Mass Coders - SQL Notes

Index

    - SELECT
    - UPDATE
    - DELETE
3. **SQL Functions and Operators**
    - String Functions
        - CONCAT
        - SUBSTRING
        - LENGTH
    - Numeric Functions
        - ABS
        - ROUND
        - FLOOR
    - Date Functions
        - CURRENT_DATE
        - DATEADD
        - DATEDIFF
    - Operators
        - Arithmetic Operators
        - Comparison Operators
        - Logical Operators
4. **SQL Clauses**
    - WHERE Clause
        - Filtering Data
    - ORDER BY Clause
        - Sorting Data
    - GROUP BY Clause
        - Aggregating Data
    - HAVING Clause
        - Filtering Aggregated Data

5. **Joins and Subqueries**
    - Types of Joins
        - INNER JOIN
        - LEFT JOIN
        - RIGHT JOIN
        - FULL JOIN
        - CROSS JOIN
    - Subqueries
        - Single-Row Subqueries
        - Multi-Row Subqueries
        - Correlated Subqueries
6. **Advanced SQL Concepts**
    - Views
        - Creating and Managing Views
    - Indexes
        - Types and Uses of Indexes
        - Creating and Dropping Indexes
    - Stored Procedures
        - Creating and Executing Stored Procedures
    - Triggers
        - Creating and Managing Triggers
    - Transactions
        - ACID Properties
        - COMMIT, ROLLBACK, and SAVEPOINT
7. **Performance Tuning and Optimization**
    - Query Optimization
        - Understanding Execution Plans
        - Index Optimization
    - Database Normalization

    - Normal Forms (1NF, 2NF, 3NF, BCNF)
- Common Performance Issues
    - Identifying and Resolving Bottlenecks

8. **SQL Security**
    - User Management
        - Creating and Managing Users
    - Roles and Permissions
        - Granting and Revoking Privileges

Introduction to SQL

What is SQL?

SQL, or Structured Query Language, is the standard language used to interact with relational databases. It's the tool we use to communicate with a database, to perform tasks such as retrieving data, updating records, and creating tables. Think of SQL as a way to ask your database a question or give it an instruction. Whether you want to find a list of customers, add a new order, or update a product's price, SQL is the language that makes it happen.

History and Evolution

SQL was born out of the need for a standard method of managing and manipulating relational databases. It all started in the early 1970s at IBM, where Donald D. Chamberlin and Raymond F. Boyce developed SEQUEL (Structured English Query Language) to interact with the company's System R, one of the first relational database systems. SEQUEL was later renamed SQL due to trademark issues.

By the late 1970s, SQL had gained traction and was adopted by several database vendors. In 1986, the American National Standards Institute (ANSI) standardized SQL, solidifying its role in the world of databases. Over the years, SQL has evolved, adding new features and capabilities with each update. Major databases like Oracle, MySQL, SQL Server, and PostgreSQL have their own versions and extensions of SQL, but the core language remains the same.

Importance of SQL in Databases

SQL is vital because it provides a unified way to interact with and manage databases. Here are some reasons why SQL is so important:

1. **Efficiency**: SQL allows users to perform complex queries with simple statements. For example, fetching data from multiple tables with just one query.
2. **Standardization**: Being a standardized language, SQL ensures consistency across different database systems. If you learn SQL, you can work with any relational database.
3. **Data Manipulation**: SQL lets you insert, update, delete, and retrieve data efficiently. It also supports transaction processing, ensuring data integrity.
4. **Security**: SQL includes robust security features, such as user permissions and roles, to protect sensitive data.

Real-life Example: Consider a large e-commerce platform like Amazon. SQL is used to manage their massive database, which includes products, customers, orders, and inventory. With SQL, they can quickly retrieve information about product availability, update stock levels, and process millions of transactions daily.

Types of SQL Commands

SQL commands are divided into different categories based on their functionality. Here's a breakdown of the main types of SQL commands:

1. **DDL (Data Definition Language)**

   DDL commands are used to define and modify the structure of database objects like tables, indexes, and schemas. Key DDL commands include:

   - **CREATE**: Used to create a new table, index, or database.
   - **ALTER**: Modifies the structure of an existing table or database.
   - **DROP**: Deletes tables, indexes, or databases.
   - **TRUNCATE**: Removes all records from a table but retains the structure for future use.

   Real-life Example: When setting up a new customer management system, a company would use DDL commands to create tables for storing customer information, orders, and product details.

2. **DML (Data Manipulation Language)**

   DML commands are used to manipulate the data within database objects. These include:

   - **SELECT**: Retrieves data from one or more tables.

- **INSERT**: Adds new records to a table.
- **UPDATE**: Modifies existing records in a table.
- **DELETE**: Removes records from a table.

Real-life Example: A retail store using an inventory system would use DML commands to update stock levels as products are sold and restocked.

3. **DCL (Data Control Language)**

   DCL commands manage access permissions and control the security of the database. These commands include:

   - **GRANT**: Gives a user permission to perform specific tasks.
   - **REVOKE**: Removes previously granted permissions from a user.

