RKIT Training Notes
Week - 2

Subqueries:

A **subquery**, also known as a nested or inner query, is simply a SELECT statement that is placed inside another SQL query (the outer query).

The easiest way to think about it is as a **two-step question**. The result of the inner question (the subquery) is used to help answer the outer question (the main query).

Ex: SELECT col FROM table

WHERE col > (SELECT AVG(col) FROM table);

Correlated Queries:

A correlated subquery is a subquery that depends on values from the outer query. Unlike a regular (non-correlated) subquery, it is evaluated once for every row in the outer query. This makes correlated subqueries dynamic and highly useful for solving complex database problems like row-by-row comparisons, filtering, and conditional updates.

- Executes once for each row of the outer query.
- Subquery uses values from the outer query.
- Ideal for ranking, row-specific calculations, and conditional logic

Ex:
```
select student_id, name, course_id
from Students s
where s.course_id = (
        select s2.course_id
   from Students s2
   where s2.course_id = s.course_id
);
```

UNION:

The UNION operator is used to combine the result-set of two or more SELECT statements. The UNION operator automatically removes duplicate rows from the result set.

Requirements for UNION:

- Every SELECT statement within UNION must have the same number of columns.
- The columns must also have similar data types
- The columns in every SELECT statement must also be in the same order.

```
Ex: select column_name(s) from table1
    UNION
    select column_name(s) from table2;
```

UNION ALL:

The UNION ALL operator is used to combine the result-set of two or more SELECT statements.

The UNION ALL operator includes all rows from each statement, including any duplicates.

Requirements for UNION ALL:

- Every SELECT statement within UNION ALL must have the same number of columns
- The columns must also have similar data types
- The columns in every SELECT statement must also be in the same order

```
Ex: select column_name(s) from table1
    UNION ALL
    select column_name(s) from table2;
```

Indexes:

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

An index is a separate data structure, usually a **B-tree**, that is created on one or more columns of a table. It stores two key pieces of information:

1. **The indexed column's value:** The value from the column you created the index on (e.g., a specific `LastName`).
2. **A pointer:** A reference to the physical location of the full row of data in the table.

Because the index is kept sorted, searching for a value within it is extremely fast.

Advantages

- **Faster Data Retrieval:** Drastically speeds up `SELECT` queries, especially those with `WHERE`, `JOIN`, and `ORDER BY` clauses.

- **Enforce Uniqueness:** A `UNIQUE` index ensures that no two rows have the same value in the indexed column(s).

Disadvantages

- **Slower Data Modification:** `INSERT`, `UPDATE`, and `DELETE` operations become slower because the database must update not only the table but also the index.
- **Requires Storage Space:** Indexes are physical objects that take up disk space. The more indexes you have, the more space they consume.

Ex: CREATE INDEX index_mail ON Students(email);

EXPLAIN SELECT * FROM Students WHERE email='rohan.mehta@example.com';

EXPLAIN: `EXPLAIN` is a command in SQL that asks the database to show you its execution plan for a query without actually running the query.

Stored Procedures:

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

You define a procedure with a name, optional input parameters (to make it dynamic), and the SQL code you want to run. The database compiles and stores this procedure. When you call it, the database engine simply executes the pre-compiled code.

```
Ex: DELIMITER //
CREATE PROCEDURE GetStudentsByCourse(IN cname VARCHAR(100))
BEGIN
   SELECT s.student_id, s.name, s.age, s.gender, s.email
   FROM Students s
   JOIN Courses c ON s.course_id = c.course_id
   WHERE c.course_name = cname;
END //
```

DELIMITER ;
CALL GetStudentsByCourse('Computer Science');

Delimiter: Special character that tells the client program where one SQL statement ends. Default (;)

`IN` parameter: Passes a value *into* the procedure. (This is the default).

`OUT` parameter: A variable you provide that the procedure can modify to pass a value *back out*.

Multi-valued Parameter

SQL databases like MySQL do **not** have a native array or list type for parameters. You cannot directly pass a list of values as a single argument.

The common workaround is to pass the values as a single **comma-separated string**. The procedure then uses string functions to work with this list.

Ex: FIND_IN_SET   //FIND_IN_SET is a MySQL function that finds a value within a comma-separated string

Functions:

SQL functions are built-in or user-defined operations that perform a calculation on data and return a result. They are used to manipulate data, perform calculations, and format output directly within SQL queries, making your code more powerful and concise.

Functions can be broadly divided into three main categories: **Scalar**, **Aggregate**, and **Window** functions.

Scalar Functions

A **scalar function** operates on a single input value and returns a single output value for each row. Think of it as a simple transformation applied to every row independently.

Common Types of Scalar Functions

String Functions
Numeric Functions
Data Functions

Aggregate Functions

An **aggregate function** takes a set of values (from multiple rows) as input and returns a single summary value. These functions are almost always used with the `GROUP BY` clause to calculate metrics for different categories.

