UNIT 3 – Advanced SQL Concepts

Unit Overview

This unit deepens your SQL skills with advanced techniques used in **real-world database systems** and **placement-level queries**.
It focuses on combining data from multiple tables, improving performance, and maintaining consistency across transactions.

Unit Topics

Subtopic Code	Topic Name	Subtopics
3.1	**Joins**	Concept of Joins, INNER, LEFT, RIGHT, FULL, SELF, CROSS, Multi-table Joins
3.2	**Subqueries**	Nested Queries, Single-row & Multi-row Subqueries, IN, ANY, ALL, Correlated Subqueries
3.3	**Views**	Creating, Modifying, Dropping Views, Advantages & Limitations
3.4	**Indexing**	Concept, Clustered vs Non-Clustered, Creating Indexes, Performance Effects
3.5	**Transactions**	Concept, ACID Properties, COMMIT, ROLLBACK, SAVEPOINT, Concurrency Control
3.6	**Cursors**	Concept, Syntax, Iterating over Results in MySQL
3.7	**Checkpoint Review**	Output Prediction Queries & Logical Tracing Practice

3.1 Joins

Subtopics

- Concept of Joins
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN (Simulation in MySQL)
- SELF JOIN
- CROSS JOIN
- Multi-table Joins

Learning Outcome

After completing this topic, you will be able to:

- Combine data from multiple tables using various join operations.
- Understand the difference between inner and outer joins.
- Write complex multi-table join queries confidently.

Concept Explanation

In relational databases, data is stored in multiple related tables for **normalization** and **modularity**. To fetch related data across these tables, we use **JOIN operations**.

A **JOIN** merges rows from two or more tables based on a related column between them — usually a **foreign key** and a **primary key**.

1. Types of Joins

Join Type	Description	Diagram / Behavior
INNER JOIN	Returns rows where matching values exist in both tables.	Intersection of both tables
LEFT JOIN (LEFT OUTER JOIN)	Returns all rows from the left table and matching rows from the right.	All left + matched right
RIGHT JOIN (RIGHT OUTER JOIN)	Returns all rows from the right table and matching rows from the left.	All right + matched left
FULL JOIN (FULL OUTER JOIN)	Returns all rows when there is a match in either table.	Union of LEFT + RIGHT
SELF JOIN	Joins a table with itself.	Useful for hierarchical data
CROSS JOIN	Returns all possible combinations (Cartesian product).	n × m rows
MULTI-TABLE JOIN	Involves joining more than two tables.	Common in normalized schemas

2. INNER JOIN

Definition:

Fetches only the records that have matching values in both tables.

Example

```
CREATE DATABASE join_demo;
USE join_demo;

CREATE TABLE department (
    dept_id CHAR(3) PRIMARY KEY,
    dept_name VARCHAR(30)
```

```
);

CREATE TABLE employee (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id CHAR(3),
     salary DECIMAL(10,2),
    FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);

INSERT INTO department VALUES
('D01', 'HR'), ('D02', 'IT'), ('D03', 'Finance');

INSERT INTO employee VALUES
(1, 'Arjun', 'D01', 55000),
(2, 'Meera', 'D02', 70000),
(3, 'Kavya', 'D02', 65000),
(4, 'Rahul', 'D04', 50000);  -- invalid dept
```

Query

```
SELECT e.emp_name, d.dept_name
FROM employee e
INNER JOIN department d
ON e.dept_id = d.dept_id;
```

Output:

emp_name	dept_name
Arjun	HR
Meera	IT
Kavya	IT

Only employees with matching departments appear.

3. LEFT JOIN

Returns **all rows from the left table** (employee) and **matching rows** from the right (department).
If no match, NULLs are shown for right table columns.

```
SELECT e.emp_name, d.dept_name
```

```
FROM employee e
LEFT JOIN department d
ON e.dept_id = d.dept_id;
```

Output:

emp_name	dept_name
Arjun	HR
Meera	IT
Kavya	IT
Rahul	NULL

Employee Rahul's department doesn't exist, so NULL is shown.

4. RIGHT JOIN

Opposite of LEFT JOIN — returns **all rows from the right table** and matching ones from the left.

```
SELECT e.emp_name, d.dept_name
FROM employee e
RIGHT JOIN department d
ON e.dept_id = d.dept_id;
```

Output:

emp_name	dept_name
Arjun	HR
Meera	IT
Kavya	IT
NULL	Finance

Department "Finance" exists but has no employee.

5. FULL JOIN (Simulated in MySQL)

MySQL doesn't directly support FULL OUTER JOIN, but it can be simulated using UNION.

```
SELECT e.emp_name, d.dept_name
FROM employee e
LEFT JOIN department d ON e.dept_id = d.dept_id

UNION
```

```
SELECT e.emp_name, d.dept_name
FROM employee e
RIGHT JOIN department d ON e.dept_id = d.dept_id;
```

Output:

Combines both unmatched left and right results.

6. SELF JOIN

A **self join** joins a table with itself — typically used for hierarchical or parent-child relationships.

