MySQL — Week 2: Advanced Notes & Examples

Table of Contents

---

1. Subqueries

A **subquery** is a query inside another query. It is useful for breaking down complex problems.

Nested Subquery

Executed once and passed to the outer query.

```
SELECT name
FROM students
WHERE id IN (
  SELECT student_id
  FROM marks
  WHERE score > 80
);
```

Correlated Subquery

Depends on the outer query and executes for each row.

```
SELECT s.name, s.id
FROM students s
WHERE score > (
  SELECT AVG(score)
  FROM marks m
  WHERE m.student_id = s.id
);
```

---

2. UNION and UNION ALL

UNION

- Combines results of two queries.
- Removes duplicates.

```
SELECT city FROM customers
UNION
SELECT city FROM suppliers;
```

UNION ALL

- Combines results but keeps duplicates.

```
SELECT city FROM customers
UNION ALL
SELECT city FROM suppliers;
```

Key Notes:

- Both queries must have the same number of columns.
- Data types must be compatible.

---

3. Stored Procedure

A **stored procedure** is a precompiled set of SQL statements stored in the database.
You can call it whenever needed, instead of writing the same SQL multiple times.

✅ **Benefits**

- Code reusability
- Better performance (precompiled)
- Security (restrict direct table access, allow only procedures)

✅ Syntax

```
DELIMITER $$
CREATE PROCEDURE procedure_name(parameter_list)
BEGIN
  -- SQL statements
END $$
DELIMITER ;
```

✅ Example 1: Simple Procedure

Create a procedure to fetch all students:

```
DELIMITER $$
CREATE PROCEDURE get_students()
BEGIN
  SELECT * FROM students;
END $$
DELIMITER ;
```

✅ Example 2: Procedure with Parameters

```
DELIMITER $$

CREATE PROCEDURE GetStudentsByDept(IN deptName VARCHAR(50))
BEGIN
   SELECT * FROM Students WHERE department = deptName;
END $$

DELIMITER ;
```

✅ Example 3: Procedure with Parameters

```
DELIMITER $$

CREATE PROCEDURE CountStudentsByDept(IN deptName VARCHAR(50), OUT total INT)
BEGIN
   SELECT COUNT(*) INTO total
   FROM Students
   WHERE department = deptName;
END $$

DELIMITER ;
```

Parameters

- **IN** → Input value.
- **OUT** → Return value.
- **INOUT** → Both input and output.

---

4. Triggers

A **trigger** is a special database object that is automatically executed (fires) when a specific **event** (INSERT, UPDATE, DELETE) happens on a table.

✅ Benefits

- Maintain data integrity
- Automatically log changes
- Enforce business rules

Syntax

```
DELIMITER $$
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON table_name
FOR EACH ROW
BEGIN
  -- actions
END $$
DELIMITER ;
```

✅ Example 1: Insert Trigger (Audit Log)

Suppose we have an `AuditLog` table:

```
CREATE TABLE AuditLog (
   log_id INT AUTO_INCREMENT PRIMARY KEY,
   action VARCHAR(50),
   student_id INT,
   action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

DELIMITER $$

CREATE TRIGGER after_student_insert
AFTER INSERT ON Students
FOR EACH ROW
BEGIN
   INSERT INTO AuditLog(action, student_id)
   VALUES ('INSERT', NEW.student_id);
END $$

DELIMITER ;
```

Now, whenever you insert into `Students`, an entry is added to `AuditLog`.

- **NEW** → Refers to new row values.

- **OLD** → Refers to existing row values.

---

5. Functions

A **function** in MySQL is similar to a procedure but:

- Always returns a **single value** (mandatory).
- Can be used inside SQL queries (unlike procedures).
- Cannot perform INSERT/UPDATE/DELETE (read-only).

Syntax

```
DELIMITER $$
CREATE FUNCTION function_name(parameters)
RETURNS datatype
DETERMINISTIC
BEGIN
  -- SQL statements
  RETURN value;
END $$
DELIMITER ;
```

✅ Example 1: Simple Function

Function to calculate square of a number:

```
DELIMITER $$

CREATE FUNCTION SquareNum(x INT)
RETURNS INT
DETERMINISTIC
BEGIN
   RETURN x * x;
END $$
```

DELIMITER ;

HOW TO USE IT:
SELECT SquareNum(5); -- Output: 25

Feature	Stored Procedure	Trigger	Function
Return Value	Optional (IN/OUT parameters)	No return	Must return a single value
Invocation	Called manually (`CALL`)	Fires automatically on event	Called inside SQL queries
Operations allowed	Can perform DML (INSERT/UPDATE/DELETE)	Executes automatically on table changes	Cannot perform DML, only calculations/returns
Use Case	Reusable business logic	Data integrity, audit logging	Calculations, reusable formulas

---

6. Views

What is a view?

