Q1. What is SQL, and why is it important in data analytics?

SQL (Structured Query Language) is a programming language designed for managing and manipulating relational databases. It is widely used to interact with databases to perform operations such as querying, inserting, updating, and deleting data. SQL also enables the creation and modification of database structures like tables, views, and indexes.

SQL is crucial in data analytics for several reasons:

1. **Data Access and Retrieval**
   * SQL enables analysts to extract specific data from large datasets efficiently.
   * It supports querying data with filters, aggregations, and sorting.
2. **Data Manipulation**
   * SQL provides tools to clean, transform, and prepare data for analysis.
   * Analysts can combine data from multiple tables using joins, create calculated columns, and normalize data.
3. **Widespread Use in Databases**
   * Relational databases like MySQL, PostgreSQL, Microsoft SQL Server, and Oracle use SQL as their core language.
   * Knowledge of SQL allows analysts to work across multiple database systems.
4. **Efficiency and Scalability**
   * SQL is optimized for handling large volumes of data efficiently, which is critical for analytics in business intelligence.
5. **Integration with Tools**
   * Many data analytics tools (e.g., Tableau, Power BI) integrate seamlessly with SQL databases, enabling direct querying and visualization.
6. **Foundational Skill for Data Roles**
   * SQL is a fundamental skill for data analysts, data scientists, and database administrators, making it an essential part of the analytics workflow.

**Example SQL Query:**

SELECT department, AVG(salary) AS avg\_salary

FROM employees

WHERE hire\_date > '2020-01-01'

GROUP BY department

ORDER BY avg\_salary DESC;

Q2. Explain the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN

**SQL Joins: Differences Between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN**

SQL joins allow you to combine rows from two or more tables based on related columns. Here's a breakdown of the differences among these types of joins:

**1. INNER JOIN**

* **Definition**: Returns rows that have matching values in both tables.
* **Result**: Only rows where the join condition is true are included.
* **Use Case**: When you only need data that exists in both tables.

**Syntax:**

SELECT columns

FROM table1

INNER JOIN table2

ON table1.common\_column = table2.common\_column;

**Example:**

SELECT employees.name, departments.dept\_name

FROM employees

INNER JOIN departments

ON employees.dept\_id = departments.dept\_id;

* Returns employees who belong to a department.

**2. LEFT JOIN (or LEFT OUTER JOIN)**

* **Definition**: Returns all rows from the left table and the matched rows from the right table. Rows in the left table without matches are included with NULL values for the right table's columns.
* **Result**: All rows from the left table, matched or unmatched, are returned.
* **Use Case**: When you want all data from the left table, even if there are no matches in the right table.

**Syntax:**

SELECT columns

FROM table1

LEFT JOIN table2

ON table1.common\_column = table2.common\_column;

**Example:**

SELECT employees.name, departments.dept\_name

FROM employees

LEFT JOIN departments

ON employees.dept\_id = departments.dept\_id;

* Includes employees who don't belong to any department.

**3. RIGHT JOIN (or RIGHT OUTER JOIN)**

* **Definition**: Returns all rows from the right table and the matched rows from the left table. Rows in the right table without matches are included with NULL values for the left table's columns.
* **Result**: All rows from the right table, matched or unmatched, are returned.
* **Use Case**: When you want all data from the right table, even if there are no matches in the left table.

**Syntax:**

SELECT columns

FROM table1

RIGHT JOIN table2

ON table1.common\_column = table2.common\_column;

**Example:**

SELECT employees.name, departments.dept\_name

FROM employees

RIGHT JOIN departments

ON employees.dept\_id = departments.dept\_id;

* Includes departments that have no employees.

**4. FULL OUTER JOIN**

* **Definition**: Returns all rows from both tables. Unmatched rows in either table are included with NULL values for the missing columns.
* **Result**: A combination of LEFT JOIN and RIGHT JOIN.
* **Use Case**: When you need all data from both tables, regardless of matches.