Example: Employee → Manager

```
CREATE TABLE employee_self (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    manager_id INT
);

INSERT INTO employee_self VALUES
(1, 'Arjun', NULL),
(2, 'Meera', 1),
(3, 'Kavya', 1),
(4, 'Divya', 2);

SELECT e.emp_name AS Employee, m.emp_name AS Manager
FROM employee_self e
LEFT JOIN employee_self m
ON e.manager_id = m.emp_id;
```

Output:

Employee	Manager
Arjun	NULL
Meera	Arjun
Kavya	Arjun
Divya	Meera

7. CROSS JOIN

Generates a **Cartesian product** — all combinations of rows from both tables.
Used rarely, mainly for data generation or testing.

```
SELECT e.emp_name, d.dept_name
FROM employee e
CROSS JOIN department d;
```

If 4 employees and 3 departments → 12 total combinations.

8. Multi-Table Joins

You can join more than two tables in a single query.

Example: employee → department → location

```
CREATE TABLE location (
    loc_id CHAR(3) PRIMARY KEY,
    city VARCHAR(30)
);
```

```
INSERT INTO location VALUES
('L01','Chennai'),
('L02','Bangalore');

ALTER TABLE department ADD loc_id CHAR(3);
UPDATE department SET loc_id = 'L01' WHERE dept_id IN ('D01','D02');
UPDATE department SET loc_id = 'L02' WHERE dept_id = 'D03';

SELECT e.emp_name, d.dept_name, l.city
FROM employee e
JOIN department d ON e.dept_id = d.dept_id
JOIN location l ON d.loc_id = l.loc_id;
```

Output:

emp_name	dept_name	city
Arjun	HR	Chennai
Meera	IT	Chennai
Kavya	IT	Chennai

Checkpoint Review

1. What is a JOIN in SQL?
2. Differentiate between INNER JOIN and LEFT JOIN.
3. Write a query to display all employees with their departments, including those without departments.
4. How can you simulate a FULL JOIN in MySQL?
5. Write a query to list employees with their managers (using SELF JOIN).

Reflection Prompt

In a college database, how would you join the students, courses, and enrollments tables to display each student's enrolled courses?

Summary

- **Joins** combine data from multiple related tables.
- **INNER JOIN** returns matches in both; **LEFT/RIGHT JOIN** keep unmatched sides.
- **FULL JOIN** can be emulated using UNION.
- **SELF JOIN** compares rows within the same table.
- **CROSS JOIN** creates Cartesian products.
- Multi-table joins integrate complete relational views of data.

3.2 Subqueries

Subtopics

- Concept of Subqueries
- Nested Queries
- Single-row & Multi-row Subqueries
- Operators: IN, ANY, ALL
- Correlated Subqueries

Learning Outcome

After completing this topic, you will be able to:

- Write SQL queries that use subqueries for dynamic data comparison.
- Differentiate between single-row and multi-row subqueries.

- Use operators like IN, ANY, and ALL effectively.
- Understand and implement correlated subqueries for row-by-row evaluation.

Concept Explanation

A **subquery** (or nested query) is a **query inside another query**.
It allows you to use the result of one query as input for another.

They make SQL **dynamic**, **modular**, and **powerful** for solving multi-step problems.

Syntax

```
SELECT column_list
FROM table
WHERE column_name [operator] (SELECT column_list FROM another_table WHERE
condition);
```

1. Simple Example of Subquery

```
SELECT department, AVG(salary)
FROM employee
GROUP BY department;
If we want to find employees earning more than the average salary, we can
use a subquery:
SELECT emp_name, salary
FROM employee
WHERE salary > (SELECT AVG(salary) FROM employee);
```

Explanation:

- Inner query → calculates average salary.
- Outer query → retrieves employees who exceed that average.

2. Nested Queries (Multi-level Subqueries)

Subqueries can be nested **multiple levels deep**.

```
SELECT emp_name
FROM employee
WHERE dept_id = (
    SELECT dept_id FROM department
    WHERE dept_name = (
        SELECT dept_name FROM department WHERE dept_id = 'D02'
    )
);
```

Each inner subquery executes first, passing its result to the next query.

3. Types of Subqueries

Type	Description	Returns
Single-row Subquery	Returns only one value.	Single value
Multi-row Subquery	Returns multiple values.	Set of values
Correlated Subquery	Depends on outer query for each row.	Dynamic per row

4. Single-row Subqueries

Used with comparison operators like =, >, <, <=, >=.

Example:

```
SELECT emp_name, salary
FROM employee
WHERE salary > (SELECT AVG(salary) FROM employee);
Output:
All employees earning above the company average.
Another Example:
SELECT emp_name, department
FROM employee
WHERE department = (SELECT dept_name FROM department WHERE dept_id =
'D02');
```

5. Multi-row Subqueries

Return multiple rows. Used with operators like IN, ANY, and ALL.

a. IN Operator

Checks if a value matches any from a list.

```
SELECT emp_name, department
FROM employee
WHERE dept_id IN (SELECT dept_id FROM department WHERE dept_name IN
('HR','IT'));
```

b. ANY Operator

Compares a value to *any* value returned by the subquery.

```
SELECT emp_name, salary
```

```
FROM employee
WHERE salary > ANY (SELECT salary FROM employee WHERE department = 'HR');
```

✅ Returns employees whose salary is greater than at least one HR employee.

c. ALL Operator

Compares a value to *all* returned values.

```
SELECT emp_name, salary
FROM employee
WHERE salary > ALL (SELECT salary FROM employee WHERE department = 'HR');
```

✅ Returns employees earning more than *every* HR employee.

