARCHAR(100) NOT NULL,
    ->     DepartmentID INT NOT NULL,
    ->     Salary DECIMAL(10,2) NOT NULL
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO Employees_SE2 (EmployeeName, DepartmentID, Salary) VALUES
    -> ('Alice', 1, 5000.00),
    -> ('Bob', 2, 6000.00),
    -> ('Charlie', 1, 5500.00),
    -> ('David', 3, 7000.00),
    -> ('Emma', 2, 6200.00);
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE GetEmployeesByDept(IN dept_id INT)
    -> BEGIN
    ->     SELECT * FROM Employees_SE2 WHERE DepartmentID = dept_id;
    -> END
    -> //
Query OK, 0 rows affected (0.01 sec)

mysql> DELIMITER ;
mysql> CALL GetEmployeesByDept(2);
+------------+--------------+--------------+---------+
| EmployeeID | EmployeeName | DepartmentID | Salary  |
+------------+--------------+--------------+---------+
|          2 | Bob          |            2 | 6000.00 |
|          5 | Emma         |            2 | 6200.00 |
+------------+--------------+--------------+---------+
2 rows in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

```
mysql> DELIMITER //
mysql> CREATE FUNCTION GetTotalSalaryByDept(dept_id INT) RETURNS DECIMAL(10,2)
    -> DETERMINISTIC
    -> BEGIN
    ->     DECLARE total_salary DECIMAL(10,2);
    ->
    ->     SELECT SUM(Salary) INTO total_salary
    ->     FROM Employees_SE2
    ->     WHERE DepartmentID = dept_id;
    ->        RETURN total_salary;
    -> END //
Query OK, 0 rows affected (0.01 sec)

mysql> DELIMITER ;
mysql> SELECT GetTotalSalaryByDept(2) AS TotalSalary;
+-------------+
| TotalSalary |
+-------------+
|    12200.00 |
+-------------+
1 row in set (0.00 sec)
```

Conclusion: The experiment successfully implemented SQL functions and procedures, showcasing their distinction and versatility in database operations. Functions demonstrated the ability to return values and handle input parameters, while procedures highlighted flexibility by utilizing input, output, and mixed parameters, enabling efficient execution of complex tasks and data manipulation in relational databases.

Experiment No.9
Views and Triggers
Date of Performance:27/3/25
Date of Submission:28/3/25

Aim: Views and Triggers

Objective: Views can join and simplify multiple **tables** into a single virtual table A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should also be created in the tables of the taxes, vacations and salaries.

Theory:

VIEWS

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

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

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

Creating Views

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW** syntax is as follows −

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

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

TIGGERS:

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

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

Triggers 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 −

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

Creating Triggers

The syntax for creating a trigger is −

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

```
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Where,

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

Implementation:

```
mysql> USE EXP9;
Database changed
mysql>
mysql> CREATE TABLE Departments (
    ->     DepartmentID INT PRIMARY KEY,
    ->     DepartmentName VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql>
mysql> INSERT INTO Departments (DepartmentID, DepartmentName) VALUES
    -> (1, 'HR'),
    -> (2, 'IT'),
    -> (3, 'Sales');
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql>
mysql> CREATE TABLE Employees_SE (
    ->     EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
    ->     EmployeeName VARCHAR(100),
    ->     DepartmentID INT,
    ->     Salary DECIMAL(10,2),
    ->     FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
    -> );
Query OK, 0 rows affected (0.02 sec)

mysql>
mysql> INSERT INTO Employees_SE (EmployeeName, DepartmentID, Salary) VALUES
    -> ('Alice', 1, 5000.00),
    -> ('Bob', 2, 6000.00),
    -> ('Charlie', 1, 5500.00),
    -> ('David', 3, 7000.00),
    -> ('Emma', 2, 6200.00);
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
mysql> CREATE VIEW HighSalaryEmployees AS
    -> SELECT EmployeeID, EmployeeName, Salary
    -> FROM Employees_SE
    -> WHERE Salary > 6000;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM HighSalaryEmployees;
+------------+--------------+---------+
| EmployeeID | EmployeeName | Salary  |
+------------+--------------+---------+
|          4 | David        | 7000.00 |
|          5 | Emma         | 6200.00 |
+------------+--------------+---------+
2 rows in set (0.00 sec)

mysql>
mysql> CREATE TABLE SalaryChanges (
    ->     ChangeID INT PRIMARY KEY AUTO_INCREMENT,
    ->     EmployeeID INT,
    ->     OldSalary DECIMAL(10,2),
    ->     NewSalary DECIMAL(10,2),
    ->     ChangeDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    -> );
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> DELIMITER //
mysql> CREATE TRIGGER AfterSalaryUpdate
    -> AFTER UPDATE ON Employees_SE
    -> FOR EACH ROW
    -> BEGIN
    ->     INSERT INTO SalaryChanges (EmployeeID, OldSalary, NewSalary)
    ->     VALUES (OLD.EmployeeID, OLD.Salary, NEW.Salary);
    -> END //
Query OK, 0 rows affected (0.02 sec)

mysql> DELIMITER ;
```

```
mysql> UPDATE Employees_SE
    -> SET Salary = 7500
    -> WHERE EmployeeID = 4;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> Select * from SalaryChanges;
+----------+------------+-----------+-----------+---------------------+
| ChangeID | EmployeeID | OldSalary | NewSalary | ChangeDate          |
+----------+------------+-----------+-----------+---------------------+
|        1 |          4 |   7000.00 |   7500.00 | 2025-03-28 12:33:03 |
+----------+------------+-----------+-----------+---------------------+
1 row in set (0.00 sec)
```

Conclusion: The experiment successfully demonstrated the use of SQL views and triggers to enhance database functionality. Views allowed efficient data structuring and simplified access to multiple tables, enabling intuitive data representation and secure access controls. Triggers provided automated responses to database events, ensuring integrity, synchronization, and security while supporting advanced features like derived column generation and transaction validation. These implementations significantly contribute to the efficient management and operation of relational databases.

Experiment No.10
Mini project- Course project report using database connectivity
Date of Performance:
Date of Submission:

Aim: Mini project- Creating a Two-tier client-server database applications using database connectivity

Objective: Java Database Connectivity (JDBC)/ODBC is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is a Java-based data access technology used for Java database connectivity

Implementation:

Prepare Report and show demonstration

Conclusion: Comment on the Prototype of given application using database connectivity