   Real-life Example: In a banking system, administrators use DCL commands to grant access to different levels of data to employees based on their roles and responsibilities.

4. **TCL (Transaction Control Language)**

   TCL commands are used to manage transactions in a database, ensuring that operations are completed successfully before making changes permanent. Key TCL commands include:

   - **COMMIT**: Saves all changes made during the current transaction.
   - **ROLLBACK**: Reverts the database to its previous state before the transaction began.
   - **SAVEPOINT**: Sets a savepoint within a transaction to which you can later roll back.

   Real-life Example: During an online purchase, TCL commands ensure that payment processing, order placement, and inventory updates are all completed successfully. If any part of the process fails, a rollback ensures the database remains consistent and accurate.

Basic SQL Commands

Creating a Database and Tables

Creating databases and tables is the first step in organizing your data. Let's break down the commands and concepts involved.

CREATE DATABASE

The `CREATE DATABASE` command is used to create a new database. A database is a collection of tables and other objects that store and organize data.

Syntax:

```
CREATE DATABASE database_name;
```

Example:
Imagine you're setting up a database for a Telugu movie rental store. You would start by creating a database named `TeluguMoviesDB`:

```
CREATE DATABASE TeluguMoviesDB;
```

Real-life Example:
Think of `CREATE DATABASE` as setting up a new file cabinet for organizing all the documents related to your project. Just like you would name a file cabinet, you give your database a name that reflects its purpose.

CREATE TABLE

The `CREATE TABLE` command is used to create a new table within a database. A table is a collection of related data held in a structured format within a database. It consists of rows and columns.

Syntax:

```
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...
);
```

Example:
Continuing with the Telugu movie rental store example, you might create a table for movies:

```
CREATE TABLE Movies (
    MovieID INT PRIMARY KEY,
    Title VARCHAR(100),
```

```
    Director VARCHAR(100),
    ReleaseYear INT,
    Genre VARCHAR(50)
);
```

Real-life Example:
Creating a table is like setting up a new drawer in your file cabinet with labeled folders (columns) where each folder holds documents (rows) of a specific type.

Data Types

Data types define the kind of data that can be stored in a column. Common data types include:

- **INT**: Integer numbers.
- **VARCHAR(size)**: Variable-length strings.
- **CHAR(size)**: Fixed-length strings.
- **DATE**: Date values.
- **FLOAT**: Floating-point numbers.
- **BOOLEAN**: True/false values.

Example:
In the `Movies` table example above:

- `MovieID` is an integer.
- `Title` and `Director` are variable-length strings.
- `ReleaseYear` is an integer.
- `Genre` is a variable-length string.

Real-life Example:
Choosing data types is like deciding what kind of documents go into each folder in your drawer. Some folders might hold numbers, others text, and some might have dates.

Basic Data Manipulation

Once your database and tables are set up, you need to manipulate the data. This includes inserting new records, selecting data to view, updating existing records, and deleting records.

INSERT INTO

The `INSERT INTO` command is used to add new records to a table.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:
Adding a new Telugu movie to the
`Movies` table:

```
INSERT INTO Movies (MovieID, Title, Director, ReleaseYear, Gen
re)
VALUES (1, 'Baahubali: The Beginning', 'S. S. Rajamouli', 201
5, 'Action/Drama');
```

Real-life Example:
Inserting a record is like adding a new document to one of the folders in your drawer.

SELECT

The `SELECT` command is used to retrieve data from one or more tables. It's the most commonly used SQL command.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:
Retrieving all movies directed by 'S. S. Rajamouli':

```
SELECT * FROM Movies
WHERE Director = 'S. S. Rajamouli';
```

Real-life Example:
Using
`SELECT` is like pulling out documents from your drawer that match a certain criteria.

UPDATE

The `UPDATE` command is used to modify existing records in a table.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:
Updating the genre of a movie:

```
UPDATE Movies
SET Genre = 'Epic Action/Drama'
WHERE MovieID = 1;
```

Real-life Example:
Updating a record is like editing a document in one of your folders.

DELETE

The `DELETE` command is used to remove records from a table.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example:
Deleting a movie from the
`Movies` table:

```
DELETE FROM Movies
WHERE MovieID = 1;
```

Real-life Example:
Deleting a record is like removing a document from a folder and discarding it.

SQL Functions and Operators

String Functions

String functions allow you to manipulate text data in various ways. Here are some of the most commonly used string functions in SQL:

CONCAT

The `CONCAT` function is used to combine two or more strings into one.

Syntax:

```
CONCAT(string1, string2, ...);
```

Example:
Imagine you have a table
`Actors` with columns `FirstName` and `LastName`. You want to create a full name for each actor.

```
SELECT CONCAT(FirstName, ' ', LastName) AS FullName
FROM Actors;
```

Real-life Example:
Think of
`CONCAT` as pasting together different pieces of text to form a complete message. For example, combining a first name and last name to form a full name.

SUBSTRING

The `SUBSTRING` function extracts a portion of a string.

