Day 1 Tuesday: SQL Fundamentals:

1. Create New Database

To get started with any database-related task, we must first create a database. The command to do so is:

```
CREATE DATABASE my_database;
```

This command creates an empty database that we can later populate with tables and data.

2. Create New Table

Once a database is created, we define the structure of our data using tables. Here's an example to create a table for storing employee details:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    hire_date DATE,
    salary DECIMAL(10, 2),
    department_id INT
);
```

This defines the `employees` table, specifying column names and their data types.

3. Insert Records

With our table in place, we can insert records (data) into it using the `INSERT INTO` statement. For example:

```
INSERT INTO employees (employee_id, first_name, last_name, hire_date, salary, department_id)
VALUES (1, 'John', 'Doe', '2020-01-15', 50000.00, 101);
```

This statement adds a new record into the `employees` table.

4. SQL SELECT Statement

To retrieve data from a table, we use the `SELECT` statement. For example:

SELECT * FROM employees;

This returns all columns and rows from the `employees` table.

5. SQL Distinct Statement

The `DISTINCT` keyword allows us to filter out duplicate values from the result set. Example:

SELECT DISTINCT department_id FROM employees;

This returns only unique department IDs from the `employees` table.

6. Where Clause

To filter records based on specific conditions, we use the `WHERE` clause. Example:

SELECT * FROM employees WHERE salary > 60000;

This retrieves employees with a salary greater than 60,000.

7. Order By Clause

The `ORDER BY` clause is used to sort the result set. By default, sorting is in ascending order, but we can also specify descending order:

SELECT * FROM employees ORDER BY salary DESC;

This orders employees by salary in descending order.

8. AND Operator

The `AND` operator allows us to combine multiple conditions in a `WHERE` clause. For example:

SELECT * FROM employees WHERE salary > 50000 AND department_id = 101;

This retrieves employees with a salary greater than 50,000 and who belong to department 101.

9. OR Operator

The `OR` operator is used when we want to satisfy at least one of multiple conditions. Example:

SELECT * FROM employees WHERE department_id = 101 OR department_id = 102;

This returns employees working in either department 101 or 102.

10. NOT Operator

The `NOT` operator is used to exclude records that match a specific condition:

SELECT * FROM employees WHERE NOT department_id = 101;

This retrieves all employees except those in department 101.

11. IN Operator

The `IN` operator simplifies querying for multiple values in a column:

SELECT * FROM employees WHERE department_id IN (101, 102, 103);

This fetches employees working in any of the listed departments.

12. BETWEEN Operator

The `BETWEEN` operator is used to filter records within a certain range. Example:

SELECT * FROM employees WHERE salary BETWEEN 40000 AND 70000;

This retrieves employees with salaries between 40,000 and 70,000.

13. LIKE Operator

The `LIKE` operator helps us find records with matching patterns. Example:

SELECT * FROM employees WHERE first_name LIKE 'J%';

This retrieves employees whose first name starts with 'J'.

14. MAX Function

The `MAX()` function returns the maximum value from a specified column:

SELECT MAX(salary) AS max_salary FROM employees;

This fetches the highest salary in the `employees` table.

15. MIN Function

The `MIN()` function returns the minimum value from a specified column:

SELECT MIN(salary) AS min_salary FROM employees;

This retrieves the lowest salary in the `employees` table.

16. SUM Function

The `SUM()` function calculates the total of a numeric column:

SELECT SUM(salary) AS total_salary FROM employees;

This computes the total salary of all employees.

17. AVG Function

The `AVG()` function returns the average value from a column:

SELECT AVG(salary) AS average_salary FROM employees;

This calculates the average salary of all employees.

18. COUNT Function

The `COUNT()` function counts the number of rows or non-null values in a column:

SELECT COUNT(*) AS employee_count FROM employees;

This returns the total number of employees in the table.

19. NOT NULL Constraint

The `NOT NULL` constraint ensures that a column cannot have a `NULL` value. Example:

CREATE TABLE employees (

    employee_id INT PRIMARY KEY,

    first_name VARCHAR(50) NOT NULL,

    last_name VARCHAR(50) NOT NULL

);

This ensures that both `first_name` and `last_name` must have values when inserting records.

20. Unique Constraint

The `UNIQUE` constraint ensures that all values in a column are unique:

CREATE TABLE employees (

    employee_id INT PRIMARY KEY,

    email VARCHAR(100) UNIQUE

);

This ensures that each email address in the table is unique.

21. Primary Key

A `PRIMARY KEY` uniquely identifies each record in a table. In the following example, the `employee_id` is set as the primary key:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50)
);
```

