Department of Computer Science & Engineering

LAB MANUAL

Subject Name : - DBMS Lab

Subject Code : - CS220

Branch : - CSE

Year : - 2nd

Semester : - 4th

INTEGRAL UNIVERSITY LUCKNOW

Dasauli, Kursi Road, PO Basha-226026

VISION

To produce highly skilled personnel who are empowered enough to transform the society by their education, research and innovations.

MISSION

- To offer diverse academic programs at undergraduate, postgraduate, and doctorate levels that are . in line with the current trends in Computer Science and Engineering.
- To provide the state-of-the-art infrastructure for teaching, learning and research.
- To facilitate collaborations with other universities, industry and research labs.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1	To produce graduates who have strong foundation of knowledge and skills in the field of computer science and engineering.
PEO2	To produce graduates who are employable in industries/public sector/research organizations or work as an entrepreneur.
PEO3	To produce graduates who can provide solutions to challenging problems in their profession by applying computer engineering theory and practices.
PEO4	To produce graduates who can provide leadership and are effective in multidisciplinary environment

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1	Ability to understand the principles and working of computer systems. Students have a sound knowledge about the hardware and software aspects of computer systems.
PSO2	Ability to design and develop computer programs and understand the structure and development methodologies of software systems.
PSO3	Ability to apply their skills in the field of algorithms, networking, web design, cloud computing and data analytics.
PSO4	Ability to apply knowledge to provide innovative novel solutions to existing problem and identify research gaps.

Course Objectives

1. To explain basic database concepts, applications, data models, schemas, and instances.
2. To describe the basics of SQL and construct queries using SQL.
3. To demonstrate the use of constraints and relational algebra operations.
4. To facilitate students in developing solutions for database applications.
5. To describe the concepts of ER-Model and normalization in databases.

Course Outcomes

After undergoing this laboratory module, the participant should be able to:

1. Able to understand the basics of SQL and construct queries using SQL in database creation.
2. Ability to formulate queries for DML/DDL/DCL commands.
3. Able to use aggregate functions, GROUP BY, HAVING, ANY, ALL, IN, EXISTS, NOT EXISTS, UNION, INTERSECT, MINUS.
4. Understand various advanced query execution such as relational constraints, joins, nested queries, VIEWS creation and dropping.
5. Able to design and implement a relational database system by taking up case studies

CERTIFICATE

	Name of Teacher	Signature
Prepared By	Nida Khan	
Date of Preparation	With Effect From 11th July 2024	
Review On Date	15th July 2024	
Verified By	Dr. Roshan Jahan	

Integral University, Lucknow

Effective from Session: 2024-25							
Course Code	CS220	**Title of the Course**	DBMS Lab	**L**	**T**	**P**	**C**
Year	II	**Semester**	IV	**0**	**0**	**2**	**1**
Pre-Requisite	None	**Co-requisite**	None				
Course Objectives	• To explain basic database concepts, applications, data models, schemas, and instances. • To describe the basics of SQL and construct queries using SQL. • To demonstrate the use of constraints and relational algebra operations. • To facilitate students in developing solutions for database applications. • To describe the concepts of ER-Model and normalization in databases						

Course Outcomes	
CO1	Able to understand the basics of SQL and construct queries using SQL in database creation.
CO2	Ability to formulate queries for DML/DDL/DCL commands.
CO3	Able to use aggregate functions, GROUP BY, HAVING, ANY, ALL, IN, EXISTS, NOT EXISTS, UNION, INTERSECT, MINUS.
CO4	Understand various advanced query execution such as relational constraints, joins, nested queries, VIEWS creation and dropping.
CO5	Able to design and implement a relational database system by taking up case studies.

S. No.	**List of Experiments**	**Contact Hrs.**	**Mapped CO**
1	Overview of using SQL, data types in SQL, concept of DDL, DML & DCL commands, creating tables (along with primary and foreign keys), altering tables, and dropping tables.	2	1
2	Practicing DML commands- Insert, Select, Update, Delete.	2	1
3	Write queries using Logical Operators (=, <, > etc).	2	2
4	Write queries using SQL operators (BETWEEN…AND, IN (list), LIKE, ISNULL and along with negation expressions).	2	2
5	Write queries using COUNT, SUM, AVG, MAX, MIN, GROUP BY, HAVING.	2	3
6	Write queries using ANY, ALL, IN, EXISTS, NOT EXISTS, UNION, INTERSECT, MINUS, CONSTRAINTS etc.	2	3
7	Write queries for extracting data from more than one table (Equi-Join, Non-Equi Join, Inner Join, Outer Join).	2	4
8	Write queries for Sub queries, Nested queries, VIEWS Creation and Dropping.	2	4
9	CASE STUDY: Student should decide on a case study and formulate the problem statement, Database Design using ER Model (Identifying entities, attributes, keys and relationships between entities, cardinalities, generalization, specialization etc.) **Note: Student is required to submit a document by drawing ER-Diagram to the Lab teacher.**	2	5
10	Converting ER Model to Relational Model (Represent entities and relationships in Tabular form, represent attributes as columns, identifying keys), Create tables using SQL. **Note: Student is required to submit a document showing the database tables created from ER Model.**	2	5
11	Normalization -To remove the redundancies and anomalies in the above relational tables, Normalize up to Third Normal Form.	2	5

PO-PSO CO	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**	**PSO4**
CO1	3	2	3	2	2						2	3	1	2		1
CO2	2	3	3	2	3						2	1	3	1		1
CO3	3	3	2	3	3						1	2	1	1		1
CO4	3	3	3	2	3	1					3	1	2	1	2	1
CO5	3	2	3	1	3	1			1	1	2	1	1	3	2	1

1-Low Correlation; 2- Moderate Correlation; 3- Substantial Correlation

List of Lab Exercises

Week	Experiment/ Objective No.	Name of the Experiments/Title	Page No.
Week-1		Overview of using SQL, data types in SQL, concept of DDL, DML & DCL commands.	6-8
	1	Creating tables (along with primary and foreign keys), altering tables, and dropping tables.	9-11
Week-2	2	Practicing DML commands- Insert, Select, Update, Delete.	12-13
Week-3	3	Write Queries using Logical Operators (<,>,=, etc.)	14-17
Week-4	4	Write queries using SQL operators (BETWEEN…AND, IN (list), LIKE, ISNULL &along with Negation expressions.)	18-21
Week-5	5	Write queries using COUNT, SUM, AVG, MAX, MIN, GROUP BY, HAVING.	22-24
Week-6	6	Write queries using ANY, ALL, IN, EXISTS, NOT EXISTS, UNION, INTERSECT, MINUS, CONSTRAINTS etc.	25-28
Week-7	7	Write queries for extracting data from more than one table (Equi-Join, Non-Equi Join, Inner Join, Outer Join).	29-32
Week-8	8	Write queries for Sub queries, Nested queries, VIEWS Creation and Dropping.	33-37
Week-9	9	CASE STUDY: Student should decide on a case study and formulate the problem statement, Database Design using ER Model (Identifying entities, attributes, keys and relationships between entities, cardinalities, generalization, specialization etc.) Note: Student is required to submit a document by	38-39
Week-10	10	Converting ER Model to Relational Model (Represent entities and relationships in Tabular form, represent attributes as columns, identifying keys), Create tables using SQL. Note: Student is required to submit a document showing the database tables created from ER Model.	40-42
Week-11	11	Normalization -To remove the redundancies and anomalies in the above relational tables, Normalize up to Third Normal Form.	43-45

Overview of using SQL

SQL (Structured Query Language) is a standardized language designed for managing and manipulating relational databases. It enables users to interact with the database to perform various operations, such as retrieving data, inserting records, updating existing records, and deleting records. Here's an overview of SQL's key aspects and common usage:

1. Basic Concepts

- **Database**: A structured collection of data stored electronically.
- **Table**: A collection of rows (records) and columns (fields) within a database.
- **Schema**: The structure that defines the organization of data, including tables, fields, and relationships.
- **SQL Statements**: Commands used to interact with the database.