Syntax:

```
SUBSTRING(string, start, length);
```

Example:
You have a table
`Movies` with a column `Title`. You want to extract the first three characters of each title.

```
SELECT SUBSTRING(Title, 1, 3) AS ShortTitle
FROM Movies;
```

Real-life Example:
Using

`SUBSTRING` is like taking a specific segment of text from a paragraph, such as extracting the first word from a sentence.

LENGTH

The `LENGTH` function returns the number of characters in a string.

Syntax:

```
LENGTH(string);
```

Example:
To find out the length of movie titles in the `Movies` table:

```
SELECT Title, LENGTH(Title) AS TitleLength
FROM Movies;
```

Real-life Example:
Using `LENGTH` is similar to counting the number of characters in a sentence, including spaces and punctuation.

Numeric Functions

Numeric functions perform operations on numerical data. Here are some key numeric functions:

ABS

The `ABS` function returns the absolute value of a number, removing any negative sign.

Syntax:

```
ABS(number);
```

Example:
If you have a column `RevenueChange` in the `Movies` table that stores the change in revenue, and you want to get the absolute values:

```
SELECT Title, ABS(RevenueChange) AS AbsoluteChange
FROM Movies;
```

Real-life Example:
Using
`ABS` is like considering only the magnitude of a number without its sign, similar to how we only care about distance in terms of positive values.

ROUND

The `ROUND` function rounds a number to a specified number of decimal places.

Syntax:

```
ROUND(number, decimal_places);
```

Example:
Rounding the revenue of movies to the nearest thousand in the
`Movies` table:

```
SELECT Title, ROUND(Revenue, -3) AS RoundedRevenue
FROM Movies;
```

Real-life Example:
Using
`ROUND` is like rounding prices to the nearest dollar when shopping to simplify calculations.

FLOOR

The `FLOOR` function returns the largest integer less than or equal to a number.

Syntax:

```
FLOOR(number);
```

Example:
Finding the floor value of ratings in a
`Movies` table:

```
SELECT Title, FLOOR(Rating) AS FloorRating
```

```
FROM Movies;
```

Real-life Example:
Using
`FLOOR` is like rounding down a number to the nearest whole number, similar to dropping any fractional part when dealing with counts.

Date Functions

Date functions are used to manipulate date and time values. Here are some essential date functions:

CURRENT_DATE

The `CURRENT_DATE` function returns the current date.

Syntax:

```
SELECT CURRENT_DATE;
```

Example:
To get the current date in a report:

```
SELECT CURRENT_DATE AS TodayDate;
```

Real-life Example:
Using
`CURRENT_DATE` is like checking today's date on your calendar.

DATEADD

The `DATEADD` function adds a specified number of intervals to a date.

Syntax:

```
DATEADD(interval, number, date);
```

Example:
Adding 7 days to the current date:

```
SELECT DATEADD(day, 7, CURRENT_DATE) AS NextWeek;
```

Real-life Example:
Using
`DATEADD` is like scheduling a reminder for a week from today.

DATEDIFF

The `DATEDIFF` function returns the difference between two dates.

Syntax:

```
DATEDIFF(interval, start_date, end_date);
```

Example:
Calculating the number of days between the release date and today's date for movies in the
`Movies` table:

```
SELECT Title, DATEDIFF(day, ReleaseDate, CURRENT_DATE) AS Days
SinceRelease
FROM Movies;
```

Real-life Example:
Using
`DATEDIFF` is like calculating the number of days between two events, such as the number of days left until a movie release.

Operators

Operators are used to perform operations on data. Here are some commonly used SQL operators:

Arithmetic Operators

Arithmetic operators perform mathematical operations.

Operators:

- `+` (Addition)
- ` ` (Subtraction)
- ` ` (Multiplication)
- `/` (Division)
- `%` (Modulus)

Example:
Calculating the total revenue after applying a discount:

```
SELECT Title, Revenue, Revenue - (Revenue * 0.1) AS Discounted
Revenue
FROM Movies;
```

Real-life Example:
Using arithmetic operators is like calculating the total cost of items in your shopping cart after applying discounts.

Comparison Operators

Comparison operators compare two values.

Operators:

- `=` (Equal)
- `!=` or `<>` (Not Equal)
- `>` (Greater Than)
- `<` (Less Than)
- `>=` (Greater Than or Equal To)
- `<=` (Less Than or Equal To)

Example:
Finding movies released after 2015:

```
SELECT Title
FROM Movies
WHERE ReleaseYear > 2015;
```

Real-life Example:
Using comparison operators is like filtering search results based on criteria, such as finding all movies released after a certain year.

Logical Operators

Logical operators are used to combine multiple conditions.

Operators:

- `AND`

- `OR`
- `NOT`

Example:
Finding movies that are either action or drama and have a rating greater than 8:

```
SELECT Title
FROM Movies
WHERE (Genre = 'Action' OR Genre = 'Drama') AND Rating > 8;
```

Real-life Example:
Using logical operators is like refining your search criteria on a movie streaming service to find movies that meet multiple conditions.

SQL Clauses

WHERE Clause

The `WHERE` clause is used to filter records that meet a certain condition. It is essential for retrieving specific data from a database.