6. Correlated Subqueries

A **correlated subquery** runs once for every row in the outer query.
It references columns from the outer query, creating a dependency.

Example

Find employees who earn more than the **average salary of their department**.

```
SELECT e.emp_name, e.department, e.salary
FROM employee e
WHERE e.salary > (
    SELECT AVG(salary)
    FROM employee
    WHERE department = e.department
);
```

Explanation:

- The subquery runs separately for each department in the outer query.
- Each employee's salary is compared with their department's average.

Another Example – Using EXISTS

EXISTS checks whether the subquery returns *any rows* (boolean result).

```
SELECT emp_name
FROM employee e
WHERE EXISTS (
    SELECT * FROM department d
    WHERE e.dept_id = d.dept_id
);
```

✅ Returns employees whose department exists in the department table.

7. Subqueries in Different Clauses

Subqueries can appear in:

- **WHERE** (most common)
- **FROM** (used as a temporary table)
- **SELECT** (for derived values)

Example: Subquery in FROM

```
SELECT department, avg_salary
FROM (
    SELECT department, AVG(salary) AS avg_salary
    FROM employee
    GROUP BY department
) AS dept_avg
WHERE avg_salary > 60000;
```

Explanation:

The inner query creates a temporary table (dept_avg), filtered by the outer query.

8. Subquery vs JOIN

Aspect	Subquery	JOIN
Execution	Inner query runs first	Tables combined before filtering
Readability	Easier for layered logic	Easier for combined datasets
Performance	Sometimes slower	Usually faster
Use Case	When one query depends on result of another	When merging related tables

Hands-On Practice (MySQL)

1. **Create Schema**

```
CREATE DATABASE subquery_demo;
USE subquery_demo;

CREATE TABLE department (
    dept_id CHAR(3) PRIMARY KEY,
    dept_name VARCHAR(30)
```

```
);

CREATE TABLE employee (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id CHAR(3),
    salary DECIMAL(10,2),
    FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);

INSERT INTO department VALUES ('D01','HR'), ('D02','IT'),
('D03','Finance');
INSERT INTO employee VALUES
(1,'Arjun','D01',55000),
(2,'Kavya','D02',70000),
(3,'Meera','D02',65000),
(4,'Divya','D03',52000),
(5,'Rahul','D01',48000);
```

2. **Single-row Subquery**

```
SELECT emp_name
FROM employee
WHERE salary > (SELECT AVG(salary) FROM employee);
```

3. **Multi-row Subquery**

```
SELECT emp_name
FROM employee
WHERE dept_id IN (SELECT dept_id FROM department WHERE dept_name IN
('IT','Finance'));
```

4. **Correlated Subquery**

```
SELECT emp_name, dept_id, salary
FROM employee e
WHERE salary > (SELECT AVG(salary)
                FROM employee
                WHERE dept_id = e.dept_id);
```

Checkpoint Review

1. Define a subquery.
2. Differentiate between single-row and multi-row subqueries.
3. Write a query to find employees earning more than the *maximum HR salary*.
4. How does ANY differ from ALL?
5. Explain correlated subquery with an example.

Reflection Prompt

In a student-course database, how could you use a subquery to find students who scored above the average marks in their course?

Summary

- **Subqueries** let one query depend on the result of another.
- **Single-row subqueries** return one value; **multi-row** return sets.
- Use IN, ANY, and ALL for comparing multiple results.
- **Correlated subqueries** depend on outer query values.
- Subqueries can appear in WHERE, FROM, or SELECT.
- They make queries modular, flexible, and powerful.

3.3 Views

Subtopics

- Concept of Views
- Creating Views
- Modifying Views
- Dropping Views
- Advantages and Limitations

Learning Outcome

After completing this topic, you will be able to:

- Explain what a view is and why it’s used in databases.
- Create, modify, and delete views in MySQL.
- Use views to simplify complex queries and enhance security.

Concept Explanation

A **View** in SQL is a **virtual table** that displays data from one or more base tables.
It doesn't store data physically — it stores a **query definition**, and the result is generated dynamically whenever accessed.

Think of a *view* as a **custom window into your database** — showing only the data you want, the way you want.

1. What is a View?

Definition:

A **View** is a saved SQL query that acts as a virtual table.
It can join multiple tables, include computed columns, or restrict access to sensitive data.

Syntax

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

Example: Basic View

```
CREATE DATABASE view_demo;
USE view_demo;

CREATE TABLE employee (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    department VARCHAR(30),
    salary DECIMAL(10,2)
);

INSERT INTO employee VALUES
(1, 'Arjun', 'IT', 70000),
(2, 'Meera', 'HR', 50000),
(3, 'Kavya', 'Finance', 60000),
(4, 'Divya', 'IT', 65000);
```

Creating a View

```
CREATE VIEW it_employees AS
SELECT emp_name, salary
FROM employee
WHERE department = 'IT';
```

Querying the View

```
SELECT * FROM it_employees;
```

Output:

emp_name	salary
Arjun	70000
Divya	65000

2. Advantages of Views

Advantage	Explanation
Simplifies Complex Queries	You can save frequently used joins or filters.
Data Security	Restricts access to specific columns or rows.
Logical Independence	Base table structure can change without affecting users.
Reusability	Once defined, can be reused across multiple queries.
Data Abstraction	Hides unnecessary details from the end user.