**Syntax:**

SELECT columns

FROM table1

FULL OUTER JOIN table2

ON table1.common\_column = table2.common\_column;

**Example:**

SELECT employees.name, departments.dept\_name

FROM employees

FULL OUTER JOIN departments

ON employees.dept\_id = departments.dept\_id;

* Includes all employees and departments, even if no match exists.

**Visual Representation:**

| **Type** | **Matched Rows** | **Unmatched Rows (Left Table)** | **Unmatched Rows (Right Table)** |
| --- | --- | --- | --- |
| **INNER JOIN** | ✅ | ❌ | ❌ |
| **LEFT JOIN** | ✅ | ✅ | ❌ |
| **RIGHT JOIN** | ✅ | ❌ | ✅ |
| **FULL OUTER JOIN** | ✅ | ✅ | ✅ |

Q3. What is the difference between WHERE and HAVING clauses?

**Difference Between WHERE and HAVING Clauses in SQL**

Both WHERE and HAVING are used to filter records in SQL queries, but they differ in how and where they are applied.

**1. WHERE Clause**

* **Purpose**: Filters rows before any grouping or aggregation occurs.
* **Scope**: Operates on individual rows in a table.
* **Used With**: Columns, conditions, and simple filters.
* **Cannot Use**: Aggregate functions (e.g., SUM(), AVG()).

**Syntax:**

SELECT columns

FROM table

WHERE condition;

**Example:**

SELECT name, salary

FROM employees

WHERE salary > 50000;

* This filters rows where the salary is greater than 50,000 before any grouping.

**2. HAVING Clause**

* **Purpose**: Filters rows after grouping or aggregation has been performed.
* **Scope**: Operates on aggregated data (results of GROUP BY).
* **Used With**: Aggregate functions (e.g., SUM(), COUNT(), AVG()).
* **Must Follow**: A GROUP BY clause if used.

**Syntax:**

SELECT columns, aggregate\_function

FROM table

GROUP BY column

HAVING condition;

**Example:**

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

HAVING AVG(salary) > 60000;

* This filters departments where the average salary exceeds 60,000, after grouping employees by department.

**Key Differences**

| **Feature** | **WHERE Clause** | **HAVING Clause** |
| --- | --- | --- |
| **When Applied** | Before grouping/aggregation. | After grouping/aggregation. |
| **Filters** | Individual rows in the dataset. | Aggregated or grouped data. |
| **Aggregate Use** | Cannot use aggregate functions. | Can use aggregate functions. |
| **Use Case** | Filter raw data before processing. | Filter summarized data after processing. |

**Combined Example:**

To see the difference in action:

SELECT department, COUNT(\*) AS num\_employees

FROM employees

WHERE salary > 50000 -- Filters individual rows first

GROUP BY department

HAVING COUNT(\*) > 5; -- Filters grouped results

* **WHERE**: Filters employees with a salary greater than 50,000.
* **HAVING**: Filters departments with more than 5 employees (after counting them).

Q4. How do you use GROUP BY and HAVING in a query?

**Using GROUP BY and HAVING in a Query**

The GROUP BY and HAVING clauses are commonly used together in SQL to organize data into groups and filter aggregated results. Here's how they work:

**1. GROUP BY Clause**

* **Purpose**: Groups rows with the same values in specified columns into summary rows.
* **Use Case**: When you need aggregated results (e.g., SUM(), COUNT(), AVG()).
* **Requirement**: Must be used with aggregate functions or calculations.

**2. HAVING Clause**

* **Purpose**: Filters aggregated results generated by GROUP BY.
* **Use Case**: When you want to filter groups based on conditions applied to aggregate functions (e.g., SUM(salary) > 100000).
* **Position**: Comes after GROUP BY.

**Query Syntax**

SELECT column1, aggregate\_function(column2)

FROM table

GROUP BY column1

HAVING condition\_on\_aggregate\_function;

**Example Query**

**Problem: Find departments with an average salary greater than 60,000 and display their total salaries.**

SELECT department, AVG(salary) AS avg\_salary, SUM(salary) AS total\_salary

FROM employees

GROUP BY department

HAVING AVG(salary) > 60000;

**Steps:**

1. **GROUP BY department**: Groups employees by their department.
2. **AVG(salary)**: Calculates the average salary for each department.
3. **HAVING AVG(salary) > 60000**: Filters out departments where the average salary is 60,000 or less.