Filtering Data

The `WHERE` clause allows you to specify conditions that the data must meet to be included in the result set.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:
Imagine you have a table `TeluguMovies` with columns `MovieID`, `Title`, `Director`, and `ReleaseYear`. To find all movies directed by S. S. Rajamouli:

```
SELECT Title, ReleaseYear
FROM TeluguMovies
WHERE Director = 'S. S. Rajamouli';
```

Real-life Example:
Using the
`WHERE` clause is like applying a filter to your search results on a movie database to only show movies by a specific director.

ORDER BY Clause

The `ORDER BY` clause is used to sort the result set by one or more columns, either in ascending (default) or descending order.

Sorting Data

The `ORDER BY` clause arranges the records in a specified order.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

Example:
To list all movies sorted by their release year in descending order:

```
SELECT Title, ReleaseYear
FROM TeluguMovies
ORDER BY ReleaseYear DESC;
```

Real-life Example:
Using the
`ORDER BY` clause is like sorting your movie collection by release date to find the most recent ones easily.

GROUP BY Clause

The `GROUP BY` clause groups rows that have the same values in specified columns into summary rows, like "total", "average", "count", etc.

Aggregating Data

The `GROUP BY` clause is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result set by one or more columns.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

Example:
To find the number of movies directed by each director:

```
SELECT Director, COUNT(*) AS NumberOfMovies
FROM TeluguMovies
GROUP BY Director;
```

Real-life Example:
Using
`GROUP BY` is like organizing your movie list by director and then counting how many movies each director has made.

HAVING Clause

The `HAVING` clause is used to filter groups based on a specified condition. It is similar to the `WHERE` clause but is used for groups rather than individual rows.

Filtering Aggregated Data

The `HAVING` clause filters data after it has been grouped.

Syntax:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Example:
To find directors who have directed more than 3 movies:

```
SELECT Director, COUNT(*) AS NumberOfMovies
FROM TeluguMovies
GROUP BY Director
HAVING COUNT(*) > 3;
```

Real-life Example:
Using the
`HAVING` clause is like filtering a summary report to only show entries that meet certain criteria, such as showing only those directors who have made a significant number of films.

Detailed Breakdown with Telugu Movie Examples

WHERE Clause

Let's explore more about the `WHERE` clause with Telugu movie examples.

Example 1:
Finding movies released after the year 2010:

```
SELECT Title, ReleaseYear
FROM TeluguMovies
WHERE ReleaseYear > 2010;
```

Example 2:
Finding movies with the genre 'Action':

```
SELECT Title, Director
FROM TeluguMovies
WHERE Genre = 'Action';
```

Real-life Example:
Using
`WHERE` is like searching for movies in your personal collection that are only from the action genre and released after 2010.

ORDER BY Clause

The `ORDER BY` clause helps organize your result set.

Example 1:
Sorting movies by title in ascending order:

```
SELECT Title, ReleaseYear
FROM TeluguMovies
ORDER BY Title ASC;
```

Example 2:
Sorting movies by release year and then by title:

```
SELECT Title, ReleaseYear
FROM TeluguMovies
ORDER BY ReleaseYear ASC, Title ASC;
```

Real-life Example:
Using
`ORDER BY` is like sorting your movie shelf first by release year and then alphabetically by title.

GROUP BY Clause

The `GROUP BY` clause helps summarize data.

Example 1:
Finding the average rating of movies by each director:

```
SELECT Director, AVG(Rating) AS AverageRating
FROM TeluguMovies
GROUP BY Director;
```

Example 2:
Counting the number of movies in each genre:

```
SELECT Genre, COUNT(*) AS NumberOfMovies
FROM TeluguMovies
GROUP BY Genre;
```

Real-life Example:
Using
`GROUP BY` is like categorizing your movies by genre and counting how many you have in each category.

HAVING Clause

The `HAVING` clause refines your grouped data.

Example 1:
Finding genres with more than 5 movies:

```
SELECT Genre, COUNT(*) AS NumberOfMovies
FROM TeluguMovies
GROUP BY Genre
HAVING COUNT(*) > 5;
```

Example 2:
Finding directors with an average movie rating above 7:

```
SELECT Director, AVG(Rating) AS AverageRating
FROM TeluguMovies
GROUP BY Director
HAVING AVG(Rating) > 7;
```

Real-life Example:
Using
`HAVING` is like filtering your movie summary report to only show genres that have a significant number of movies or directors whose movies have high ratings.

Joins and Subqueries

Types of Joins

Joins are used to combine rows from two or more tables based on a related column. Let's explore the different types of joins with examples.

