WEEK 2

SUBQUERY

A **subquery** is simply a query written inside another SQL query. It is often used when we need the result of one query to be used by another. Subqueries are enclosed in parentheses and can appear in different clauses such as `WHERE`, `FROM`, or `SELECT`. For example, finding students who scored above the class average can be done using a subquery that first calculates the average and then compares each student's marks with it.

NESTED QUERY

It means a query placed inside another query, often to break down a complex task into smaller steps.

```
SELECT name
FROM students
WHERE marks > (SELECT AVG(marks) FROM students);
```

CORRELATED QUERY

A **special type of subquery** that depends on the outer query for its values. Executes **row by row** with the main query. It is usually slower than normal subqueries.

```
-- students having score greater than the average
SELECT s1.name, s1.marks
FROM students s1
WHERE s1.marks > (
   SELECT AVG(s2.marks)
   FROM students s2
   WHERE s1.class = s2.class
);
```

- **Subquery/Nested Query** → Query inside another query.
- **Correlated Subquery** → Inner query depends on outer query values.

•

UNION & UNION ALL

The **UNION** operator in SQL is used to combine the results of two or more `SELECT` statements into a single result set. By default, `UNION` removes duplicate rows, so the final output only contains unique records from all the queries combined. For this to work, each `SELECT` must have the same number of columns with compatible data types.

On the other hand, **UNION ALL** also combines results of multiple `SELECT` queries, but it does **not remove duplicates**. It simply appends the results together, including any repeating rows. Because it skips the extra step of checking for duplicates, `UNION ALL` is usually faster than `UNION`.

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

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

Difference between union and union all

Feature	UNION	UNION ALL
Duplicates	Removes duplicates	Keeps duplicates
Performance	Slightly slower (due to duplicate check)	Faster (no duplicate check)
Use Case	When only unique results are needed	When duplicates are acceptable/needed

TRIGGERS

A **trigger** is a special kind of stored program in SQL that automatically executes (fires) in response to certain events on a table or a view. Triggers are mainly used to maintain data integrity, enforce business rules, or perform automatic actions such as logging changes.

Triggers can be set to fire on events like **INSERT**, **UPDATE**, or **DELETE**. Depending on when they are executed, triggers are classified as **BEFORE** triggers (executed before the event) or **AFTER** triggers (executed after the event).

SYNTAX

```
CREATE TRIGGER trigger_name
BEFORE | AFTER INSERT | UPDATE | DELETE
ON table_name
FOR EACH ROW
BEGIN
  -- SQL statements to execute
END;
```

Suppose we have an `employees` table and we want to **keep track of any deletions** in a separate table `emp_log`. We can create a trigger like this:

```
CREATE TRIGGER after_employee_delete
AFTER DELETE
ON employees
FOR EACH ROW
BEGIN
  INSERT INTO emp_log(emp_id, action_time, action)
  VALUES (OLD.employee_id, NOW(), 'DELETED');
END;
```

Types of Triggers

1. **BEFORE Trigger** – Executes before an `INSERT`, `UPDATE`, or `DELETE`. Often used for validation.
2. **AFTER Trigger** – Executes after the operation. Commonly used for logging or audit purposes.

Advantages of Triggers

- Enforces rules automatically.
- Helps maintain audit logs without manual effort.

- Ensures consistency between related tables.

Disadvantages of Triggers

- Can make debugging harder since actions happen automatically.
- Too many triggers may slow down performance.

STORED PROCEDURE

A **stored procedure** is a group of SQL statements that are stored in the database and can be executed as a single unit. Instead of writing the same SQL queries again and again, you can save them as a procedure and just call it whenever needed.

Stored procedures are mainly used for **code reusability, modularity, security, and performance**. They can accept **parameters** (input/output) and return results, which makes them very flexible.

We can think of SP as functions in any coding language.

Example

Suppose we want a procedure to get all employees from a specific department:

```
DELIMITER//
CREATE PROCEDURE getEmployeesByDept(IN deptId INT)
BEGIN
  SELECT * FROM employees WHERE department_id = deptId;
END//
DELIMITER;
```

To execute:

```
CALL getEmployeesByDept(2);
```

In MySQL, the **default delimiter** (end of a statement) is the semicolon `;`.

When writing **stored procedures, functions, or triggers**, the code block itself may contain multiple SQL statements ending with `;`. This confuses MySQL, because it thinks the procedure ends after the first `;`.

To solve this, we use the **DELIMITER** command to temporarily change the statement terminator to something else (like `//` or `$$`). This way, MySQL

understands where the procedure starts and ends.