**Result:**

| **Department** | **Avg\_Salary** | **Total\_Salary** |
| --- | --- | --- |
| IT | 75,000 | 450,000 |
| HR | 65,000 | 325,000 |

**Combining with WHERE**

You can use WHERE to filter rows before grouping and HAVING to filter after grouping.

SELECT department, COUNT(\*) AS num\_employees

FROM employees

WHERE salary > 50000 -- Filters individual rows first

GROUP BY department

HAVING COUNT(\*) > 5; -- Filters grouped results

**Key Points:**

* **WHERE filters raw data**, while **HAVING filters grouped/aggregated data**.
* The **order of execution**: WHERE → GROUP BY → HAVING.

Q5. Write a query to find duplicate records in a table.

To find duplicate records in a table, you can use the GROUP BY clause along with an aggregate function like COUNT(). Here’s a query that identifies duplicate rows based on specific columns:

**Query:**

SELECT column1, column2, COUNT(\*) AS duplicate\_count

FROM table\_name

GROUP BY column1, column2

HAVING COUNT(\*) > 1;

**Explanation:**

1. **GROUP BY column1, column2**: Groups rows with the same values in column1 and column2.
2. **COUNT(\*)**: Counts the number of rows in each group.
3. **HAVING COUNT(\*) > 1**: Filters out groups with only one row, leaving only duplicates.

**Example:**

**Table: employees**

| **ID** | **Name** | **Department** |
| --- | --- | --- |
| 1 | Alice | IT |
| 2 | Bob | HR |
| 3 | Alice | IT |
| 4 | Carol | Finance |
| 5 | Bob | HR |

**Query:**

SELECT Name, Department, COUNT(\*) AS duplicate\_count

FROM employees

GROUP BY Name, Department

HAVING COUNT(\*) > 1;

**Result:**

| **Name** | **Department** | **Duplicate\_Count** |
| --- | --- | --- |
| Alice | IT | 2 |
| Bob | HR | 2 |

Q6. How do you retrieve unique values from a table using SQL?

To retrieve unique values from a table in SQL, you can use the DISTINCT keyword. This ensures that duplicate rows in the result set are removed, returning only unique values.

**Syntax:**

SELECT DISTINCT column1, column2, ...

FROM table\_name;

**Example 1: Retrieve Unique Values from One Column**

**Table: employees**

| **ID** | **Name** | **Department** |
| --- | --- | --- |
| 1 | Alice | IT |
| 2 | Bob | HR |
| 3 | Alice | IT |
| 4 | Carol | Finance |
| 5 | Bob | HR |

**Query:**

SELECT DISTINCT Department

FROM employees;

**Result:**

| **Department** |
| --- |
| IT |
| HR |
| Finance |

**Example 2: Retrieve Unique Combinations of Columns**

To retrieve unique combinations of values across multiple columns:

**Query:**

SELECT DISTINCT Name, Department

FROM employees;

**Result:**

| **Name** | **Department** |
| --- | --- |
| Alice | IT |
| Bob | HR |
| Carol | Finance |

**Key Points:**

* The DISTINCT keyword applies to all the columns in the SELECT statement. If you specify multiple columns, only rows with unique combinations of those column values will be returned.
* If you want unique rows across the entire table, include all columns in the SELECT statement.

Q7. Explain the use of aggregate functions like COUNT(), SUM(), AVG(), MIN(), and MAX().

**Aggregate Functions in SQL**

Aggregate functions are used to perform calculations on a set of values and return a single summary result. These functions are commonly used in conjunction with the GROUP BY clause for summarizing data.

**1. COUNT()**

* **Purpose**: Returns the number of rows in a result set.
* **Use Case**: To count rows, including or excluding NULL values.

**Syntax:**

SELECT COUNT(column\_name) FROM table\_name;

**Example:**

SELECT COUNT(\*) AS total\_employees FROM employees;

* Counts all employees in the table.