This ensures that each employee has a unique identifier.

22. Foreign Key

A `FOREIGN KEY` is used to create a relationship between two tables. Example:

```
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

The `department_id` in `employees` table is a foreign key that references `department_id` in the `departments` table.

Day 2-Thursday: Advanced SQL Topics:

1. SQL Check Constraint

The `CHECK` constraint is used to limit the values that can be inserted into a column. It ensures that the data satisfies certain conditions before being entered into the table.

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    salary DECIMAL(10, 2),
    hire_date DATE,
    CHECK (salary > 0)
);
```

In this example, the `CHECK` constraint ensures that only positive values for the `salary` column are allowed.

2. SQL ASC Command

The `ASC` (ascending) keyword is used with the `ORDER BY` clause to sort the results in ascending order. This is the default sorting order in SQL.

```
SELECT * FROM employees ORDER BY salary ASC;
```

This will return the employees sorted by salary in ascending order (from lowest to highest).

3. SQL DESC Command

The `DESC` (descending) keyword is used with the `ORDER BY` clause to sort the results in descending order.

```
SELECT * FROM employees ORDER BY salary DESC;
```

This will return the employees sorted by salary in descending order (from highest to lowest).

4. SQL ALTER TABLE Statement

The `ALTER TABLE` statement allows you to modify an existing table. You can use it to add, delete, or modify columns in a table.

Add a Column:

```
ALTER TABLE employees ADD email VARCHAR(100);
```

This adds a new column `email` to the `employees` table.

Modify a Column:

```
ALTER TABLE employees MODIFY salary DECIMAL(12, 2);
```

This changes the data type of the `salary` column to allow for more precision (12 digits in total, with 2 after the decimal point).

Drop a Column:

ALTER TABLE employees DROP COLUMN email;
This removes the `email` column from the `employees` table.

5. SQL UPDATE Statement

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

UPDATE employees
SET salary = 55000
WHERE employee_id = 1;
This updates the salary of the employee with `employee_id = 1` to 55,000.
Multiple Column Update:
UPDATE employees
SET salary = 60000, department_id = 102
WHERE employee_id = 1;

This updates both the `salary` and `department_id` for the employee with `employee_id = 1`.

6. SQL Aliases

SQL aliases are used to give a table or column a temporary name. This makes queries easier to read, especially when dealing with complex joins or calculations.

Column Alias:

SELECT first_name AS "First Name", last_name AS "Last Name", salary AS "Annual Salary"

FROM employees;

In this example, we are renaming the `first_name`, `last_name`, and `salary` columns to "First Name", "Last Name", and "Annual Salary" in the result set.

Table Alias:

SELECT e.first_name, e.last_name, e.salary

FROM employees AS e;

Here, `employees` is aliased as `e`. This makes the query more concise.

7. SQL Stored Procedures

A **Stored Procedure** is a set of SQL statements that can be executed as a single unit. Stored procedures are stored in the database and can be reused for operations such as data manipulation or business logic.

Creating a Stored Procedure:

```
CREATE PROCEDURE update_salary (IN emp_id INT, IN new_salary DECIMAL(10,2))
BEGIN
    UPDATE employees
    SET salary = new_salary
    WHERE employee_id = emp_id;
END;
```

In this example, we created a stored procedure called `update_salary` that takes two parameters: `emp_id` and `new_salary`. It updates the salary of the employee with the given `emp_id`.

