Basic SQL Concepts:

1. What is SQL, and why is it important in data analytics?

SQL, or **Structured Query Language**, is a standardized programming language specifically designed **for managing and manipulating relational databases**. It allows users to perform a wide range of operations on database data, such as **querying, updating, inserting, and deleting records**.

Importance of SQL in Data Analytics:

1. **Efficient Data Retrieval**:
   - SQL enables analysts to extract data quickly from large databases, which is essential for analytics workflows.
2. **Data Integration**:
   - SQL facilitates the combination of data from multiple tables or even different databases using operations like **JOIN.**
3. **Scalability**:
   - SQL is used in a wide range of databases, from small-scale applications to enterprise-level systems, making it versatile and suitable for various analytics needs.
4. **Data Transformation**:
   - SQL provides tools for cleaning, aggregating, and restructuring data, making it ready for analysis.
5. **Standardization**:
   - As a standardized language, SQL is universally understood, ensuring compatibility across platforms.
6. **Decision Support**:
   - SQL queries provide the backbone for business intelligence tools, enabling organizations to make data-driven decisions.
7. **Automation**:
   - SQL scripts can automate repetitive data tasks, reducing manual effort and increasing productivity

2. Explain the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN.

In SQL, **JOIN** operations are used to combine rows from two or more tables based on a related column. The type of join determines which rows are included in the result. Here's a breakdown of the main types of joins:

---

1. **INNER JOIN**

- **Definition**: Returns rows that have matching values in both tables.
- **Use Case**: Use when you only want records that exist in **both** tables.
- **Diagram**: **Intersection** of the two tables.

Example:

```
SELECT A.column1, B.column2
```

```
FROM TableA A
INNER JOIN TableB B ON A.common_column = B.common_column;
```

- **Result**: Only rows where A.common_column = B.common_column.

---

2. LEFT JOIN (LEFT OUTER JOIN)

- **Definition**: Returns all rows from the **left table** (TableA), and the matching rows from the right table (TableB). If no match is found, NULLs are returned for columns from TableB.
- **Use Case**: Use when you want to include all rows from the **left table**, regardless of whether there's a match in the right table.
- **Diagram**: Left side of the Venn diagram (left table and intersection).

Example:

```
SELECT A.column1, B.column2
FROM TableA A
LEFT JOIN TableB B ON A.common_column = B.common_column;
```

- **Result**: All rows from TableA, with matching rows from TableB or NULL.

---

3. RIGHT JOIN (RIGHT OUTER JOIN)

- **Definition**: Returns all rows from the **right table** (TableB), and the matching rows from the left table (TableA). If no match is found, NULLs are returned for columns from TableA.
- **Use Case**: Use when you want to include all rows from the **right table**, regardless of whether there's a match in the left table.
- **Diagram**: Right side of the Venn diagram (right table and intersection).

Example:

```
SELECT A.column1, B.column2
FROM TableA A
RIGHT JOIN TableB B ON A.common_column = B.common_column;
```

- **Result**: All rows from TableB, with matching rows from TableA or NULL.

---

4. FULL OUTER JOIN

- **Definition**: Returns all rows when there is a match in either TableA or TableB. If no match is found, NULLs are returned for columns from the non-matching table.
- **Use Case**: Use when you want to include **all rows** from both tables, with matches where possible.
- **Diagram**: Entire Venn diagram (left table, right table, and intersection).

Example:

```
SELECT A.column1, B.column2
FROM TableA A
FULL OUTER JOIN TableB B ON A.common_column = B.common_column;
```

- **Result**: All rows from TableA and TableB, with NULLs where no match exists.

3. What is the difference between WHERE and HAVING clauses?

The **WHERE** and **HAVING** clauses in SQL are both used to filter data in a query, but they are applied at different stages of query execution and are used for different purposes.

1. WHERE Clause

- **Purpose**: Filters rows **before** any grouping or aggregation occurs.
- **Scope**: Works on individual rows of data.
- **Common Use Cases**: Used to filter raw data based on column values or conditions.

Example:

```
SELECT name, age
FROM employees
WHERE age > 30;
```

This filters rows where the age is greater than 30 **before** any further processing like grouping or aggregation.

2. HAVING Clause

- **Purpose**: Filters groups of rows **after** aggregation or grouping has been performed.
- **Scope**: Works on aggregated data (e.g., results of COUNT, SUM, AVG, etc.).
- **Common Use Cases**: Used to apply conditions to aggregated values (e.g., totals, averages).

Example:

```
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;
```

- This filters out groups where the employee_count is 5 or fewer **after** the rows have been grouped by department.