INNER JOIN

An `INNER JOIN` returns only the rows that have matching values in both tables.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Example:
Suppose we have two tables,
`Movies` and `Directors`:

```
CREATE TABLE Movies (
    MovieID INT,
    Title VARCHAR(100),
    DirectorID INT,
    ReleaseYear INT
);

CREATE TABLE Directors (
    DirectorID INT,
    DirectorName VARCHAR(100)
);

INSERT INTO Movies (MovieID, Title, DirectorID, ReleaseYear) V
ALUES
(1, 'Baahubali: The Beginning', 101, 2015),
(2, 'Eega', 101, 2012),
(3, 'Ala Vaikunthapurramuloo', 102, 2020);

INSERT INTO Directors (DirectorID, DirectorName) VALUES
(101, 'S. S. Rajamouli'),
(102, 'Trivikram Srinivas');
```

To get a list of movies along with their directors:

```
SELECT Movies.Title, Directors.DirectorName
FROM Movies
INNER JOIN Directors
ON Movies.DirectorID = Directors.DirectorID;
```

Result (Varies on the input data you entered):

Title	DirectorName
Baahubali: The Beginning	S. S. Rajamouli
Eega	S. S. Rajamouli
Ala Vaikunthapurramuloo	Trivikram Srinivas

Real-life Example:
An
`INNER JOIN` is like finding books in a library that are written by authors present in the authors' list.

LEFT JOIN

A `LEFT JOIN` returns all rows from the left table and the matched rows from the right table. If there is no match, the result is NULL on the right side.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Example:
To get a list of all movies along with their directors, including movies without a director in the
`Directors` table:

```
SELECT Movies.Title, Directors.DirectorName
FROM Movies
LEFT JOIN Directors
ON Movies.DirectorID = Directors.DirectorID;
```

Result (Varies on the input data you entered):

Title	DirectorName
Baahubali: The Beginning	S. S. Rajamouli
Eega	S. S. Rajamouli
Ala Vaikunthapurramuloo	Trivikram Srinivas
Vinaya Vidheya Rama	NULL

Real-life Example:
A
`LEFT JOIN` is like getting a list of students and their grades, including those who haven't taken any exams yet.

RIGHT JOIN

A `RIGHT JOIN` returns all rows from the right table and the matched rows from the left table. If there is no match, the result is NULL on the left side.

Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Example:
To get a list of all directors and the movies they have directed, including directors with no movies:

```
SELECT Movies.Title, Directors.DirectorName
FROM Movies
RIGHT JOIN Directors
ON Movies.DirectorID = Directors.DirectorID;
```

Result (Varies on the input data you entered):

Title	DirectorName
Baahubali: The Beginning	S. S. Rajamouli
Eega	S. S. Rajamouli
Ala Vaikunthapurramuloo	Trivikram Srinivas
NULL	Sukumar

Real-life Example:
A
`RIGHT JOIN` is like getting a list of all courses and the students enrolled in them, including courses with no students enrolled.

FULL JOIN

A `FULL JOIN` returns all rows when there is a match in either the left or right table. If there is no match, the result is NULL from the side where there is no match.

Syntax:

```
SELECT columns
FROM table1
FULL JOIN table2
ON table1.column = table2.column;
```

Example:
To get a complete list of all movies and directors, including those with no matching

records:

```
SELECT Movies.Title, Directors.DirectorName
FROM Movies
FULL JOIN Directors
ON Movies.DirectorID = Directors.DirectorID;
```

Result (Varies on the input data you entered):

Title	DirectorName
Baahubali: The Beginning	S. S. Rajamouli
Eega	S. S. Rajamouli
Ala Vaikunthapurramuloo	Trivikram Srinivas
NULL	Sukumar
Vinaya Vidheya Rama	NULL

Real-life Example:
A
`FULL JOIN` is like merging two lists of employees and departments, showing all employees and all departments, including those without a corresponding match.

CROSS JOIN

A `CROSS JOIN` returns the Cartesian product of the two tables, combining all rows from the left table with all rows from the right table.

Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

Example:
To get all possible combinations of movies and directors:

```
SELECT Movies.Title, Directors.DirectorName
FROM Movies
CROSS JOIN Directors;
```

Result (Varies on the input data you entered):

Title	DirectorName
Baahubali: The Beginning	S. S. Rajamouli
Baahubali: The Beginning	Trivikram Srinivas
Eega	S. S. Rajamouli
Eega	Trivikram Srinivas
Ala Vaikunthapurramuloo	S. S. Rajamouli
Ala Vaikunthapurramuloo	Trivikram Srinivas

Real-life Example:
A
`CROSS JOIN` is like creating all possible pairs of shirts and pants to see every possible outfit combination.

Subqueries

Subqueries are queries nested inside another query. They can be used to perform operations that require multiple steps or to simplify complex queries.

Single-Row Subqueries

A single-row subquery returns one row. It is typically used with comparison operators like `=`, `<`, `>`, etc.

Example:
Finding the director of the most recent movie:

```
SELECT DirectorName
FROM Directors
WHERE DirectorID = (SELECT DirectorID
                    FROM Movies
                    ORDER BY ReleaseYear DESC
                    LIMIT 1);
```

Result (Varies on the input data you entered):

DirectorName

Trivikram Srinivas

Real-life Example:
A single-row subquery is like finding the best-performing student in a class and then getting their details.

Multi-Row Subqueries

A multi-row subquery returns multiple rows. It is typically used with operators like `IN`, `ANY`, `ALL`.