Uses of SQL

SQL is used for interacting with databases. These interactions include:

Data definition: It is used to define the structure and organization of the stored data and the relationships among the stored data items.

Data retrieval: SQL can also be used for data retrieval.

Data manipulation: If the user wants to add new data, remove data, or modifying in existing data then SQL provides this facility also.

Access control: SQL can be used to restrict a user's ability to retrieve, add, and modify data, protecting stored data against unauthorized access.

Data sharing: SQL is used to coordinate data sharing by concurrent users, ensuring that changes made by one user do not inadvertently wipe out changes made at nearly the same time by another user.

Key Features of SQL

- **Declarative Nature**: Users specify what they want, not how to achieve it.
- **Standardization**: Widely adopted across various database management systems (DBMS), though there are vendor-specific extensions.
- **Relational Operations**: SQL works well with relational data models.

SQL in Practice

- Used for **data analysis**, **application development**, and **database administration**.
- Popular database systems include MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, and SQLite.

SQL Best Practices

- Use prepared statements to prevent SQL injection.

- Normalize databases to reduce redundancy.
- Index frequently used columns for faster querying.
- Regularly back up databases to prevent data loss.

Data Types in SQL

SQL provides a variety of **data types** to define the kind of data a column can store. These data types ensure data integrity and help the database efficiently manage storage. Below is a categorized overview of SQL data types:

1. Numeric Data Types

Used for storing numbers, including integers and floating-point numbers.

a. Integer Types

- **TINYINT**: Small integer, typically 1 byte.
- **SMALLINT**: Small integer, typically 2 bytes.
- **MEDIUMINT**: Medium-sized integer, typically 3 bytes.
- **INT / INTEGER**: Standard integer, typically 4 bytes.
- **BIGINT**: Large integer, typically 8 bytes.

b. Floating-Point Types

- **FLOAT**: Single-precision floating-point, size depends on precision.
- **DOUBLE / DOUBLE PRECISION**: Double-precision floating-point.
- **DECIMAL / NUMERIC**: Fixed-point numbers with exact precision. Format: DECIMAL(p, s) (precision p, scale s).

2. Character/String Data Types

Used for storing text or string data.

a. Fixed-Length Strings

- **CHAR(n)**: Fixed-length string of size n (1 to 255 characters).

b. Variable-Length Strings

- **VARCHAR(n)**: Variable-length string with a maximum size of n.

c. Text Data

- **TINYTEXT**: Maximum size of 255 characters.
- **TEXT**: Maximum size of 65,535 characters.
- **MEDIUMTEXT**: Maximum size of 16,777,215 characters.
- **LONGTEXT**: Maximum size of 4,294,967,295 characters.

d. Binary Strings

- **BINARY(n)**: Fixed-length binary data.
- **VARBINARY(n)**: Variable-length binary data.
- **BLOB** (Binary Large Object): Used for storing binary data (e.g., images, files).

3. Date and Time Data Types

Used for storing dates, times, and timestamps.

- **DATE**: Stores date values (YYYY-MM-DD).
- **TIME**: Stores time values (HH:MM:SS).
- **DATETIME**: Stores date and time values (YYYY-MM-DD HH:MM:SS).
- **TIMESTAMP**: Stores date and time values with time zone (YYYY-MM-DD HH:MM:SS).
- **YEAR**: Stores a 4-digit year (YYYY).

4. Boolean Data Type

- **BOOLEAN**: Stores TRUE or FALSE values. Often implemented as a TINYINT (0 = FALSE, 1 = TRUE).

Concept of DDL, DML & DCL commands,

a. Data Definition Language (DDL)

- **Purpose**: Define or modify database structures.
- **Examples**:

```
CREATE TABLE table_name (column1 datatype, column2 datatype);
ALTER TABLE table_name ADD column_name datatype;
DROP TABLE table_name;
```

b. Data Manipulation Language (DML)

- **Purpose**: Manage data within tables.
- **Examples**:

```
INSERT INTO table_name (column1, column2) VALUES (value1, value2);
UPDATE table_name SET column1 = value1 WHERE condition;
DELETE FROM table_name WHERE condition;
```

c. Data Control Language (DCL)

- **Purpose**: Control access to the database.
- **Examples**:

```
GRANT SELECT, INSERT ON table_name TO user;
REVOKE SELECT ON table_name FROM user;
```

EXPERIMENT-1

Objective: Creating tables (along with primary and foreign keys), altering tables, and dropping tables.

Creating Tables

Tables are the fundamental structure in SQL databases for storing data. While creating a table, you can define its columns and specify primary and foreign keys.

Syntax for Creating Tables

```
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...,
    PRIMARY KEY (column_name),
    FOREIGN KEY (column_name) REFERENCES other_table (column_name)
);
```

Example: Creating Tables with Primary and Foreign Keys

Create the Departments table

```
CREATE TABLE Departments (
    dept_id INT PRIMARY KEY, -- Primary Key
    dept_name VARCHAR(100) NOT NULL
);
```

Create the Employees table

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY, -- Primary Key
    emp_name VARCHAR(100) NOT NULL,
    dept_id INT, -- Foreign Key column
    salary DECIMAL(10, 2),
    Foreign Key Constraint
    FOREIGN KEY (dept_id) REFERENCES Departments(dept_id)
);
```

Output

Table Departments created successfully.
Table Employees created successfully.

2. Altering Tables

The ALTER TABLE command is used to modify an existing table's structure, such as adding/removing columns, modifying constraints, or changing data types.

Syntax for Altering Tables

```
ALTER TABLE table_name
    ADD column_name datatype [constraints];

ALTER TABLE table_name
    MODIFY column_name datatype [constraints];
```

```
ALTER TABLE table_name
   DROP COLUMN column_name;
```

Example: Altering Tables

- **Add a new column:**

```
ALTER TABLE Employees
ADD hire_date DATE;
```

- **Modify a column:**

```
ALTER TABLE Employees
MODIFY salary DECIMAL(15, 2);
```

Output

Table Employees altered successfully: Column hire_date added.
Table Employees altered successfully: Column salary modified.

- **Add a foreign key constraint:**

```
ALTER TABLE Employees
ADD CONSTRAINT fk_department
FOREIGN KEY (dept_id) REFERENCES Departments(dept_id);
```

- **Drop a column:**

```
ALTER TABLE Employees
DROP COLUMN hire_date;
```

- **Rename a column (DB-specific, e.g., MySQL):**