4. How do you use GROUP BY and HAVING in a query?

The **GROUP BY** and **HAVING** clauses in SQL are used together to group rows of data and filter aggregated results.

1. Purpose of GROUP BY

- **GROUP BY** is used to group rows based on one or more columns. Once grouped, aggregate functions (like SUM, COUNT, AVG, etc.) can be applied to each group.
- **Example**: Summing sales amounts by region.

2. Purpose of HAVING

- **HAVING** is used to filter groups after aggregation. It allows you to specify conditions based on the aggregated values.
- **Example**: Filtering regions where total sales exceed a certain value.

5. Write a query to find duplicate records in a table.

- use the **GROUP BY** clause along with an aggregate function like **COUNT** to identify rows that occur more than once based on specific columns.
  Example:
  SELECT column1, column2, COUNT(*) AS count
  FROM table_name
  GROUP BY column1, column2
  HAVING COUNT(*) > 1;

6. How do you retrieve unique values from a table using SQL

DISTINCT -Removing duplicates in a simple, single-query scenario.

GROUP BY -When you need aggregation or grouping logic in addition to uniqueness.

ROW_NUMBER() -Selecting specific rows (e.g., the first occurrence) from duplicate groups.

UNION -Combining unique rows from multiple queries or tables.

7. **Explain the use of aggregate functions like COUNT(), SUM(), AVG(), MIN(), and MAX().**

Aggregate functions in SQL are used to perform calculations on a set of rows and return a single value. They are commonly used in combination with the **GROUP BY** clause to group rows and perform calculations on each group.

1. COUNT():

The **COUNT()** function is used to count the number of rows in a result set or the number of non-null values in a specific column.

Syntax:
SELECT COUNT(column_name) FROM table_name;

Example:
SELECT COUNT(*) FROM employees;

This will return the total number of rows in the employees table.

To count the number of non-null values in a specific column:

SELECT COUNT(department) FROM employees;

This will count how many department entries are not null.

---

2. SUM():

The **SUM()** function calculates the total sum of a numeric column.

Syntax:
SELECT SUM(column_name) FROM table_name;

Example:
SELECT SUM(salary) FROM employees;

This will return the total salary paid to all employees.

---

3. AVG():

The **AVG()** function calculates the average value of a numeric column.

Syntax:
SELECT AVG(column_name) FROM table_name;

Example:
SELECT AVG(salary) FROM employees;

This will return the average salary of all employees.

---

4. MIN():

The **MIN()** function returns the smallest value in a specified column.

Syntax:
SELECT MIN(column_name) FROM table_name;

Example:
SELECT MIN(salary) FROM employees;

This will return the lowest salary in the employees table.

---

5. MAX():

The **MAX()** function returns the largest value in a specified column.

Syntax:
SELECT MAX(column_name) FROM table_name;

Example:
SELECT MAX(salary) FROM employees;

This will return the highest salary in the employees table.

Using Aggregate Functions with GROUP BY:

Aggregate functions are often used with the GROUP BY clause to calculate values for each group of rows.

Example:

If you want to find the average salary for each department:

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department;
```

This query will return the average salary for each department, grouping by the department column.

Function	Description	Example
COUNT()	Returns the number of rows or non-null values.	SELECT COUNT(*) FROM employees;
SUM()	Returns the sum of a numeric column.	SELECT SUM(salary) FROM employees;
AVG()	Returns the average value of a numeric column.	SELECT AVG(salary) FROM employees;
MIN()	Returns the smallest value in a column.	SELECT MIN(salary) FROM employees;
MAX()	Returns the largest value in a column.	SELECT MAX(salary) FROM employees;

These aggregate functions are very useful for summarizing data, calculating totals, averages, and identifying extremes in datasets. They are often used in reporting and analytics queries.

8. What is the purpose of a DISTINCT keyword in SQL?

The **DISTINCT** keyword in SQL is used to remove duplicate rows from the result set and return only unique values for the selected columns. When you use DISTINCT, it ensures that the result includes each combination of values from the specified columns only once.

Purpose:

- To **filter out duplicates** and retrieve only unique values.
- Useful when you want to see a list of distinct values in one or more columns, without repeating the same values.

SYNTAX:

```
SELECT DISTINCT column1, column2, ...

FROM table_name;
```

- **DISTINCT applies to all the columns listed** in the SELECT statement. It returns only unique combinations of the selected columns.
- It is typically used when you want to remove duplicate rows or values from the result.
- If DISTINCT is used on a single column, it returns unique values from that column.