3. Modifying a View

You can **update or redefine** an existing view using the CREATE OR REPLACE VIEW statement.

Syntax

```
CREATE OR REPLACE VIEW view_name AS
SELECT column_list
FROM table_name
WHERE condition;
```

Example

```
CREATE OR REPLACE VIEW it_employees AS
SELECT emp_name, salary, department
FROM employee
WHERE department IN ('IT', 'HR');
```

✅ The view definition is now updated to include HR employees too.

4. Dropping a View

You can remove a view definition using DROP VIEW.

Syntax

```
DROP VIEW view_name;
```

Example:

```
DROP VIEW it_employees;
```

✅ The view is deleted (but base table remains intact).

5. Using Views with Joins

Views can include complex joins and aggregations for simplified access.

Example:

```
CREATE TABLE department (
    dept_id CHAR(3) PRIMARY KEY,
    dept_name VARCHAR(30)
);

ALTER TABLE employee ADD dept_id CHAR(3);
UPDATE employee
SET dept_id = CASE
    WHEN department = 'IT' THEN 'D01'
    WHEN department = 'HR' THEN 'D02'
    WHEN department = 'Finance' THEN 'D03'
END;

CREATE VIEW employee_department AS
SELECT e.emp_name, e.salary, d.dept_name
FROM employee e
JOIN department d ON e.dept_id = d.dept_id;
Now you can access employee details with department names without writing
the join every time:
SELECT * FROM employee_department;
```

6. Updatable Views

Some views allow INSERT, UPDATE, and DELETE operations directly — these are called **updatable views**.

However, a view must follow certain rules:

- The view should reference **only one base table**.
- It must not use aggregate functions (SUM, AVG, etc.).
- It must not use DISTINCT, GROUP BY, UNION, or subqueries.

Example

```
CREATE VIEW hr_employees AS
SELECT emp_id, emp_name, salary
FROM employee
WHERE department = 'HR';
Now, we can insert directly:
INSERT INTO hr_employees VALUES (5, 'Rahul', 48000);
```

✅ Automatically inserted into the base table employee.

7. Read-only Views

Views using aggregations or multiple tables are **read-only** — you cannot directly modify them.

Example:

```
CREATE VIEW salary_summary AS
SELECT department, AVG(salary) AS avg_salary
FROM employee
GROUP BY department;
```

Any attempt to INSERT or UPDATE through this view will throw an error.

8. Limitations of Views

Limitation	Explanation
Performance Overhead	Views execute their query each time they're accessed.
No Indexing (MySQL)	Views don't store data, so they can't have indexes.
Updatable Restrictions	Not all views support DML operations.
Dependency Issues	Dropping base tables breaks dependent views.
Nested Views Complexity	Deeply nested views can reduce readability and performance.

9. System Views

MySQL provides metadata views under the INFORMATION_SCHEMA database.

They store details about databases, tables, columns, and constraints.

Example

```
SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA = 'view_demo';
```

✅ Helps in auditing, maintenance, and system-level insights.

Hands-On Practice (MySQL)

1. **Create Database and Tables**

```
CREATE DATABASE view_practice;
USE view_practice;

CREATE TABLE student (
    roll_no INT PRIMARY KEY,
    name VARCHAR(50),
    dept VARCHAR(30),
    mark INT
);

INSERT INTO student VALUES
(1,'Arjun','CSE',89),
(2,'Kavya','IT',92),
(3,'Meera','ECE',84),
(4,'Rahul','CSE',76);
```

2. **Create and Query a View**

```
CREATE VIEW cse_students AS
SELECT name, mark FROM student WHERE dept = 'CSE';

SELECT * FROM cse_students;
```

3. **Modify and Drop View**

```
CREATE OR REPLACE VIEW cse_students AS
SELECT name, mark FROM student WHERE mark > 80;

DROP VIEW cse_students;
```

Checkpoint Review

1. Define a view.

2. What is the difference between a table and a view?
3. Write SQL to create a view that shows employee names and salaries of the “IT” department.
4. Can all views be updated? Why or why not?
5. Write SQL to drop a view named high_salary_view.

Reflection Prompt

Imagine you’re developing a web portal where students can only see their marks and department name.
How would you use **views** to hide sensitive fields like roll number or total database schema?

Summary

- **Views** are virtual tables derived from SQL queries.
- They enhance **security**, **simplicity**, and **data abstraction**.
- Created with CREATE VIEW, modified with CREATE OR REPLACE VIEW, and removed with DROP VIEW.
- Some are **updatable**, others **read-only** depending on complexity.
- Widely used in real-world applications for dashboards, access control, and modular query management.

3.4 Indexing

Subtopics

- Concept of Indexing
- Clustered vs Non-Clustered Index
- Creating and Dropping Indexes
- Performance Effects and Use Cases

Learning Outcome

After completing this topic, you will be able to:

- Understand what indexes are and how they improve query performance.
- Differentiate between clustered and non-clustered indexes.
- Create, view, and remove indexes in MySQL.
- Apply indexing strategies for performance optimization.

Concept Explanation

An **index** in a database is similar to an **index in a book** — it helps locate information quickly without scanning the entire table.

Technically, an index is a **data structure** (usually a B-tree or hash table) that speeds up retrieval of rows based on specific columns.