**2. SUM()**

* **Purpose**: Returns the total sum of a numeric column.
* **Use Case**: To calculate totals, such as sales, salaries, or revenue.

**Syntax:**

SELECT SUM(column\_name) FROM table\_name;

**Example:**

SELECT SUM(salary) AS total\_salary FROM employees;

* Calculates the total salary of all employees.

**3. AVG()**

* **Purpose**: Returns the average value of a numeric column.
* **Use Case**: To calculate averages, such as average salary or grades.

**Syntax:**

SELECT AVG(column\_name) FROM table\_name;

**Example:**

SELECT AVG(salary) AS avg\_salary FROM employees;

* Computes the average salary of employees.

**4. MIN()**

* **Purpose**: Returns the smallest value in a column.
* **Use Case**: To find the minimum value, such as the lowest price or smallest age.

**Syntax:**

SELECT MIN(column\_name) FROM table\_name;

**Example:**

SELECT MIN(salary) AS min\_salary FROM employees;

* Finds the lowest salary among employees.

**5. MAX()**

* **Purpose**: Returns the largest value in a column.
* **Use Case**: To find the maximum value, such as the highest salary or largest order.

**Syntax:**

SELECT MAX(column\_name) FROM table\_name;

**Example:**

SELECT MAX(salary) AS max\_salary FROM employees;

* Finds the highest salary among employees.

**Combined Example:**

**Problem: Find the total, average, minimum, and maximum salary for each department.**

SELECT department,

SUM(salary) AS total\_salary,

AVG(salary) AS avg\_salary,

MIN(salary) AS min\_salary,

MAX(salary) AS max\_salary

FROM employees

GROUP BY department;

**Result:**

| **Department** | **Total\_Salary** | **Avg\_Salary** | **Min\_Salary** | **Max\_Salary** |
| --- | --- | --- | --- | --- |
| IT | 300,000 | 75,000 | 60,000 | 90,000 |
| HR | 200,000 | 50,000 | 45,000 | 55,000 |

**Key Points:**

* **COUNT()** works with any data type, while **SUM()** and **AVG()** require numeric columns.
* **MIN()** and **MAX()** work with both numeric and non-numeric data (e.g., dates, strings).

Q8. What is the purpose of a DISTINCT keyword in SQL?

**Purpose of the DISTINCT Keyword in SQL**

The DISTINCT keyword in SQL is used to remove duplicate rows from the result set of a query. It ensures that only unique values or combinations of values are returned.

**When to Use DISTINCT**

1. **To Retrieve Unique Values in a Single Column**  
   Use DISTINCT to find all unique values in a specific column.

**Syntax:**

SELECT DISTINCT column\_name

FROM table\_name;

**Example:**

SELECT DISTINCT department

FROM employees;

* Retrieves a list of unique departments from the employees table.

1. **To Retrieve Unique Combinations of Multiple Columns**  
   Use DISTINCT to find unique combinations of values across multiple columns.

**Syntax:**

SELECT DISTINCT column1, column2

FROM table\_name;

**Example:**

SELECT DISTINCT department, job\_title

FROM employees;

* Retrieves unique combinations of departments and job titles.

1. **To Eliminate Duplicates in Entire Rows**  
   Use DISTINCT without specific columns to eliminate duplicates in all selected rows.

**Example:**

SELECT DISTINCT \*

FROM employees;

* Returns unique rows in the table.

**Key Points:**

* **Applies to All Columns in the SELECT Statement**: When multiple columns are selected, DISTINCT ensures that the combination of values across these columns is unique.
* **Order of Operations**: DISTINCT is applied before ORDER BY, so the uniqueness is established before sorting.
* **Performance Consideration**: Using DISTINCT can be resource-intensive for large datasets because the database must compare all rows.

**Example Scenario**

**Problem: Find all unique job titles in the employees table.**

SELECT DISTINCT job\_title

FROM employees;

**Result:**

| **Job\_Title** |
| --- |
| Manager |
| Software Engineer |
| Accountant |