```
ALTER TABLE Employees
CHANGE COLUMN emp_name employee_name VARCHAR(150);
```

3. Dropping Tables

The DROP TABLE command is used to permanently delete a table and all its data from the database. Be cautious as this action is irreversible.

Syntax for Dropping Tables

```
DROP TABLE table_name;
```

Example: Dropping Tables

- **Delete the Employees table:**

```
DROP TABLE Employees;
```

Output

Table Employees dropped successfully.

- **Delete multiple tables:**

```
DROP TABLE Employees, Departments;
```

4. Additional Notes

- **On Constraints:**
  - Primary keys ensure the uniqueness of a record in a table.
  - Foreign keys establish relationships between tables, enforcing referential integrity.
- **Dropping a Table with Constraints:** If a table has foreign key dependencies, you may need to drop or disable those constraints before dropping the table.
- **Example of Dropping Foreign Key Constraints:**

```
ALTER TABLE Employees
DROP FOREIGN KEY fk_department;
```

- **Truncate vs Drop:**
  - **TRUNCATE**: Removes all rows from a table but retains its structure.
  - **DROP**: Deletes the entire table, including its structure.

EXPERIMENT-2

Objective: Practicing DML commands- Insert, Select, Update, Delete.

1. DML (Data Manipulation Language)

DML queries are used to insert, update, and delete data within tables.

Insert data into a table:

INSERT INTO employees (employee_id, first_name, last_name, department, salary) VALUES (1, 'John', 'Doe', 'HR', 50000.00);

Output:
1 row inserted successfully.

Update data in a table:

UPDATE employees
SET salary = 55000.00
WHERE employee_id = 1;

Output:
1 row updated successfully.

Delete data from a table:

DELETE FROM employees
WHERE employee_id = 1;

Output:
1 row deleted successfully.

Select data from a table:

SELECT * FROM employees;

Output (Example if there were records):

employee_id	first_name	last_name	department	salary	email
1	John	Doe	HR	55000.00	john.doe@example.com

2. DCL (Data Control Language)

DCL queries are used to grant and revoke permissions on database objects.

Grant permissions:

GRANT SELECT, INSERT ON employees TO user_name;

Output:
Permissions granted: user 'user_name' can SELECT and INSERT on table 'employees'.

Revoke permissions:

REVOKE INSERT ON employees FROM user_name;

Output:
Permissions revoked: user 'user_name' no longer has INSERT permission on table 'employees'.

Overall Conclusion

These SQL statements work together to provide full functionality in managing, securing, and operating within a database environment.

EXPERIMENT-3

Objective: Write queries using Logical Operators (=, <,> etc.).

SQL also includes **comparison operators**, which are often used in conjunction with logical operators like AND, OR, and NOT. These comparison operators are typically used in the WHERE clause to specify conditions that help filter data based on specific criteria. Here's an overview of the most common comparison operators in SQL:

1. Equal To (=)

- Checks if a value is equal to another value.
- **Example**: SELECT * FROM Employees WHERE Department = 'Sales';
    - This query returns all employees who work in the Sales department.

2. Not Equal To (<> or !=)

- Checks if a value is not equal to another value.
- <> is the SQL standard, but some databases also support !=.
- **Example**: SELECT * FROM Employees WHERE Department <> 'Sales';
    - This query returns all employees who do not work in the Sales department.

3. Greater Than (>)

- Checks if a value is greater than another value.
- **Example**: SELECT * FROM Employees WHERE Age > 30;
    - This query returns all employees who are older than 30.

4. Less Than (<)

- Checks if a value is less than another value.
- **Example**: SELECT * FROM Employees WHERE Age < 30;
    - This query returns all employees who are younger than 30.

5. Greater Than or Equal To (>=)

- Checks if a value is greater than or equal to another value.
- **Example**: SELECT * FROM Employees WHERE Age >= 30;
    - This query returns all employees who are 30 years old or older.

6. Less Than or Equal To (<=)

- Checks if a value is less than or equal to another value.
- **Example**: SELECT * FROM Employees WHERE Age <= 30;
    - This query returns all employees who are 30 years old or younger.

Assume we have the following **tables** in a school database:

1. Student Table:

StudentID FirstName LastName Age Gender GPA

1	John	Doe	18	M	3.5
2	Jane	Smith	20	F	3.8
3	Alice	Johnson	19	F	2.9
4	Bob	Brown	17	M	3.0

2. Course Table:

CourseID	CourseName	Credits
101	Mathematics	3
102	English	2
103	Computer Science	4
104	History	3

Queries Using Logical Operators Example

1. Select Students Older Than 18

```
SELECT * FROM Student
WHERE Age > 18;
```

Output:

StudentID	FirstName	LastName	Age	Gender	GPA
2	Jane	Smith	20	F	3.8
3	Alice	Johnson	19	F	2.9

Conclusion: This query retrieves students who are older than 18 years. The > operator filters records based on the Age column. Here, only Jane and Alice meet this condition.

2. Select Courses with 3 or More Credits

```
SELECT * FROM Course
WHERE Credits >= 3;
```

Output:

CourseID	CourseName	Credits
101	Mathematics	3
103	Computer Science	4
104	History	3

Conclusion: This query uses the >= operator to filter courses with 3 or more credits. It returns "Mathematics," "Computer Science," and "History," all of which meet the condition.

3. Select Students with GPA Not Equal to 3.0

```
SELECT * FROM Student
WHERE GPA != 3.0;
```

Output:

StudentID	FirstName	LastName	Age	Gender	GPA
1	John	Doe	18	M	3.5
2	Jane	Smith	20	F	3.8
3	Alice	Johnson	19	F	2.9

Conclusion: This query retrieves students with a GPA other than 3.0 using the! = operator. John, Jane, and Alice have GPAs that satisfy this condition, while Bob does not.

4. Select Female Students

```
SELECT * FROM Student
WHERE Gender = 'F';
```

Output:

StudentID	FirstName	LastName	Age	Gender	GPA
2	Jane	Smith	20	F	3.8
3	Alice	Johnson	19	F	2.9

Conclusion: This query uses the = operator to filter female students. It only selects records where Gender is "F."

5. Select Courses with Credits Less Than 4

```
SELECT * FROM Course
WHERE Credits < 4;
```

Output:

CourseID	CourseName	Credits
101	Mathematics	3
102	English	2
104	History	3

Conclusion: This query retrieves courses with fewer than 4 credits using the < operator. "Mathematics," "English," and "History" match this condition.

6. Select Students Whose Age is Less Than or Equal to 18