However, indexes **consume memory** and **slow down write operations**, so they must be used wisely.

1. What is an Index?

Definition:

An **index** is a database object that improves the **speed of data retrieval** operations on a table at the cost of additional storage and maintenance overhead.

When you query a column with an index, MySQL can locate rows **directly**, instead of scanning every record.

Analogy

Without an index → You read every page of a book to find a topic.
With an index → You check the index section, jump straight to the page.

2. How Indexes Work

Internally, MySQL (InnoDB engine) uses **B-Tree** or **Hash-based** indexes.
Indexes store sorted values of the indexed columns along with pointers to the actual data rows.

When a query uses a WHERE, ORDER BY, or JOIN condition on an indexed column, MySQL uses the index to quickly find matching rows.

3. Types of Indexes

Type	Description	Use Case
Clustered Index	The table data is physically stored in the order of the index key.	Primary keys (default in InnoDB).

Non-Clustered Index	Separate index structure pointing to the table data.	For columns frequently used in search conditions.
Composite Index	Index on multiple columns.	For queries involving multiple conditions.
Unique Index	Ensures all values are unique (like UNIQUE constraint).	On fields like email, username, etc.
Full-text Index	Optimized for searching words in text columns.	For search engines or blog data.
Spatial Index	Used for geometric data types.	GIS applications.

4. Clustered vs Non-Clustered Index

Feature	Clustered Index	Non-Clustered Index
Structure	Reorders physical table data.	Creates separate structure from table.
Count per Table	Only one (usually on Primary Key).	Multiple allowed.
Storage	Stores actual data.	Stores pointers to data.
Speed	Faster for range and key lookups.	Slightly slower, more flexible.
Example	PRIMARY KEY (emp_id)	CREATE INDEX idx_dept ON employee(department)

5. Creating Indexes

a. Single-column Index

```
CREATE INDEX idx_salary
ON employee(salary);
```

b. Composite (Multi-column) Index

```
CREATE INDEX idx_dept_salary
ON employee(department, salary);
```

c. Unique Index

```
CREATE UNIQUE INDEX idx_email
ON employee(email);
```

d. Full-text Index

```
CREATE FULLTEXT INDEX idx_description
```

```
ON products(description);
```

6. Viewing and Dropping Indexes

View Existing Indexes

```
SHOW INDEX FROM employee;
```

Drop Index

```
DROP INDEX idx_salary ON employee;
```

7. Example Scenario

Without Index

```
SELECT * FROM employee WHERE emp_name = 'Arjun';
```

👉 MySQL performs a *full table scan* (checks each row).

With Index

```
CREATE INDEX idx_emp_name ON employee(emp_name);
```

👉 MySQL uses the index to locate 'Arjun' instantly.

8. Performance Analysis

Advantages

✅ Speeds up **SELECT**, **WHERE**, **ORDER BY**, **JOIN** queries.

✅ Improves **search and sorting** efficiency.

✅ Enforces **uniqueness** with unique indexes.

Disadvantages

❌ Consumes extra **disk space**.

❌ Slows down **INSERT**, **UPDATE**, **DELETE** (index must be updated).

❌ Over-indexing can hurt performance.

9. When to Use Indexes

✅ Use indexes on:

- Columns frequently used in WHERE, ORDER BY, or JOIN.
- Columns with high cardinality (many unique values).
- Primary and foreign keys.

🚫 Avoid indexes on:

- Columns frequently updated.
- Columns with few distinct values (like gender or boolean).
- Small tables (where full scan is faster).

10. Practical Example (MySQL)

```
CREATE DATABASE index_demo;
USE index_demo;

CREATE TABLE employee (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    department VARCHAR(30),
    salary DECIMAL(10,2),
    email VARCHAR(100)
);

INSERT INTO employee VALUES
(1,'Arjun','IT',70000,'arjun@xyz.com'),
(2,'Kavya','HR',55000,'kavya@xyz.com'),
(3,'Meera','Finance',60000,'meera@xyz.com'),
(4,'Rahul','IT',68000,'rahul@xyz.com');
```

Create Index

```
CREATE INDEX idx_dept ON employee(department);
CREATE UNIQUE INDEX idx_email ON employee(email);
```

Query Optimization Example

```
EXPLAIN SELECT * FROM employee WHERE department = 'IT';
```

✅ Shows that MySQL uses idx_dept to optimize retrieval.

Drop Index

```
DROP INDEX idx_dept ON employee;
```

11. Understanding EXPLAIN (Performance Insight)

EXPLAIN shows how MySQL executes your query — including which indexes are used.

Example:

```
EXPLAIN SELECT * FROM employee WHERE emp_name = 'Arjun';
Key Columns in Output:
  • key → which index is used.
  • rows → estimated number of rows scanned.
  • type → access type (ALL, INDEX, REF, CONST).
```

✅ Lower row count = better performance.

12. Best Practices

1. Index **primary & foreign keys**.
2. Avoid redundant or overlapping indexes.
3. Use **composite indexes** for multi-condition queries.
4. Monitor queries using EXPLAIN ANALYZE.
5. Rebuild or drop unused indexes periodically.