Example:
Finding all movies directed by directors who have directed more than one movie:

```
SELECT Title
FROM Movies
WHERE DirectorID IN (SELECT DirectorID
                     FROM Movies
                     GROUP BY DirectorID
                     HAVING COUNT(*) > 1);
```

Result (Varies on the input data you entered):

Title
Baahubali: The Beginning
Eega

Real-life Example:
A multi-row subquery is like finding all courses taught by professors who teach multiple courses.

Correlated Subqueries

A correlated subquery is a subquery that uses values from the outer query. It is executed once for each row processed by the outer query.

Example:
Finding movies where the director has directed another movie in the same year:

```
SELECT Title
FROM Movies m1
WHERE EXISTS (SELECT 1
              FROM Movies m2
              WHERE m1.DirectorID = m2.DirectorID
              AND m1.ReleaseYear = m2.ReleaseYear
              AND m1.MovieID <> m2.MovieID);
```

Result (Varies on the input data you entered):

Title
Eega

Real-life Example:
A correlated subquery is like checking each student's score and comparing it with the class average to identify above-average students.

Advanced SQL Concepts

Views

A view is a virtual table based on the result set of an SQL query. Views simplify complex queries, enhance security, and make data more accessible.

Creating and Managing Views

Creating a View

A view is created using the `CREATE VIEW` statement.

Syntax:

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

Example:
Imagine we have a table `TeluguMovies` and we want to create a view that shows only movies directed by 'S. S. Rajamouli'.

```
CREATE VIEW RajamouliMovies AS
SELECT Title, ReleaseYear
FROM TeluguMovies
WHERE Director = 'S. S. Rajamouli';
```

To query the view:

```
SELECT * FROM RajamouliMovies;
```

Managing Views

Views can be updated, dropped, or replaced as needed.

Updating a View

```
CREATE OR REPLACE VIEW RajamouliMovies AS
SELECT Title, ReleaseYear, Genre
FROM TeluguMovies
WHERE Director = 'S. S. Rajamouli';
```

Dropping a View

```
DROP VIEW RajamouliMovies;
```

Real-life Example:
Creating a view is like setting up a specific filter or report in your software application that shows only relevant data to the user without them needing to write complex queries.

Indexes

Indexes improve the speed of data retrieval operations on a database table at the cost of additional storage space and slower write operations.

Types and Uses of Indexes

Types of Indexes

1. **Clustered Index**: Determines the physical order of data in a table. Each table can have only one clustered index.
2. **Non-Clustered Index**: Does not alter the physical order of the data. Each table can have multiple non-clustered indexes.
3. **Unique Index**: Ensures all values in the index are unique.
4. **Full-Text Index**: Used for performing full-text searches.

Uses of Indexes

Indexes are used to:

- Speed up the retrieval of rows.
- Enforce uniqueness with unique indexes.
- Improve performance of search queries with full-text indexes.

Example:
Creating an index on the `Title` column of the `TeluguMovies` table.

```
CREATE INDEX idx_title ON TeluguMovies (Title);
```

Creating and Dropping Indexes

Creating an Index

Syntax:

```
CREATE [UNIQUE] INDEX index_name ON table_name (column1, colum
n2, ...);
```

Example:
Creating a unique index on the `MovieID` column.

```
CREATE UNIQUE INDEX idx_movie_id ON TeluguMovies (MovieID);
```

Dropping an Index

Syntax:

```
DROP INDEX index_name;
```

Example:
Dropping the index on the `Title` column.

```
DROP INDEX idx_title;
```

Real-life Example:
Using an index is like having a detailed table of contents in a book that allows you to quickly find the information you need without flipping through every page.

Stored Procedures

Stored procedures are a collection of SQL statements that can be executed as a single unit. They help in reusing code and improving performance.

Creating and Executing Stored Procedures

Creating a Stored Procedure

Syntax:

```
CREATE PROCEDURE procedure_name
AS
BEGIN
    SQL statements;
END;
```

Example:
Creating a stored procedure to add a new movie.

```
CREATE PROCEDURE AddMovie
    @Title VARCHAR(100),
    @DirectorID INT,
    @ReleaseYear INT,
    @Genre VARCHAR(50)
AS
BEGIN
    INSERT INTO TeluguMovies (Title, DirectorID, ReleaseYear,
Genre)
    VALUES (@Title, @DirectorID, @ReleaseYear, @Genre);
END;
```

Executing a Stored Procedure

```
EXEC AddMovie 'RRR', 101, 2021, 'Action/Drama';
```

Real-life Example:
A stored procedure is like a macro in Excel that performs a series of actions automatically when you run it.

Triggers

Triggers are special types of stored procedures that automatically execute in response to certain events on a table or view.