```
SELECT * FROM Student
WHERE Age <= 18;
```

Output:

StudentID	FirstName	LastName	Age	Gender	GPA
1	John	Doe	18	M	3.5
4	Bob	Brown	17	M	3.0

Conclusion: This query uses the <= operator to filter students who are 18 or younger. John and Bob meet this condition, so they are selected.

Summary Conclusion

Logical operators in SQL (=, <, >, <=, >=, !=) are essential for filtering data based on specific conditions. Using these operators:

1. **Filters Data**: Queries narrow down the data to only those rows that meet specified criteria.
2. **Improves Query Efficiency**: Retrieving only necessary records reduces processing time.
3. **Enhances Data Analysis**: Selectively displaying records based on conditions helps analyze data more effectively.

Each logical operator has a specific function, making it possible to construct precise and meaningful queries for any dataset.

EXPERIMENT-4

Objective: Write queries using SQL Operators (BETWEEN, AND, IN, LIKE, ISNULL and along with Negation expressions.).

1. BETWEEN

- Checks if a value falls within a specific range, inclusive of the start and end values.
- **Example**: SELECT * FROM Orders WHERE OrderDate BETWEEN '2023-01-01' AND '2023-12-31';
    - This query returns all orders placed within the specified date range.

2. IN

- Checks if a value matches any value in a specified list.
- Simplifies multiple OR conditions.
- **Example**: SELECT * FROM Products WHERE Category IN ('Electronics', 'Books', 'Clothing');
    - This query returns products that belong to the Electronics, Books, or Clothing categories.

3. LIKE

- Used with pattern matching to search for a specified pattern in a column (usually with strings).
- Wildcards like % (matches any sequence of characters) and _ (matches a single character) can be used.
- **Example**: SELECT * FROM Customers WHERE Name LIKE 'A%';
    - This query returns customers whose names start with "A."

4. IS NULL / IS NOT NULL

- Checks if a column value is NULL (i.e., has no value) or is not NULL.
- **Example**: SELECT * FROM Employees WHERE Phone IS NULL;
    - This query returns employees who do not have a phone number listed.

Using Comparison Operators with Logical Operators

- Comparison operators are commonly used with logical operators (AND, OR, NOT) for complex conditions.
- **Example**: SELECT * FROM Employees WHERE Age > 30 AND Department = 'Sales';
    - This query returns employees who are older than 30 and work in the Sales department.

Precedence of Comparison Operators

- Comparison operators are evaluated based on precedence rules in SQL, which are similar to standard mathematical operator precedence.

SQL queries using the specified SQL operators (BETWEEN, AND, IN, LIKE, IS NULL) and including negation expressions where applicable. Let's assume we have two sample tables, **Employees** and **Orders**:

Employees Table

EmployeeID	Name	Age	Department	Salary	HireDate	Phone
1	Alice Green	28	Sales	60000	2021-03-15	NULL
2	Bob White	35	HR	55000	2018-08-10	123-456-789
3	Carol Black	42	IT	75000	2016-12-05	987-654-321
4	David Blue	29	Marketing	62000	2022-07-21	456-789-123
5	Eve Purple	33	Sales	58000	2020-11-02	NULL

Orders Table

OrderID	CustomerName	OrderAmount	OrderDate
101	Alice Green	250	2023-01-20
102	Bob White	500	2023-03-15
103	Carol Black	300	2022-11-10
104	David Blue	150	2023-04-22
105	Eve Purple	200	2023-08-05

1. Query Using BETWEEN with AND

Query

```
SELECT *
FROM Orders
WHERE OrderDate BETWEEN '2023-01-01' AND '2023-12-31'
AND OrderAmount >= 200;
```

Output

OrderID	CustomerName	OrderAmount	OrderDate
101	Alice Green	250	2023-01-20
102	Bob White	500	2023-03-15
105	Eve Purple	200	2023-08-05

Conclusion
This query returns orders placed within the year 2023 with an order amount of at least 200. The BETWEEN operator is used to filter records within a date range, while AND ensures the order amount is also checked. **2. Query Using IN with Negation (NOT IN)**

Query

```
SELECT *
FROM Employees
WHERE Department NOT IN ('Sales', 'IT');
```

Output

EmployeeID	Name	Age	Department	Salary	HireDate	Phone
2	Bob White	35	HR	55000	2018-08-10	123-456-789
4	David Blue	29	Marketing	62000	2022-07-21	456-789-123

Conclusion
This query returns employees who do not belong to the Sales or IT departments. The NOT IN operator is used to exclude certain departments from the results.

3. Query Using LIKE with a Wildcard and NOT LIKE

Query

```
SELECT *
FROM Employees
WHERE Name LIKE 'A%' AND Department NOT LIKE 'IT';
```

Output

EmployeeID	Name	Age	Department	Salary	HireDate	Phone
1	Alice Green	28	Sales	60000	2021-03-15	NULL

Conclusion
This query returns employees whose names start with "A" (LIKE 'A%') and who do not work in the IT department (NOT LIKE 'IT'). The LIKE operator with a wildcard (%) helps filter based on partial matches.

4. Query Using IS NULL with Negation (IS NOT NULL)
Query

```
SELECT *
FROM Employees
WHERE Phone IS NULL OR Department IS NOT NULL;
```

Output

EmployeeID	Name	Age	Department	Salary	HireDate	Phone
1	Alice Green	28	Sales	60000	2021-03-15	NULL
2	Bob White	35	HR	55000	2018-08-10	123-456-789
3	Carol Black	42	IT	75000	2016-12-05	987-654-321
4	David Blue	29	Marketing	62000	2022-07-21	456-789-123
5	Eve Purple	33	Sales	58000	2020-11-02	NULL

Conclusion

This query retrieves all employees who either do not have a phone number listed (Phone IS NULL) or have a non-null department (Department IS NOT NULL). The IS NULL and IS NOT NULL operators are used to check for null values in specific columns.

5. Complex Query Combining BETWEEN, AND, IN, and IS NOT NULL

Query

```
SELECT *
FROM Employees
WHERE HireDate BETWEEN '2020-01-01' AND '2023-12-31'
    AND Department IN ('Sales', 'Marketing')
    AND Phone IS NOT NULL;
```

Output

EmployeeID	Name	Age	Department	Salary	HireDate	Phone
4	David Blue	29	Marketing	62000	2022-07-21	456-789-123

Conclusion

This query retrieves employees hired between January 1, 2020, and December 31, 2023, who work in either Sales or Marketing and have a listed phone number. By combining BETWEEN, IN, and IS NOT NULL, we can apply multiple conditions to filter down to very specific results.

These examples illustrate how SQL operators like BETWEEN, AND, IN, LIKE, IS NULL, and their negations can be used together to form complex queries for refined data filtering. Each operator has a unique purpose, and combining them can provide highly specific results from the database.

EXPERIMENT-5

Objective: Write queries using COUNT, SUM, AVG, MAX, MIN, GROUP BY, HAVING.

1. Group Functions (Aggregate Functions)

Group functions perform calculations on sets of rows and return a single result. They are commonly used with the GROUP BY clause.

Function	Description	Example
COUNT	Counts the number of rows or values in a column.	SELECT COUNT(EmployeeID) AS EmployeeCount FROM Employees;
SUM	Returns the sum of a numeric column.	SELECT SUM(Salary) AS TotalSalary FROM Employees;
AVG	Returns the average value of a numeric column.	SELECT AVG(Salary) AS AverageSalary FROM Employees;
MAX	Returns the maximum value in a column.	SELECT MAX(Salary) AS HighestSalary FROM Employees;
MIN	Returns the minimum value in a column.	SELECT MIN(Salary) AS LowestSalary FROM Employees;

SQL aggregation functions (COUNT, SUM, AVG, MAX, MIN) along with GROUP BY and HAVING,

Sample Table: Employees
Structure:

emp_id	emp_name	dept_id	salary	hire_date
101	John Doe	1	50000	2020-01-15
102	Jane Smith	2	60000	2019-03-22
103	Michael Brown	1	45000	2021-07-18
104	Emily Davis	3	70000	2018-11-05
105	David Wilson	2	65000	2020-09-30

Queries with Aggregation Functions
1. Count the Total Number of Employees