A **view** is a named `SELECT` query stored in the database as a virtual table. When you `SELECT` from a view, MySQL runs the underlying `SELECT` and returns the result. Views:

- simplify complex queries,
- provide column/table abstraction and security (limit what users can see),
- can be reused and nested (views on views).

> A view is *virtual* — it does not store rows by default (MySQL does not have built-in materialized views). The result is generated when you query the view.

Basic syntax

```
CREATE [OR REPLACE]
  [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
  [DEFINER = user]
  [SQL SECURITY {DEFINER | INVOKER}]
VIEW view_name [(col1, col2, ...)]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION];
```

Create a View

```
CREATE VIEW v_emp_basic AS
SELECT emp_id, name, dept_id, salary
FROM employees;
```

Using a View

```
SELECT * FROM v_emp_basic ;
```

Updating Through Views

```
UPDATE v_emp_basic SET salary= 95000 WHERE name = 'John';
```

Dropping a View

```
DROP VIEW v_emp_basic ;
```

---

7. Backup And Restore methods.

Backup → making a copy of your database data so that if the original database is lost, corrupted, or accidentally deleted, you can get it back.

Restore → the process of putting that backup copy back into MySQL so the database becomes usable again.

🔹 Why do we need Backup & Restore?

1. **Data loss prevention** → accidental `DELETE` or corruption.
2. **Hardware/software failure** → if your server crashes.
3. **Migration** → moving a database from one server to another.
4. **Testing** → copy production DB into test environment.

Without backups, once data is gone, it's **gone forever**.

🔹 Types of Backups in MySQL

There are two basic ways:

1) Logical Backup

- Saves **SQL commands** (like `CREATE TABLE`, `INSERT INTO ...`) that can rebuild the database.
- Tool: **`mysqldump`** (most common).
- Example of backup file:

```
CREATE DATABASE company;

USE company;

CREATE TABLE employees(id INT, name VARCHAR(50));

INSERT INTO employees VALUES (1, 'Shahil'), (2, 'Neha');
```

- To restore → run this SQL file back into MySQL.

2) Physical Backup

- Copies the **actual database files** (the `.ibd`, `.frm`, etc. stored on disk).
- Faster for large databases.
- Tools: **XtraBackup**, snapshots, filesystem copy.

🔹 The Easiest Way: Using `mysqldump`

Step 1: Backup (export)

Suppose you have a database `school`. Run:

```
mysqldump -u root -p school > school_backup.sql
```

👉 This creates a file `school_backup.sql` containing all tables and data.

Step 2: Restore (import)

If `school` database got deleted, you can restore it:

```
mysql -u root -p < school_backup.sql
```

👉 This re-creates the database with the original data.

🔹 Small Example (Full Cycle)

1. Create sample DB

```
CREATE DATABASE school;

USE school;


CREATE TABLE students (

  id INT PRIMARY KEY,

  name VARCHAR(50),

  age INT

);


INSERT INTO students VALUES (1,'Amit',20),(2,'Neha',21);
```

2. Take Backup

```
mysqldump -u root -p school > school_backup.sql
```

3. Drop the database (simulate loss)

```
DROP DATABASE school;
```

4. Restore from backup

```
mysql -u root -p < school_backup.sql
```

5. Verify

```
USE school;

SELECT * FROM students;
```

👉 `Amit` and `Neha` are back.

🔹 What is `EXPLAIN`?

- `EXPLAIN` is a **diagnostic tool** in MySQL used before a `SELECT`, `DELETE`, `INSERT`, `UPDATE`, or `REPLACE` statement.
- It shows **how MySQL executes your query** — i.e., how it retrieves the rows, which indexes are used, join types, etc.
- This helps you identify **slow queries** and optimize them.

👉 In short: `EXPLAIN` tells you *what MySQL plans to do* when executing your query.

🔹 Basic Syntax

EXPLAIN SELECT * FROM students WHERE age > 20;

🔹 Columns in `EXPLAIN` Output

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	students	range	age	age	4	NULL	2	Using where