STORED PROCEDURE VS FUNCTIONS

A **stored procedure** is like a program that can perform multiple tasks such as inserting, updating, or deleting data, as well as returning results. It can accept input and output parameters but does not necessarily return a value.

A **function**, on the other hand, always returns a single value (scalar) or a table. Functions are usually used in calculations, queries, and expressions where a return value is required. Unlike procedures, functions cannot modify database data (no `INSERT`, `UPDATE`, `DELETE` allowed in most SQL systems).

CURSORS

A **cursor** is a database object used to **fetch and process query results row by row**. Normally, SQL works with entire sets of rows at once (set-based operations), but sometimes we need to handle each row individually — in such cases, cursors are useful.

Cursors are commonly used inside **stored procedures, functions, or triggers** when row-by-row operations are required (though they are slower compared to set-based queries).

Steps to Use a Cursor

1. **Declare** the cursor – Define it with a `SELECT` query.
2. **Open** the cursor – Execute the query and make the result set available.
3. **Fetch** rows – Retrieve one row at a time.
4. **Close** the cursor – Release the result set.
5. (Optional) **Deallocate** – Remove the cursor definition from memory.

Example

Suppose we want to display employee names one by one:

```
DECLARE emp_cursor CURSOR FOR
SELECT name FROM employees;

OPEN emp_cursor;
```

```
FETCH NEXT FROM emp_cursor INTO @emp_name;

WHILE @@FETCH_STATUS = 0
BEGIN
  PRINT @emp_name;
  FETCH NEXT FROM emp_cursor INTO @emp_name;
END;

CLOSE emp_cursor;
DEALLOCATE emp_cursor;
```

Advantages

- Allows row-by-row processing.

Disadvantages

- Slower compared to set-based SQL queries.
- Uses more memory and resources.

DETERMINISTIC VS NON_DETERMINISTIC

Deterministic → For the **same input**, the function **always returns the same result**.

Non-deterministic → The result **can change even for the same input** (e.g., uses NOW(), RAND(), UUID()).- used by default

VIEWS

A **view** in SQL is a virtual table that is based on the result of a query. Unlike a physical table, it does not store data itself but instead provides a way to look at data stored in one or more tables. Views are created to simplify complex queries, improve security by restricting access to certain columns or rows, and provide a consistent way of presenting data. Since views behave like tables, users can query them just like they query normal tables, although the underlying data always comes from the base tables.

```
CREATE VIEW employee_dept AS
SELECT e.emp_id, e.emp_name, d.dept_name
FROM employees e
JOIN department d ON e.dept_id = d.dept_id;
```

Here, employee_dept becomes a view that combines employee details with their department. This way, a user doesn't need to write the join every time; they can just use:

```
SELECT * FROM employee_dept;
```

Views can also be updated, but only under certain conditions, such as when they are based on a single table without aggregate functions or GROUP BY. They are also useful in restricting access—for example, if a company wants to show employees' names and departments but not their salaries, a view can hide sensitive columns.
To use or modify views, SQL provides commands like ALTER VIEW to change the query inside a view, and DROP VIEW to remove it.

Indexing

An **index** in a database works like the index in a book – it allows faster retrieval of rows from a table without scanning the whole table. Instead of checking every row (full table scan), the database jumps directly to the required data using indexing structures such as **B-Trees** or **Hash Maps**.

- **Without Index:** Database performs `ALL` scan → checks every row (slow for large tables).
- **With Index:** Database performs **ref lookup** using index (logarithmic search with B-Tree or hashing) → very fast.

Note: In SQL, creating a **Primary Key** automatically creates a **B-Tree index**.

Indexing Structures

- **B-Tree Index:** Balanced tree, sorted, allows efficient searching (`O(log n)`).
- **Hash Index:** Uses hashing for equality lookups (not good for ranges).
- **Full-text Index:** Specialized structure for searching words/phrases.

Types of Index

1. **Primary Key Index**
   - Created automatically when `PRIMARY KEY` is defined.
   - Ensures uniqueness and no NULLs.
2. **Unique Index**
   - Ensures uniqueness on non-primary columns.
   - Allows one or more `NULL` values.
3. **Normal Index (Non-Unique Index)**
   - Speeds up searches on `WHERE`, `ORDER BY`, `JOIN`.
   - Allows duplicates.
4. **Composite Index**
   - Index on two or more columns together.
   - Useful when queries filter using multiple columns.
5. **Full-Text Index**
   - For text searching in large text fields.
   - Works with `MATCH() ... AGAINST()`.
6. **Spatial Index**
   - Used for geographic/location data.
   - Helpful in GIS (Geographic Information Systems).