Calling a Stored Procedure:

```
CALL update_salary(1, 60000);
```

This will call the `update_salary` procedure and set the salary of employee `1` to 60,000.

Benefits of Stored Procedures:

- **Modularity:** Stored procedures allow you to encapsulate logic, making it reusable.
- **Security:** You can restrict direct access to tables and provide controlled access via procedures.
- **Performance:** Since stored procedures are precompiled, they can be more efficient than executing raw queries.

Day 3_Friday SQL Advanced Operations:

1. SQL CREATE INDEX Statement

The `CREATE INDEX` statement is used to create an index on a table to speed up the retrieval of rows. Indexes are particularly useful when dealing with large tables and frequent searches.

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

This creates an index named `idx_salary` on the `salary` column in the `employees` table. Now, queries filtering or sorting by `salary` will be faster.

Creating a Unique Index:

CREATE UNIQUE INDEX idx_email

ON employees(email);

This ensures that the `email` column contains unique values and improves the performance of queries that use the `email` column for lookups.

Dropping an Index:

DROP INDEX idx_salary ON employees;

This removes the index `idx_salary` from the `employees` table.

2. SQL SELECT INTO Statement

The `SELECT INTO` statement is used to create a new table and insert the result of a `SELECT` query into it. It is often used for creating backups or making temporary tables.

Example:

SELECT * INTO employees_backup

FROM employees;

This creates a new table `employees_backup` and copies all rows from the `employees` table into it.

With Filtering (Selective Insert):

SELECT employee_id, first_name, last_name, salary

INTO high_salary_employees

FROM employees

WHERE salary > 70000;

This creates a table `high_salary_employees` containing only employees with a salary greater than 70,000.

3. SQL SELECT TOP Clause

The `SELECT TOP` clause is used to limit the number of rows returned by a query. This is particularly useful when working with large datasets or for performing sampling.

Example:

SELECT TOP 5 * FROM employees ORDER BY salary DESC;

This query returns the top 5 employees with the highest salaries.

With Percentage:

SELECT TOP 10 PERCENT * FROM employees ORDER BY hire_date DESC;

This returns the top 10% of the employees based on the most recent hire dates.

4. Backup Database in SQL

Backing up a database is critical to prevent data loss. SQL Server and other database systems provide mechanisms for backing up a database.

Full Backup:

BACKUP DATABASE my_database

TO DISK = 'C:\backup\my_database.bak';

This creates a full backup of `my_database` and saves it to the specified location (`C:\backup\`).

Differential Backup:

A **differential backup** backs up only the changes made since the last full backup.

BACKUP DATABASE my_database

TO DISK = 'C:\backup\my_database_diff.bak'

WITH DIFFERENTIAL;

Transaction Log Backup:

For point-in-time recovery, a **transaction log backup** is used.

BACKUP LOG my_database

TO DISK = 'C:\backup\my_database_log.trn';

5. SQL Views

A **View** is a virtual table based on the result of a `SELECT` query. Views do not store data themselves; they store the SQL query that retrieves data from the underlying tables. They can be used to simplify complex queries or provide a security layer by restricting access to certain columns or rows.

Creating a View:

CREATE VIEW high_salary_employees_view AS

SELECT employee_id, first_name, last_name, salary

FROM employees

WHERE salary > 70000;

This creates a view called `high_salary_employees_view` that contains a subset of employees earning more than 70,000.

Using a View in a Query:

SELECT * FROM high_salary_employees_view;

This retrieves all employees from the `high_salary_employees_view` view.

Dropping a View:

DROP VIEW high_salary_employees_view;

This removes the `high_salary_employees_view` from the database.

6. Drop a Table in SQL

The `DROP TABLE` statement is used to permanently delete a table and all of its data from the database.

Example:

DROP TABLE employees_backup;

This will permanently remove the `employees_backup` table from the database.