```
SELECT COUNT(*) AS total_employees
FROM Employees;
```

Output:

total_employees
5

2. Sum of Salaries by Department

```
SELECT dept_id, SUM(salary) AS total_salary
FROM Employees
GROUP BY dept_id;
```

Output:

dept_id total_salary

1 95000

2 125000

3 70000

3. Average Salary of All Employees

```
SELECT AVG(salary) AS average_salary
FROM Employees;
```

Output:

average_salary

58000.00

4. Maximum and Minimum Salaries in the Company

```
SELECT MAX(salary) AS highest_salary, MIN(salary) AS lowest_salary
FROM Employees;
```

Output:

highest_salary lowest_salary

70000 45000

5. Group Salaries by Department and Count Employees

```
SELECT dept_id, COUNT(emp_id) AS employee_count, AVG(salary) AS average_salary
FROM Employees
GROUP BY dept_id;
```

Output:

dept_id employee_count average_salary

1 2 47500.00

2 2 62500.00

3 1 70000.00

Using HAVING Clause

6. Departments with Total Salaries Greater than 100,000

```
SELECT dept_id, SUM(salary) AS total_salary
FROM Employees
GROUP BY dept_id
HAVING SUM(salary) > 100000;
```

Output:

dept_id total_salary

2 125000

7. Departments with More than One Employee

```
SELECT dept_id, COUNT(emp_id) AS employee_count
FROM Employees
GROUP BY dept_id
HAVING COUNT(emp_id) > 1;
```

Output:

dept_id employee_count

dept_id employee_count

1 2

2 2

Combining Aggregation Functions

8. Total Salary, Average Salary, and Employee Count by Department

```
SELECT
   dept_id,
   COUNT(emp_id) AS total_employees,
   SUM(salary) AS total_salary,
   AVG(salary) AS average_salary
FROM Employees
GROUP BY dept_id;
```

Output:

dept_id	total_employees	total_salary	average_salary
1	2	95000	47500.00
2	2	125000	62500.00
3	1	70000	70000.00

9. Combining GROUP BY and HAVING for Complex Queries

Query: Departments with Average Salary Above 50,000 and More Than One Employee

```
SELECT dept_id, AVG(salary) AS average_salary, COUNT(emp_id) AS employee_count
FROM Employees
GROUP BY dept_id
HAVING AVG(salary) > 50000 AND COUNT(emp_id) > 1;
```

Output:

dept_id	average_salary	employee_count
2	62500.00	2

Summary

- **COUNT**: Counts rows in a table or group.
- **SUM**: Adds values of a specific column.
- **AVG**: Calculates the average value of a column.
- **MAX / MIN**: Finds the highest/lowest value in a column.
- **GROUP BY**: Groups rows by specified columns.
- **HAVING**: Filters aggregated data after grouping.

EXPERIMENT-6

Objective: Write queries using ANY, ALL, IN, EXISTS, NOT EXISTS, UNION, INTERSECT, MINUS, CONSTRAINTS etc.

Sample Tables

Table: Employees

emp_id	emp_name	dept_id	salary
101	John Doe	1	50000
102	Jane Smith	2	60000
103	Michael Brown	1	45000
104	Emily Davis	3	70000
105	David Wilson	2	65000

Table: Departments

dept_id	dept_name
1	HR
2	Finance
3	IT

1. Using ANY

Find employees whose salary is greater than **any** employee in department 1.

```
SELECT emp_name, salary
FROM Employees
WHERE salary > ANY (SELECT salary FROM Employees WHERE dept_id = 1);
```

Output:

emp_name	salary
Jane Smith	60000
Emily Davis	70000
David Wilson	65000

2. Using ALL

Find employees whose salary is greater than **all** employees in department 1.

```
SELECT emp_name, salary
FROM Employees
WHERE salary > ALL (SELECT salary FROM Employees WHERE dept_id = 1);
```

Output:

emp_name	salary
Emily Davis	70000
David Wilson	65000

3. Using IN

List employees who belong to departments HR or Finance.

```
SELECT emp_name, dept_id
FROM Employees
WHERE dept_id IN (1, 2);
```

Output:

emp_name	dept_id
John Doe	1
Jane Smith	2
Michael Brown	1
David Wilson	2

4. Using EXISTS
Find departments with employees.
SELECT dept_name
FROM Departments d
WHERE EXISTS (SELECT 1 FROM Employees e WHERE e.dept_id = d.dept_id);
Output:

dept_name
HR
Finance
IT

5. Using NOT EXISTS
Find departments without employees.
SELECT dept_name
FROM Departments d
WHERE NOT EXISTS (SELECT 1 FROM Employees e WHERE e.dept_id = d.dept_id);
Output: No rows returned (since all departments have employees).

6. Using UNION
Combine employees from HR and Finance departments.
SELECT emp_name, dept_id
FROM Employees
WHERE dept_id = 1
UNION
SELECT emp_name, dept_id
FROM Employees
WHERE dept_id = 2;
Output:

emp_name	dept_id
John Doe	1
Jane Smith	2
Michael Brown	1
David Wilson	2

7. Using INTERSECT
Find employees in both department HR and Finance. (This requires overlapping rows, which might not exist based on sample data.)
SELECT emp_name, dept_id
FROM Employees
WHERE dept_id = 1

INTERSECT
SELECT emp_name, dept_id
FROM Employees
WHERE dept_id = 2;
Output: No rows returned (since no employee belongs to both departments).

8. Using MINUS
Find employees in HR but not in Finance.
SELECT emp_name, dept_id
FROM Employees
WHERE dept_id = 1
MINUS
SELECT emp_name, dept_id
FROM Employees
WHERE dept_id = 2;
Output:

emp_name	**dept_id**
John Doe	1
Michael Brown	1

9. Using Constraints
Add a Primary Key Constraint:
CREATE TABLE Departments (
  dept_id INT PRIMARY KEY, -- Primary Key
  dept_name VARCHAR(100) NOT NULL
);
Output: Table Departments created successfully.

Add a Foreign Key Constraint:
ALTER TABLE Employees
ADD CONSTRAINT fk_department FOREIGN KEY (dept_id) REFERENCES Departments(dept_id);
Output: *Foreign key constraint added successfully.*

Add a Unique Constraint:
ALTER TABLE Employees
ADD CONSTRAINT unique_emp_name UNIQUE (emp_name);
Output: Unique constraint added successfully.

Add a Check Constraint:
ALTER TABLE Employees
ADD CONSTRAINT check_salary CHECK (salary > 0);
Output: Check constraint added successfully.

Summary of Outputs

Feature	Query Result
ANY	Returns rows that meet the condition with at least one comparison.
ALL	Returns rows that meet the condition with all comparisons.
IN	Filters rows matching a list of values.
EXISTS	Checks for the existence of rows in a subquery.
NOT EXISTS	Filters rows where subquery has no matching results.
UNION	Combines rows from multiple queries, removing duplicates.
INTERSECT	Returns rows common to both queries.
MINUS	Returns rows present in one query but not in the other.
Constraints	Enforces database integrity rules.

EXPERIMENT-7

Objective: Write queries for extracting data from more than one table (Equi-Join, Non-Equi Join, Inner Join, Outer Join).

A join is a query that combines rows from two or more tables, views, or materialized views. Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

The primary types of joins include **Equi-join**, **Non-Equi join**, and **Outer join**.

1. Equi-Join:

An equi-join is a type of join that combines tables based on equality between specified columns. It's one of the most common joins and typically involves using the = operator to match rows from two tables where the values in the specified columns are equal.

2. Non-Equi Join:

A non-equi join combines tables based on a condition other than equality. It can use operators such as <, >, <=, >=, and <> (not equal). Non-equi joins are useful when the relationship between tables doesn't involve exact matches.

3. Outer Join:

An outer join returns all rows from one table and matching rows from another table, filling in NULL where there's no match. There are three types of outer joins:

Left Outer Join (LEFT JOIN): Returns all rows from the left table and the matched rows from the right table. If there is no match, NULL is returned for columns from the right table.

Right Outer Join (RIGHT JOIN): Returns all rows from the right table and the matched rows from the left table. If there is no match, NULL is returned for columns from the left table.

Full Outer Join (FULL JOIN): Returns all rows from both tables, with NULL for columns where there's no match on either side.

Subqueries and **Nested queries**

In SQL, **subqueries** and **nested queries** are techniques used to build complex queries by embedding one query inside another. They allow you to perform multiple steps in a single query, enabling SQL to handle complex conditions and calculations.

Subqueries (or Inner Queries)

A subquery, also known as an **inner query** or **nested query**, is a query embedded within

another SQL statement, usually inside SELECT, FROM, WHERE, or HAVING clauses. The subquery executes first, and its result is used by the main (outer) query.

Types of Subqueries:

1. **Single-row subqueries**: Return a single row and are typically used with comparison operators such as =, <, >, <=, or >=.
2. **Multiple-row subqueries**: Return multiple rows and are commonly used with operators like IN, ANY, or ALL.
3. **Scalar subqueries**: Return a single value and can be used wherever a single value is expected.
4. **Correlated subqueries**: Refer to columns in the outer query and are re-evaluated for each row processed by the outer query.

SQL queries for each type of join (Equi-Join, Non-Equi Join, and Outer Join) with sample tables and output examples. Let's assume we have two tables: Employees and Departments.

Sample Tables

Employees Table

Emp_ID	Emp_Name	Dept_ID	Salary
1	Alice	101	50000
2	Bob	102	55000
3	Charlie	101	60000
4	David	103	45000
5	Eve	NULL	70000

Departments Table

Dept_ID	Dept_Name
101	Sales
102	Marketing
103	HR
104	Finance

1. Equi-Join (Inner Join)

An **Equi-Join** returns rows with matching values in the specified columns between two tables. Here, we join Employees and Departments on Dept_ID.

Query