Feature	Clustered Index	Non-Clustered Index
Data storage	Stores table data in sorted order of index.	Stores pointers (row locators) to data.
No. per table	Only **one** per table.	Multiple allowed.
Speed	Faster for range queries.	Slower than clustered for sequential data.
Example	Primary Key index.	Secondary indexes.

EXPLAIN

The `EXPLAIN` keyword is used to **analyze how MySQL executes a query**. It helps understand the query execution plan, which is useful for **optimizing performance** and checking whether indexes are being used effectively.

```
EXPLAIN SELECT * FROM employees WHERE dept_id = 2;
```

MySQL returns a table with information about how it retrieves data.

Important Columns in EXPLAIN

- **id** → Identifier for each SELECT query in a complex query.
- **select_type** → Type of SELECT (`SIMPLE`, `SUBQUERY`, `DERIVED`, `UNION`, etc.).
- **table** → Name of the table (or alias) being accessed.
- **type** → How rows are accessed; shows efficiency:
  - `system` → One row only
  - `const` → At most one row using primary/unique key
  - `eq_ref` → One row per previous table row via unique index
  - `ref` → Index lookup returning many rows
  - `range` → Index range scan (`>`, `<`, `BETWEEN`)
  - `index` → Full index scan
  - `ALL` → Full table scan (least efficient)
- **key** → Name of the index MySQL uses.
- **key_len** → Length of the index used (in bytes).
- **ref** → Shows what column or value is compared with the index.
- **rows** → Estimated number of rows MySQL will examine.
- **Extra** → Additional info like `Using where`, `Using index`, `Using temporary`.

WINDOW FUNCTIONS

Window functions are special SQL functions that perform calculations across a **set of rows related to the current row**, without collapsing the result into a single output (unlike aggregate functions). They are very useful for ranking,

running totals, moving averages, and other calculations that require context of surrounding rows.

Window functions are used with the `OVER()` clause, which defines the **window (set of rows)** for the function to operate on. You can also **partition** the data (like grouping) and **order** it within each partition.

Common Window Functions

1. **ROW_NUMBER()** – Assigns a unique sequential number to each row in a partition.
2. **RANK()** – Assigns rank to rows in a partition, with gaps for ties.
3. **DENSE_RANK()** – Similar to `RANK()` but no gaps in ranks for ties.
4. **SUM(), AVG(), MIN(), MAX()** – Aggregate functions used as window functions to calculate running totals, moving averages, etc.
5. **LEAD() / LAG()** – Accesses data from the following or preceding row.

Syntax Example

Suppose we have an `employees` table with `dept_id` and `salary`:

```
SELECT emp_name,
    dept_id,
    salary,
    ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DES
C) AS row_num,
    AVG(salary) OVER (PARTITION BY dept_id) AS avg_salary
FROM employees;
```

- `PARTITION BY dept_id` → Calculates row numbers and average salary **within each department**.
- `ORDER BY salary DESC` → Orders rows by salary in descending order for `ROW_NUMBER()`.
- `AVG(salary)` → Shows department-wise average salary next to each row without collapsing the data.

Use Cases of Window Functions

- Ranking employees by salary within departments (`ROW_NUMBER`, `RANK`).
- Calculating running totals or cumulative sums (`SUM() OVER`).
- Comparing current row with previous/next row (`LEAD` / `LAG`).
- Analyzing trends over time without losing row-level detail.

In short:

Window functions allow **row-by-row calculations with context**, unlike regular aggregates which collapse data into a single value. They are powerful for analytics, reporting, and advanced SQL queries.

COLLATION

Collation is the set of rules MySQL uses to **compare and sort text**. It defines how characters are ordered (like A-Z, a-z, special characters) and whether comparisons are **case-sensitive** or **accent-sensitive**. Collation is always tied to a **character set** (for example, `utf8mb4`).

Think of collation like **language rules for your database**:

- In English, `a < b < c`.
- In German, `ä` might come after `a`.
- Collation tells MySQL how to handle these comparisons.

Examples of Collation in MySQL

1. **Case-Insensitive (CI)** – `utf8mb4_general_ci`
   - `ci` = case-insensitive
   - `'A' = 'a'` → TRUE
2. **Case-Sensitive (CS)** – `utf8mb4_bin`
   - `'A' = 'a'` → FALSE
   - Sorted strictly by **binary value**
3. **Accent-Insensitive (AI)** – `utf8mb4_unicode_ci_ai`
   - `'é' = 'e'` → TRUE
   - Ignores accents while comparing

In short: Collation controls **how text is sorted and compared**, including case sensitivity and accent sensitivity, according to the rules of the chosen language or character set.