Hands-On Practice (MySQL)

1. **Create Table and Insert Data**

```
CREATE DATABASE index_practice;
USE index_practice;

CREATE TABLE sales (
    sale_id INT PRIMARY KEY AUTO_INCREMENT,
    product VARCHAR(50),
    price DECIMAL(10,2),
    quantity INT
);

INSERT INTO sales (product, price, quantity)
VALUES ('Laptop', 55000, 3),
       ('Mouse', 700, 10),
       ('Keyboard', 1500, 5),
       ('Laptop', 56000, 2);
```

2. **Create Index**

```
CREATE INDEX idx_product ON sales(product);
```

3. **Compare Query Speeds**

```
EXPLAIN SELECT * FROM sales WHERE product = 'Laptop';
```

4. **Drop Index**

```
DROP INDEX idx_product ON sales;
```

Checkpoint Review

1. What is an index, and why is it used?
2. Differentiate between clustered and non-clustered indexes.
3. Write SQL to create a unique index on the email column.
4. What are the pros and cons of indexing?
5. How can you identify which index a query uses?

Reflection Prompt

If your company's orders table has millions of records, how would you decide which columns to index for better performance without overloading the system?

Summary

- Indexes improve query performance by reducing disk I/O.
- **Clustered Index**: data physically ordered; only one per table.
- **Non-Clustered Index**: separate index structure pointing to rows.
- Proper indexing strategy enhances performance but over-indexing can degrade it.
- Use EXPLAIN to verify index usage and optimize queries.

3.5 Transactions

Subtopics

- Concept of Transactions
- ACID Properties
- COMMIT, ROLLBACK, SAVEPOINT
- Concurrency Control

Learning Outcome

After completing this topic, you will be able to:

- Explain what transactions are and why they are essential.
- Understand and apply ACID properties in MySQL.
- Use COMMIT, ROLLBACK, and SAVEPOINT effectively.
- Handle concurrent access safely using transaction control commands.

Concept Explanation

A **transaction** is a **logical unit of work** that contains one or more SQL statements which must all succeed or fail together.

It ensures that the database remains **consistent and reliable**, even if system failures, errors, or interruptions occur.

Example:

In a banking system — if you transfer ₹10,000 from Account A to Account B:

1. Debit ₹10,000 from A.
2. Credit ₹10,000 to B.

Both steps must complete successfully, or neither should occur — this is handled by **transactions**.

1. What is a Transaction?

Definition:

A **transaction** is a sequence of one or more SQL operations executed as a single unit, ensuring that either **all changes are committed** or **none are applied**.

2. Transaction Lifecycle

1. **Begin Transaction** → Start a logical unit of work.
2. **Perform Operations** → Execute SQL commands.
3. **Commit or Rollback** → Save or undo changes.

Flow Diagram

START TRANSACTION

↓

Perform SQL Operations

↓

If success → COMMIT

If failure → ROLLBACK

3. ACID Properties of Transactions

The foundation of reliable transaction management is defined by the **ACID** properties:

Property	Meaning	Example
A - Atomicity	All operations complete or none do.	Debit and Credit must both succeed.
C - Consistency	Database moves from one valid state to another.	Balances always remain accurate.
I - Isolation	Transactions execute independently of others.	Two users' transfers don't conflict.
D - Durability	Changes persist even after system failure.	Committed data is stored permanently.

4. Transaction Control Commands

Command	Purpose
START TRANSACTION or BEGIN	Begins a new transaction.
COMMIT	Saves all changes made in the transaction.
ROLLBACK	Undoes changes made since the transaction began.
SAVEPOINT	Sets a checkpoint inside a transaction.
RELEASE SAVEPOINT	Deletes a defined savepoint.
SET AUTOCOMMIT	Enables/disables automatic commits.

5. Example: Basic Transaction

```
CREATE DATABASE transaction_demo;
USE transaction_demo;

CREATE TABLE accounts (
    acc_no INT PRIMARY KEY,
    holder_name VARCHAR(50),
    balance DECIMAL(10,2)
);

INSERT INTO accounts VALUES
(101, 'Arjun', 50000.00),
```

```
(102, 'Kavya', 30000.00);
```

Transaction Example

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 10000 WHERE acc_no = 101;
UPDATE accounts SET balance = balance + 10000 WHERE acc_no = 102;
```

COMMIT;

✅ Both updates succeed → Changes are permanently saved.

If there's an issue (e.g., power loss or error):

ROLLBACK;

✅ Reverts all changes to the previous state.

6. Using SAVEPOINT

A **SAVEPOINT** marks a partial checkpoint within a transaction — useful when part of the transaction may need to be rolled back without undoing everything.

Example

```
START TRANSACTION;

UPDATE accounts SET balance = balance - 5000 WHERE acc_no = 101;
SAVEPOINT step1;

UPDATE accounts SET balance = balance + 5000 WHERE acc_no = 102;
SAVEPOINT step2;

-- Suppose an error occurs here
ROLLBACK TO step1;
COMMIT;
```

✅ Only the first update (step1) remains; the second update is undone.

7. Disabling Auto-Commit

By default, MySQL commits every statement automatically.

You can disable this to manage transactions manually.