Window Functions

A **window function** is like a hybrid. It operates on a set of rows (a "window frame") related to the current row, but it **returns a value for each individual row**, unlike an aggregate function which collapses rows into one. They are powerful for creating rankings, running totals, and comparing rows within the same result set.

Window functions are identified by the `OVER()` clause.

```
Ex: DELIMITER //
CREATE FUNCTION GetGrade(score INT)
RETURNS CHAR(1)
DETERMINISTIC
BEGIN
   RETURN CASE
      WHEN score >= 90 THEN 'A'
      WHEN score >= 80 THEN 'B'
      WHEN score >= 70 THEN 'C'
      WHEN score >= 60 THEN 'D'
      ELSE 'F'
   END;
END //
DELIMITER ;
```

Triggers

A trigger is a special stored procedure in a database that automatically executes when specific events (like INSERT, UPDATE, or DELETE) occur on a table. Triggers help automate tasks, maintain data consistency, and record database activities. Each trigger is tied to a particular table and runs without manual execution.

Uses of SQL Triggers

- Automation: Handles repetitive tasks.
- Data Integrity: Ensures clean and accurate data.
- Business Rules: Enforces database logic.
- Audit Trails: Tracks and records changes.

`BEFORE`: The trigger runs **before** the event's action is actually performed on the table. This is useful for validating or changing data *before* it's saved.

`AFTER`: The trigger runs **after** the event's action has been completed. This is ideal for logging changes or updating other tables based on the result.

`NEW`: Represents the row as it will look *after* the `INSERT` or `UPDATE`. It contains the new data. (Not available in `DELETE` triggers).

`OLD`: Represents the row as it looked *before* the `UPDATE` or `DELETE`. It contains the original data. (Not available in `INSERT` triggers).

```
Ex: DELIMITER %%
CREATE TRIGGER LogDeletedStudent
BEFORE DELETE ON Students
FOR EACH ROW
BEGIN
   INSERT INTO DeletedStudents
   VALUES (OLD.student_id, OLD.name, OLD.age, OLD.gender, NOW());
END %%
DELIMITER ;
```

Views

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A standard view doesn't store any data itself. When you query a view, the database engine runs the underlying `SELECT` statement in the background and presents the results to you.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

```
Ex: CREATE VIEW StudentCourseView AS
SELECT s.name AS student_name, c.course_name
FROM Students s
JOIN Courses c ON s.course_id = c.course_id;
```

Cursors

A SQL **cursor** is a database object that allows you to process a result set one row at a time, like a loop in a programming language.

Standard SQL operations are **set-based**, meaning they operate on all the rows in a result set at once. A cursor breaks this model by providing a pointer that you can move through the result set row-by-row to perform complex, stateful operations on each individual record.

While powerful, cursors are notoriously slow and resource-intensive. They should be considered a **last resort** when a set-based solution (like a `JOIN` or window function) is not possible.

The Cursor Lifecycle

Using a cursor involves a specific sequence of five steps.

1. `DECLARE`

First, you declare the cursor and associate it with a `SELECT` statement. This defines the result set that the cursor will iterate over but does not execute the query yet.

2. `OPEN`

The `OPEN` command executes the `SELECT` statement defined in the `DECLARE` step. The results are fetched from the database tables and stored in a temporary, private work area for the cursor. The pointer is positioned *before* the first row.

3. `FETCH`

This is the core of the loop. The `FETCH` command retrieves a single row that the pointer is currently on, copies its data into local variables, and then advances the pointer to the next row in the result set. You typically run this command inside a loop until there are no more rows left to fetch.

4. `CLOSE`

After you're done looping through the rows, you must `CLOSE` the cursor. This releases the current result set and removes any database locks associated with the cursor, but the cursor's definition still exists and can be reopened.

5. `DEALLOCATE`

Finally, `DEALLOCATE` completely removes the cursor definition and frees all system resources associated with it. After this, the cursor can no longer be used unless it is declared again.

Ex:

```
DELIMITER //

CREATE PROCEDURE GetMaleStudentsCursor()
BEGIN
   DECLARE done INT DEFAULT FALSE;
   DECLARE sid INT;
   DECLARE sname VARCHAR(100);

-- cursor to fetch male students
   DECLARE male_cursor CURSOR FOR
      SELECT student_id, name FROM Students WHERE gender = 'Male';

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

   OPEN male_cursor;

   read_loop: LOOP
      FETCH male_cursor INTO sid, sname;
      IF done THEN
         LEAVE read_loop;
      END IF;
      -- show each male student (can also insert into a temp table)
      SELECT sid AS student_id, sname AS student_name;
   END LOOP;

   CLOSE male_cursor;
END //

DELIMITER ;
```