```
SELECT Employees.Emp_ID, Employees.Emp_Name, Employees.Salary,
Departments.Dept_Name
FROM Employees
JOIN Departments
ON Employees.Dept_ID = Departments.Dept_ID;
```

Output

Emp_ID	Emp_Name	Salary	Dept_Name
1	Alice	50000	Sales
2	Bob	55000	Marketing
Emp_ID	**Emp_Name**	**Salary**	**Dept_Name**
3	Charlie	60000	Sales
4	David	45000	HR

Conclusion:
This query returns only those employees who have a department assigned (Dept_ID), as it excludes rows with NULL or unmatched Dept_ID values in either table.

2. Non-Equi Join

A **Non-Equi Join** uses conditions other than equality to join two tables. Let's assume we want to find employees whose salary falls within a certain range associated with a department's budget (assuming a third table called DepartmentBudgets).

DepartmentBudgets Table

Dept_ID	Min_Salary	Max_Salary
101	45000	60000
102	50000	65000
103	40000	55000

Query

```
SELECT Employees.Emp_ID, Employees.Emp_Name, Employees.Salary,
Departments.Dept_Name
FROM Employees
JOIN DepartmentBudgets
ON Employees.Dept_ID = DepartmentBudgets.Dept_ID
AND Employees.Salary BETWEEN DepartmentBudgets.Min_Salary AND
DepartmentBudgets.Max_Salary
JOIN Departments
ON Employees.Dept_ID = Departments.Dept_ID;
```

Output

Emp_ID	Emp_Name	Salary	Dept_Name
1	Alice	50000	Sales
2	Bob	55000	Marketing
3	Charlie	60000	Sales
4	David	45000	HR

Conclusion:
This Non-Equi Join filters employees who fall within the salary range specified for their departments, showing a practical use case for range-based conditions.

3. Outer Join (Left Join)

An **Outer Join** includes rows that do not have matching values in the other table. Here, a **Left Join** on Employees and Departments shows all employees, even those without a department.

Query

```
SELECT Employees.Emp_ID, Employees.Emp_Name, Employees.Salary, Departments.Dept_Name
FROM Employees
LEFT JOIN Departments
ON Employees.Dept_ID = Departments.Dept_ID;
```

Output

Emp_ID	Emp_Name	Salary	Dept_Name
1	Alice	50000	Sales
2	Bob	55000	Marketing
3	Charlie	60000	Sales
4	David	45000	HR
5	Eve	70000	NULL

Conclusion:
The Left Join includes all rows from Employees, even if there’s no matching Dept_ID in Departments. Employee "Eve" has a NULL department, which appears in the result, demonstrating how Left Join handles missing data in the joined table.

EXPERIMENT-8

Objective: Write SQL Queries for Sub queries, Nested queries, VIEWS Creation and Dropping.

Subqueries and nested queries allow SQL to perform more complex operations by using the results of one query within another. Here are examples of both types, with sample tables, queries, outputs, and conclusions.

Sample Tables

Let's use the same tables as before, with a slight modification and an additional table:

Employees Table

Emp_ID	Emp_Name	Dept_ID	Salary
1	Alice	101	50000
2	Bob	102	55000
3	Charlie	101	60000
4	David	103	45000
5	Eve	NULL	70000

Departments Table

Dept_ID	Dept_Name
101	Sales
102	Marketing
103	HR
104	Finance

Salaries Table
(Stores details of average salaries per department)

Dept_ID	Avg_Salary
101	55000
102	56000
103	47000

1. Subquery

A **Subquery** is a query nested within another SQL query. It can be used in SELECT, WHERE, or FROM clauses. Here, we'll use a subquery to find employees who earn more than the average salary in their department.

Query

```
SELECT Emp_Name, Salary
FROM Employees
WHERE Salary > (SELECT Avg_Salary FROM Salaries WHERE Dept_ID = Employees.Dept_ID);
```

Output

Emp_Name	Salary
Charlie	60000
Eve	70000

Explanation:

1. The subquery (SELECT Avg_Salary FROM Salaries WHERE Dept_ID = Employees.Dept_ID) retrieves the average salary for each employee's department.
2. The main query then filters to show only those employees whose salary is higher than the department's average.

Conclusion:
This query highlights employees who earn above-average salaries in their respective departments. "Charlie" and "Eve" meet this condition.

2. Nested Query

A **Nested Query** is a more complex use of subqueries, where multiple subqueries are stacked or layered within each other. This example will show the highest-paid employee(s) within each department.

Query