```
SET AUTOCOMMIT = 0;
```

```
START TRANSACTION;
UPDATE accounts SET balance = balance + 2000 WHERE acc_no = 102;
COMMIT;

SET AUTOCOMMIT = 1;
```

8. Concurrency Control

When multiple users access the same data simultaneously, **isolation** ensures they don't interfere with each other.

Problems Without Isolation

Issue	Description	Example
Dirty Read	One transaction reads uncommitted data from another.	A reads temporary data from B's uncommitted update.
Non-Repeatable Read	Same query returns different results within one transaction.	A reads same row twice but sees updated value.
Phantom Read	New rows added by another transaction appear unexpectedly.	A reads 10 rows, another inserts 2 new rows mid-transaction.

Isolation Levels

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read	Use Case
READ UNCOMMITTED	✅	✅	✅	Fastest, least safe
READ COMMITTED	❌	✅	✅	Default in Oracle
REPEATABLE READ	❌	❌	✅	Default in MySQL

SERIALIZABLE	❌	❌	❌	Safest, slowest

Set Isolation Level

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
-- SQL operations
COMMIT;
```

9. Practical Example – Safe Fund Transfer

```
SET AUTOCOMMIT = 0;
START TRANSACTION;

UPDATE accounts SET balance = balance - 10000 WHERE acc_no = 101;
UPDATE accounts SET balance = balance + 10000 WHERE acc_no = 102;

-- Simulate an error
-- ROLLBACK;
```

```
COMMIT;
SET AUTOCOMMIT = 1;
```

✅ Ensures that both debit and credit happen together or not at all.

10. Checking Transaction State

You can check whether a transaction is active or committed:

```
SHOW VARIABLES LIKE 'autocommit';
SELECT @@tx_isolation;
```

11. Best Practices

1. Always use **transactions** for multi-step updates.
2. Use **SAVEPOINTS** for partial rollbacks.
3. Keep transactions **short and fast**.
4. Handle exceptions in application code for rollbacks.
5. Use appropriate **isolation levels** for balancing consistency and performance.

Hands-On Practice (MySQL)

1. **Setup**

```
CREATE DATABASE bank_system;
USE bank_system;

CREATE TABLE accounts (
    acc_no INT PRIMARY KEY,
    name VARCHAR(50),
    balance DECIMAL(10,2)
);

INSERT INTO accounts VALUES
(101, 'Meera', 80000),
(102, 'Rahul', 45000);
```

2. **Transaction Test**

```
SET AUTOCOMMIT = 0;
START TRANSACTION;

UPDATE accounts SET balance = balance - 5000 WHERE acc_no = 101;
UPDATE accounts SET balance = balance + 5000 WHERE acc_no = 102;

COMMIT;
```

3. **Rollback Scenario**

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 90000 WHERE acc_no = 102; --
invalid
ROLLBACK;
```

Checkpoint Review

1. Define a transaction and its purpose.
2. Explain each ACID property with an example.

3. Write SQL commands to transfer funds safely between two accounts.
4. What is the role of SAVEPOINT?
5. How do isolation levels affect performance and consistency?

Reflection Prompt

If you're designing an **e-commerce checkout**, how would transactions ensure that the product stock, payment, and order confirmation happen atomically and safely?

Summary

- A **transaction** groups multiple operations into a single logical unit.
- **ACID** ensures reliability and consistency of data.
- COMMIT saves changes; ROLLBACK undoes them.
- SAVEPOINT allows partial rollbacks.
- **Concurrency control** ensures isolation in multi-user systems.
- Proper transaction design guarantees data safety in critical operations like banking, e-commerce, and billing systems.

3.6 Cursors

Subtopics

- Concept of Cursors
- Syntax and Working
- Cursor Lifecycle (Declare → Open → Fetch → Close)
- Iterating Over Results in MySQL
- Practical Use Cases

Learning Outcome

After completing this topic, you will be able to:

- Understand what cursors are and why they're used.
- Write MySQL code to declare, open, fetch, and close cursors.
- Use cursors to process query results row by row.
- Apply cursors effectively in stored procedures for data processing.

Concept Explanation

A **cursor** in SQL is a **pointer** that allows you to **iterate through query results one row at a time**.

Normally, SQL operates in **set-based mode** — it processes entire result sets at once.
Cursors switch this to **row-by-row (procedural)** processing — essential for logic that depends on individual rows.

Analogy

Think of a cursor like a reading pointer in a text file —
Instead of reading the whole file at once, you move line by line, process each, and then continue.

1. When to Use Cursors

✅ **Use Cursors When:**

- You need to perform operations that depend on results of previous rows.
- You're migrating data between tables dynamically.
- You're performing **procedural logic** within stored programs.

🚫 **Avoid Cursors When:**

- A single SQL query can solve the problem (set-based is always faster).
- Data volume is very large (cursors are slower due to iterative nature).

2. Cursor Syntax and Lifecycle

The lifecycle of a cursor follows four main steps:

Step	Command	Purpose
1. Declare	DECLARE cursor_name CURSOR FOR SELECT query	Define the cursor.
2. Open	OPEN cursor_name;	Execute the SELECT and prepare result set.
3. Fetch	FETCH cursor_name INTO variable_list;	Retrieve one row at a time.
4. Close	CLOSE cursor_name;	Release resources.

3. Basic Cursor Example

Setup

```
CREATE DATABASE cursor_demo;
USE cursor_demo;
```

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    salary DECIMAL(10,2)
);

INSERT INTO employees VALUES
(1,'Arjun',70000),
(2,'Kavya',55000),
(3,'Meera',60000);
```