Creating and Managing Triggers

Creating a Trigger

Syntax:

```
CREATE TRIGGER trigger_name
ON table_name
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    SQL statements;
END;
```

Example:
Creating a trigger that logs changes to the `TeluguMovies` table.

```
CREATE TRIGGER LogMovieChanges
ON TeluguMovies
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    INSERT INTO MovieLog (ChangeType, MovieID, ChangeDate)
    VALUES (CASE
                WHEN EXISTS (SELECT * FROM INSERTED) AND EXIST
S (SELECT * FROM DELETED) THEN 'UPDATE'
                WHEN EXISTS (SELECT * FROM INSERTED) THEN 'INS
ERT'
                WHEN EXISTS (SELECT * FROM DELETED) THEN 'DELE
TE'
            END,
            COALESCE((SELECT MovieID FROM INSERTED), (SELECT M
ovieID FROM DELETED)),
            GETDATE());
END;
```

Real-life Example:
A trigger is like a notification system that alerts you whenever specific changes happen in your database.

Transactions

Transactions are used to ensure that a series of SQL operations are executed as a single unit of work. They follow the ACID properties.

ACID Properties

1. **Atomicity**: Ensures that all operations within a transaction are completed successfully. If not, the transaction is aborted and no changes are made.
2. **Consistency**: Ensures that a transaction brings the database from one valid state to another.
3. **Isolation**: Ensures that concurrent transactions do not affect each other.
4. **Durability**: Ensures that the results of a transaction are permanently stored in the system.

Real-life Example:
Consider a banking system where transferring money from one account to another must be a single transaction to ensure the money is either fully transferred or not at all.

COMMIT, ROLLBACK, and SAVEPOINT

COMMIT

The `COMMIT` statement is used to save all changes made during the current transaction.

Syntax:

```
COMMIT;
```

Example:

```
BEGIN TRANSACTION;
UPDATE TeluguMovies SET ReleaseYear = 2020 WHERE MovieID = 1;
COMMIT;
```

ROLLBACK

The `ROLLBACK` statement is used to undo changes made during the current transaction.

Syntax:

```
ROLLBACK;
```

Example:

```
BEGIN TRANSACTION;
UPDATE TeluguMovies SET ReleaseYear = 2020 WHERE MovieID = 1;
ROLLBACK;
```

SAVEPOINT

The `SAVEPOINT` statement sets a point within a transaction to which you can later roll back.

Syntax:

```
SAVEPOINT savepoint_name;
```

Example:

```
BEGIN TRANSACTION;
UPDATE TeluguMovies SET ReleaseYear = 2020 WHERE MovieID = 1;
SAVEPOINT Savepoint1;
UPDATE TeluguMovies SET Genre = 'Drama' WHERE MovieID = 2;
ROLLBACK TO Savepoint1;
COMMIT;
```

Real-life Example:
Using
`COMMIT`, `ROLLBACK`, and `SAVEPOINT` is like writing in pencil. You can save your work (COMMIT), erase mistakes (ROLLBACK), and set bookmarks to go back to (SAVEPOINT).

Performance Tuning and Optimization

Query Optimization

Query optimization is the process of improving the efficiency of SQL queries. This involves using strategies to reduce the time and resources required to execute queries.

Understanding Execution Plans

An execution plan is a roadmap for how SQL Server will execute a query. It helps you understand the steps SQL Server takes to retrieve or modify data. Execution plans can be viewed using SQL Server Management Studio (SSMS).

Example:

```
EXPLAIN SELECT Title, ReleaseYear
FROM TeluguMovies
WHERE Director = 'S. S. Rajamouli';
```

Steps to View Execution Plan:

1. In SSMS, write the query you want to optimize.
2. Click on "Display Estimated Execution Plan" or "Include Actual Execution Plan."
3. Execute the query to see the plan.

Real-life Example:
Reading an execution plan is like following a recipe to see each step involved in cooking a dish. It helps identify where you might save time or improve efficiency.

Index Optimization

Indexes improve query performance by allowing faster data retrieval. However, too many indexes or poorly designed indexes can degrade performance.

Creating Effective Indexes:

1. **Use Indexes on Columns Frequently Used in WHERE, JOIN, and ORDER BY Clauses.**

   ```
   CREATE INDEX idx_director ON TeluguMovies (Director);
   ```

2. **Avoid Indexing Columns with High Cardinality (Many Unique Values).**
3. **Regularly Monitor and Maintain Indexes with REORGANIZE and REBUILD.**

   ```
   ALTER INDEX idx_director ON TeluguMovies REBUILD;
   ```

Real-life Example:
Using indexes effectively is like having an optimized filing system where frequently accessed files are easy to find, improving overall efficiency.

Database Normalization

Normalization is the process of organizing data to minimize redundancy and improve data integrity. It involves dividing large tables into smaller, related tables.

Normal Forms

1. **First Normal Form (1NF):** Ensures that each column contains atomic (indivisible) values and each entry in a column is of the same data type.

   Example:
   Before 1NF:

MovieID	Director
1	S. S. Rajamouli, S. S. Rajamouli

   After 1NF:

MovieID	Director
1	S. S. Rajamouli
1	S. S. Rajamouli

2. **Second Normal Form (2NF):** Achieves 1NF and ensures that non-key columns are fully dependent on the primary key.

   Example:
   Before 2NF:

MovieID	Title	DirectorID	DirectorName
1	Baahubali	101	S. S. Rajamouli
2	Eega	101	S. S. Rajamouli

   After 2NF:

MovieID	Title	DirectorID
1	Baahubali	101
2	Eega	101

DirectorID	DirectorName
101	S. S. Rajamouli

3. **Third Normal Form (3NF):** Achieves 2NF and ensures that non-key columns are not dependent on other non-key columns.