```
SELECT Emp_Name, Salary, Dept_ID
FROM Employees
WHERE Salary = (SELECT MAX(Salary)
        FROM Employees AS E
        WHERE E.Dept_ID = Employees.Dept_ID);
```

Output

Emp_Name	Salary	Dept_ID
Charlie	60000	101
Bob	55000	102
David	45000	103
Eve	70000	NULL

Explanation:

1. The inner subquery (SELECT MAX(Salary) FROM Employees AS E WHERE E.Dept_ID = Employees.Dept_ID) finds the maximum salary within each department.
2. The main query filters the Employees table to show only employees whose salary matches the maximum salary found in their department.

Conclusion:
This nested query identifies the top earners in each department. Each row in the result represents an employee who earns the highest salary within their department. This is helpful when identifying department leaders or top-paid individuals.

3. Using a Subquery with Aggregation

Let's write a query to find departments with an average employee salary greater than $50,000, using a subquery with aggregation.

Query

```
SELECT
Dept_Name
FROM
Departments
WHERE Dept_ID IN (SELECT Dept_ID
        FROM   Employees
        GROUP          BY
        Dept_ID
        HAVING AVG(Salary) > 50000);
```

Output
Dept_Name
Sales
Marketing

Explanation:

1. The subquery (SELECT Dept_ID FROM Employees GROUP BY Dept_ID HAVING AVG(Salary) > 50000) calculates the average salary for each department and returns those departments with an average salary greater than $50,000.
2. The main query uses this list of Dept_IDs to retrieve the corresponding Dept_Names from the Departments table.

Conclusion:
This query identifies departments with a higher average employee salary, helping in understanding where salaries are generally higher across departments. "Sales" and

"Marketing" departments meet this criterion.

Views:

In Database Management Systems (DBMS), a **view** is a virtual table that is based on the result-set of an SQL query. Unlike a regular table, a view does not store data physically; instead, it provides a way to look at the data stored in one or more underlying tables. Creating a **view** in a Database Management System (DBMS) is a way of creating a virtual table that represents the result of a query. This table does not store data itself; rather, it dynamically pulls data from underlying tables whenever the view is queried. Views can simplify complex queries, restrict data access, and provide a more user-friendly structure for retrieving data.

Syntax for Creating a View

To create a view, you use the CREATE VIEW command, followed by the view's name and an SQL query that defines it.

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

- **view_name**: Name of the view you want to create.
- **SELECT statement**: Defines the data that will appear in the view.

Example Scenario

Suppose you have an employees table with the following structure:

id	name	department	salary
1	Alice	HR	60000
2	Bob	Engineering	75000
3	Charlie	Marketing	50000
4	Diana	Engineering	70000
5	Eve	Marketing	55000

1. Creating a Simple View

Let's create a view that shows only employee names and departments.

Query

```
CREATE VIEW employee_overview AS
SELECT name, department
FROM employees;
```

Explanation

- The view employee_overview will now act as a virtual table showing only the

name and department columns from the employee's table.

Result

Querying employee_overview will display:

SELECT * FROM employee_overview;
Output:

name	department
Alice	HR
Bob	Engineering
Charlie	Marketing
Diana	Engineering
Eve	Marketing

2. Creating a View with Conditions
Suppose you only want to view employees in the "Engineering" department.

Query

CREATE VIEW engineering_employees
AS SELECT name, salary
FROM employees
WHERE department = 'Engineering';

Explanation

- The view engineering_employees will filter out only those employees who belong to the "Engineering" department.

Result

Querying engineering_employees:

SELECT * FROM

engineering_employees;

Output:

name	salary
Bob	75000
Diana	70000

Conclusion
Creating views in DBMS allows users to:

- Simplify complex queries,
- Restrict and control data access,
- Provide a more organized way to access data without altering the underlying tables.

EXPERIMENT-9

Objective: CASE STUDY: Student should decide on a case study and formulate the problem statement, Database Design using ER Model (Identifying entities, attributes, keys and relationships between entities, cardinalities, generalization, specialization etc.) Note: Student is required to submit a document by drawing ER-Diagram to the Lab teacher.

Case Study: School Management System

Problem Statement: Design a database for a school management system to handle information about students, teachers, classes, subjects, and administrative details. The system should efficiently manage data related to student enrollment, teacher assignments, class schedules, and examination results while supporting reporting and analytics.

1. Database Design Using ER Model

Entities and Attributes:

1. **Student**:
   - Attributes: StudentID (Primary Key), Name, DateOfBirth, Gender, Address, PhoneNumber, Email, ClassID (Foreign Key)
2. **Teacher**:
   - Attributes: TeacherID (Primary Key), Name, Qualification, SubjectID (Foreign Key), PhoneNumber, Email
3. **Class**:
   - Attributes: ClassID (Primary Key), ClassName, Section, Grade
4. **Subject**:
   - Attributes: SubjectID (Primary Key), SubjectName, Description
5. **Enrollment**:
   - Attributes: EnrollmentID (Primary Key), StudentID (Foreign Key), ClassID (Foreign Key), EnrollmentDate
6. **Exam**:
   - Attributes: ExamID (Primary Key), ExamName, Date, ClassID (Foreign Key)
7. **ExamResult**:
   - Attributes: ResultID (Primary Key), ExamID (Foreign Key), StudentID (Foreign Key), MarksObtained, Grade
8. **Admin**:
   - Attributes: AdminID (Primary Key), Name, Role, Email, PhoneNumber

Relationships:

1. **Student-Class Relationship**:
   - Type: Many-to-One
   - Description: Each student is assigned to one class, but a class can have multiple students.
2. **Class-Teacher Relationship**:
   - Type: Many-to-One
   - Description: Each class is managed by one teacher, but a teacher can manage multiple classes.

3. **Class-Subject Relationship**:
    - Type: Many-to-Many
    - Description: A class can have multiple subjects, and a subject can be taught in multiple classes.
4. **Student-Enrollment Relationship**:
    - Type: One-to-Many
    - Description: A student can have multiple enrollment records for different classes over time.
5. **Exam-Class Relationship**:
    - Type: One-to-Many
    - Description: An exam is associated with one class, but a class can have multiple exams.
6. **ExamResult-Student Relationship**:
    - Type: Many-to-One
    - Description: A student can have multiple exam results, but each result is for one student.
7. **ExamResult-Exam Relationship**:
    - Type: Many-to-One
    - Description: Each result is linked to one exam, but an exam can have multiple results.

Cardinalities:

- **Student-Class**: Many-to-One
- **Class-Teacher**: Many-to-One
- **Class-Subject**: Many-to-Many
- **Student-Enrollment**: One-to-Many
- **Exam-Class**: One-to-Many
- **ExamResult-Student**: Many-to-One
- **ExamResult-Exam**: Many-to-One

Generalization and Specialization:

- **Specialization of Roles:** The entity "User" can be specialized into "Student," "Teacher," and "Admin." Additional attributes for specific roles could be included, such as enrollment details for students or qualifications for teachers.
- **Generalization of Assessment:** Exam and Assignment entities can be generalized into an "Assessment" entity to represent different types of evaluations.

ER Diagram:

The ER Diagram represents the entities, attributes, and relationships visually. It includes the primary keys, foreign keys, and cardinalities.

Diagram Components:

1. Rectangles for entities (e.g., Student, Teacher, Class, etc.)
2. Ovals for attributes (e.g., Name, Email, etc.)
3. Diamonds for relationships (e.g., EnrolledIn, TaughtBy, etc.)
4. Lines to connect entities with relationships and attributes.

EXPERIMENT-10

Objective: Converting ER Model to Relational Model (Represent entities and relationships in Tabular form, represent attributes as columns, identifying keys), Create tables using SQL.

Note: Student is required to submit a document showing the database tables created from ER Model.

-- Creating Student Table