Cursor Example in MySQL Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE show_salaries()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE e_name VARCHAR(50);
    DECLARE e_salary DECIMAL(10,2);

    DECLARE emp_cursor CURSOR FOR
        SELECT emp_name, salary FROM employees;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN emp_cursor;

    read_loop: LOOP
        FETCH emp_cursor INTO e_name, e_salary;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SELECT CONCAT('Employee: ', e_name, ' | Salary: ', e_salary) AS
Info;
    END LOOP;

    CLOSE emp_cursor;
END $$

DELIMITER ;
```

Execute the Procedure:

```
CALL show_salaries();
```

Output:

```
Employee: Arjun | Salary: 70000
Employee: Kavya | Salary: 55000
Employee: Meera | Salary: 60000
```

4. Cursor Explanation (Step by Step)

1. **DECLARE CURSOR** → Defines the result set to iterate over.
2. **DECLARE HANDLER** → Defines what happens when no more rows remain.
3. **OPEN** → Executes the query and prepares results.
4. **FETCH** → Retrieves one row into declared variables.
5. **LOOP + IF done** → Continues until all rows are fetched.
6. **CLOSE** → Frees memory and releases the cursor.

5. Using SAVEPOINT and Cursors Together (Advanced)

```
DELIMITER $$

CREATE PROCEDURE increase_salaries()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE e_id INT;
    DECLARE e_salary DECIMAL(10,2);
    DECLARE emp_cursor CURSOR FOR SELECT emp_id, salary FROM employees;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    START TRANSACTION;
    OPEN emp_cursor;

    read_loop: LOOP
        FETCH emp_cursor INTO e_id, e_salary;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SAVEPOINT before_update;
        UPDATE employees SET salary = e_salary * 1.1 WHERE emp_id = e_id;
    END LOOP;

    CLOSE emp_cursor;
```

```
    COMMIT;
END $$

DELIMITER ;
```

Call Procedure:

```
CALL increase_salaries();
SELECT * FROM employees;
```

✅ Each salary is increased by 10%, one row at a time.

6. Cursors with Conditional Logic

You can embed IF or CASE conditions inside cursor loops.

Example:

```
DELIMITER $$

CREATE PROCEDURE adjust_salary()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE e_id INT;
    DECLARE e_salary DECIMAL(10,2);
    DECLARE emp_cursor CURSOR FOR SELECT emp_id, salary FROM employees;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN emp_cursor;

    read_loop: LOOP
        FETCH emp_cursor INTO e_id, e_salary;
        IF done THEN
            LEAVE read_loop;
        END IF;

        IF e_salary < 60000 THEN
            UPDATE employees SET salary = e_salary + 5000 WHERE emp_id =
e_id;
        ELSE
            UPDATE employees SET salary = e_salary + 2000 WHERE emp_id =
e_id;
        END IF;
    END LOOP;
```

```
    CLOSE emp_cursor;
END $$
```

DELIMITER ;

✅ Gives higher raises to lower-salaried employees.

7. Limitations of Cursors

Limitation	Description
Performance Overhead	Slower than set-based queries.
Resource Usage	Requires memory and CPU for each row processed.
Complexity	Harder to maintain than regular queries.
Locking Issues	Can lead to row locks if not handled properly.

8. Alternatives to Cursors

- **Set-based operations:** Use UPDATE, INSERT INTO ... SELECT, or CASE WHEN queries.
- **Window functions:** Use ROW_NUMBER(), RANK(), or aggregates.
- **Temporary tables:** For step-wise transformations.

Example (Set-Based Alternative):

UPDATE employees

SET salary = salary + 5000

WHERE salary < 60000;

✅ Achieves same result as a cursor — faster and cleaner.

Hands-On Practice (MySQL)

1. **Create and Populate Table**

```
CREATE TABLE sales (
    sale_id INT PRIMARY KEY AUTO_INCREMENT,
    product VARCHAR(50),
    quantity INT
);
```

```
INSERT INTO sales (product, quantity) VALUES
('Laptop', 3),
('Mouse', 10),
('Keyboard', 5);
```

2. **Cursor to Process Rows**

```
DELIMITER $$

CREATE PROCEDURE show_sales()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE p_name VARCHAR(50);
    DECLARE p_qty INT;
    DECLARE sale_cursor CURSOR FOR SELECT product, quantity FROM sales;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN sale_cursor;
    read_loop: LOOP
        FETCH sale_cursor INTO p_name, p_qty;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SELECT CONCAT(p_name, ' - Quantity: ', p_qty) AS sale_info;
    END LOOP;
    CLOSE sale_cursor;
END $$
DELIMITER ;
CALL show_sales();
```

Checkpoint Review

1. What is a cursor in SQL?
2. Name the four steps of cursor lifecycle.
3. Write a simple cursor to display each employee's name and salary.
4. Why are cursors slower than regular SQL queries?
5. What can you use instead of a cursor to process data in bulk?

Reflection Prompt

Imagine you are automating salary updates based on department and performance rating. Would you use a cursor or a set-based query? Why?

Summary

- **Cursors** allow row-by-row processing of query results.
- Lifecycle: **DECLARE → OPEN → FETCH → CLOSE**.
- Ideal for **procedural logic** in stored programs.
- Can include **conditions**, **transactions**, and **savepoints**.
- Use with caution — prefer **set-based solutions** for performance.