Example:
Before 3NF:

MovieID	Title	DirectorID	DirectorName	DirectorAge
1	Baahubali	101	S. S. Rajamouli	47

After 3NF:

MovieID	Title	DirectorID
1	Baahubali	101

DirectorID	DirectorName	DirectorAge
101	S. S. Rajamouli	47

4. **Boyce-Codd Normal Form (BCNF):** A stronger version of 3NF, ensures every determinant is a candidate key.

   Example:
   Consider a table where one column depends on another non-primary key:

MovieID	Title	DirectorID	Country
1	Baahubali	101	India

If DirectorID determines Country, split the table:

MovieID	Title	DirectorID
1	Baahubali	101

DirectorID	Country
101	India

Real-life Example:
Normalization is like organizing your library by genres, authors, and publication years to avoid duplication and ensure easy access to books.

Common Performance Issues

Understanding and identifying common performance issues helps maintain an efficient database system.

Identifying and Resolving Bottlenecks

1. **Slow Queries:Resolution:**

- Optimize queries by reducing complexity.
- Use indexes appropriately.

Example:
Identifying a slow query:

```
SELECT Title, ReleaseYear
FROM TeluguMovies
WHERE Director = 'S. S. Rajamouli';
```

Add an index:

```
CREATE INDEX idx_director ON TeluguMovies (Director);
```

2. **Locking and Blocking:Resolution:**
   - Reduce transaction scope and duration.
   - Use appropriate isolation levels.

   Example:
   Using a transaction with a smaller scope:

```
BEGIN TRANSACTION;
UPDATE TeluguMovies
SET ReleaseYear = 2021
WHERE MovieID = 1;
COMMIT;
```

3. **Insufficient Hardware Resources:Resolution:**
   - Upgrade hardware.
   - Optimize resource usage.
4. **High Disk I/O:Resolution:**
   - Optimize queries to reduce disk access.
   - Use indexing and caching strategies.
5. **Suboptimal Schema Design:Resolution:**
   - Normalize database schema.
   - Use appropriate data types and constraints.

Real-life Example:
Resolving performance bottlenecks is like fixing a traffic jam by identifying the cause (slow cars, roadblocks) and taking corrective actions (building a bypass, managing traffic flow).

SQL Security

User Management

Effective user management is crucial for database security. It involves creating and managing database users and ensuring that only authorized individuals have access to the database.

Creating and Managing Users

Creating users and assigning appropriate privileges helps control access to the database.

Creating a User

Syntax:

```
CREATE USER 'username'@'host' IDENTIFIED BY 'password';
```

Example:
Creating a user named `telugu_movie_admin`:

```
CREATE USER 'telugu_movie_admin'@'localhost' IDENTIFIED BY 'se
curepassword';
```

Managing Users

You can manage users by changing their passwords, renaming them, or deleting them.

Changing User Password

```
ALTER USER 'telugu_movie_admin'@'localhost' IDENTIFIED BY 'new
securepassword';
```

Renaming a User

```
RENAME USER 'telugu_movie_admin'@'localhost' TO 'telugu_movie_
manager'@'localhost';
```

Deleting a User

```
DROP USER 'telugu_movie_manager'@'localhost';
```

Real-life Example:
Creating and managing users is like giving employees their own access cards to enter different parts of a building, and ensuring their access is updated or revoked as needed.

Roles and Permissions

Roles simplify the management of permissions by grouping privileges and assigning them to users.

Granting and Revoking Privileges

Granting Privileges

Syntax:

```
GRANT privilege_name ON database_name.table_name TO 'usernam
e'@'host';
```

Example:
Granting SELECT and INSERT privileges to `telugu_movie_admin` on the `TeluguMoviesDB`:

```
GRANT SELECT, INSERT ON TeluguMoviesDB.* TO 'telugu_movie_admi
n'@'localhost';
```

Revoking Privileges

Syntax:

```
REVOKE privilege_name ON database_name.table_name FROM 'userna
me'@'host';
```

Example:
Revoking INSERT privilege from

`telugu_movie_admin`:

```
REVOKE INSERT ON TeluguMoviesDB.* FROM 'telugu_movie_admin'@'l
ocalhost';
```

Creating Roles

Roles are created to group specific privileges and then granted to users.

Syntax:

```
CREATE ROLE 'role_name';
GRANT privilege_name ON database_name.table_name TO 'role_nam
e';
GRANT 'role_name' TO 'username'@'host';
```

Example:
Creating a role for read-only access and assigning it to a user:

```
CREATE ROLE 'read_only';
GRANT SELECT ON TeluguMoviesDB.* TO 'read_only';
GRANT 'read_only' TO 'telugu_movie_viewer'@'localhost';
```

Real-life Example:
Granting and revoking privileges is like giving an employee keys to certain rooms in a building and taking back keys when they no longer need access.

Will see you back with some interview questions shortly :)