```
CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100),
    DateOfBirth DATE,
    Gender CHAR(1),
    Address VARCHAR(255),
    PhoneNumber VARCHAR(15),
    Email VARCHAR(100),
    ClassID INT,
    FOREIGN KEY (ClassID) REFERENCES Class(ClassID)
);
```

-- Creating Teacher Table

```
CREATE TABLE Teacher (
    TeacherID INT PRIMARY KEY,
    Name VARCHAR(100),
    Qualification VARCHAR(100),
    SubjectID INT,
    PhoneNumber VARCHAR(15),
    Email VARCHAR(100),
    FOREIGN KEY (SubjectID) REFERENCES Subject(SubjectID)
```

```
);
-- Creating Class Table
CREATE TABLE Class (
    ClassID INT PRIMARY KEY,
    ClassName VARCHAR(50),
    Section CHAR(1),
    Grade VARCHAR(10)
);
-- Creating Subject Table
CREATE TABLE Subject (
    SubjectID INT PRIMARY KEY,
    SubjectName VARCHAR(100),
    Description TEXT
);
-- Creating Enrollment Table
CREATE TABLE Enrollment (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    ClassID INT,
    EnrollmentDate DATE,
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (ClassID) REFERENCES Class(ClassID)
);
-- Creating Exam Table
CREATE TABLE Exam (
```

```
    ExamID INT PRIMARY KEY,
    ExamName VARCHAR(100),
    Date DATE,
    ClassID INT,
    FOREIGN KEY (ClassID) REFERENCES Class(ClassID)
);
-- Creating ExamResult Table
CREATE TABLE ExamResult (
    ResultID INT PRIMARY KEY,
    ExamID INT,
    StudentID INT,
    MarksObtained INT,
    Grade CHAR(2),
    FOREIGN KEY (ExamID) REFERENCES Exam(ExamID),
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID)
);
-- Creating Admin Table
CREATE TABLE Admin (
    AdminID INT PRIMARY KEY,
    Name VARCHAR(100),
    Role VARCHAR(50),
    Email VARCHAR(100),
    PhoneNumber VARCHAR(15)
);
```

EXPERIMENT-11

Objective: Normalization -To remove the redundancies and anomalies in the above relational tables, Normalize up to Third Normal Form.

To normalize a student management system up to the Third Normal Form (3NF), we first need to break down the relational tables and remove redundancies and anomalies at each level of normalization: 1NF, 2NF, and 3NF.

Assumptions for the schema:

Let's assume we have a table that stores student information, course details, and enrollment data. Here's a simplified version of what the table might look like before normalization:

Student Information (Before Normalization):

StudentID	StudentName	StudentDOB	CourseName	CourseInstructor	InstructorDOB	InstructorEmail	Grade
1	John Doe	2000-04-01	Math	Dr. Smith	1975-06-15	smith@mail.com	A
2	Jane Smith	1999-07-23	Math	Dr. Smith	1975-06-15	smith@mail.com	B
3	Mark Taylor	2001-11-10	Science	Dr. Johnson	1970-09-30	johnson@mail.com	A
1	John Doe	2000-04-01	Science	Dr. Johnson	1970-09-30	johnson@mail.com	C

This table contains redundant information like instructor details and course names, which lead to data anomalies (e.g., updates to instructor details might not be consistent).

Step 1: First Normal Form (1NF)

To achieve 1NF, we need to ensure that:

- There are no repeating groups.
- Each column contains atomic values (no lists or sets).
- Each record has a unique identifier.

1NF Transformation:

In the current table, each record corresponds to a student enrolled in a specific course, with no repeating groups, but the primary key should be a combination of StudentID and CourseName since both are needed to uniquely identify a student-course enrollment.

Normalized Table in 1NF:

StudentID	StudentName	StudentDOB	CourseName	CourseInstructor	InstructorDOB	InstructorEmail	Grade
1	John Doe	2000-04-01	Math	Dr. Smith	1975-06-15	smith@mail.com	A
2	Jane Smith	1999-07-23	Math	Dr. Smith	1975-06-15	smith@mail.com	B
3	Mark Taylor	2001-11-10	Science	Dr. Johnson	1970-09-30	johnson@mail.com	A
1	John Doe	2000-04-01	Science	Dr. Johnson	1970-09-30	johnson@mail.com	C

Step 2: Second Normal Form (2NF)

To achieve 2NF, we need to ensure that:

- The table is in 1NF.
- All non-key attributes are fully functionally dependent on the primary key (i.e., no partial dependency on part of the composite key).

In our table, the composite key is (StudentID, CourseName). But we see that StudentName, StudentDOB, CourseInstructor, InstructorDOB, and InstructorEmail are dependent on part of the primary key (i.e., StudentID and CourseName).

2NF Transformation:

We should break the table into separate tables to remove partial dependencies.

Table 1: Student Table

StudentID	StudentName	StudentDOB
1	John Doe	2000-04-01
2	Jane Smith	1999-07-23
3	Mark Taylor	2001-11-10

Table 2: Instructor Table

InstructorID	CourseInstructor	InstructorDOB	InstructorEmail
1	Dr. Smith	1975-06-15	smith@mail.com
2	Dr. Johnson	1970-09-30	johnson@mail.com

Table 3: Course Table

CourseID	CourseName	InstructorID
1	Math	1
2	Science	2

Table 4: Enrollment Table

StudentID	CourseID	Grade
1	1	A
2	1	B
3	2	A
1	2	C

Step 3: Third Normal Form (3NF)

To achieve 3NF, we need to ensure that:

- The table is in 2NF.

- No transitive dependency exists. That is, non-key attributes should not depend on other non-key attributes.

In our current schema, there are no transitive dependencies. For instance, in the Enrollment Table, Grade depends directly on the composite key (StudentID, CourseID), and in the Course Table, InstructorID is already placed separately from non-key attributes.

Therefore, the schema is already in 3NF.

Final Normalized Schema (3NF):

1. **Student Table**

StudentID	StudentName	StudentDOB
1	John Doe	2000-04-01
2	Jane Smith	1999-07-23
3	Mark Taylor	2001-11-10

2. **Instructor Table**

InstructorID	CourseInstructor	InstructorDOB	InstructorEmail
1	Dr. Smith	1975-06-15	smith@mail.com
2	Dr. Johnson	1970-09-30	johnson@mail.com

3. **Course Table**

CourseID	CourseName	InstructorID
1	Math	1
2	Science	2

4. **Enrollment Table**

StudentID	CourseID	Grade
1	1	A
2	1	B
3	2	A
1	2	C

This is the normalized schema up to the Third Normal Form (3NF), where redundancies are eliminated, and anomalies are minimized.