**SQL (Structured Query Language)**

SQL (Structured Query Language) is a standard language used for managing and manipulating relational databases. It allows you to perform various operations like querying data, updating records, and managing database structure. Below is a breakdown of the key SQL commands and their differences:

**1. SQL Commands Overview**

SQL commands are divided into several categories based on the functionality they provide. These categories are:

* **DML (Data Manipulation Language)**: Deals with data manipulation (e.g., adding, modifying, and retrieving data).
* **DDL (Data Definition Language)**: Deals with database structure (e.g., creating and modifying tables).
* **DCL (Data Control Language)**: Deals with permissions and access control (e.g., granting or revoking access).
* **TCL (Transaction Control Language)**: Deals with managing transactions in a database (e.g., commits, rollbacks).

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

* **CREATE**: Used to create database objects like tables, views, or indexes.

CREATE TABLE Students (

ID INT PRIMARY KEY,

Name VARCHAR(100),

Age INT

);

* **ALTER**: Used to modify an existing database object, like adding or deleting columns.

ALTER TABLE Students ADD Gender VARCHAR(10);

* **DROP**: Deletes an entire database object, like a table or view.

DROP TABLE Students;

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

* **SELECT**: Retrieves data from the database.

SELECT Name, Age FROM Students WHERE Age > 18;

* **INSERT**: Adds new data into a table.

INSERT INTO Students (ID, Name, Age) VALUES (1, 'John Doe', 22);

* **UPDATE**: Modifies existing data in a table.

UPDATE Students SET Age = 23 WHERE ID = 1;

* **DELETE**: Removes data from a table.

DELETE FROM Students WHERE Age < 18;

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

* **GRANT**: Provides privileges to users (e.g., allowing them to SELECT, INSERT, etc.).

GRANT SELECT ON Students TO User1;

* **REVOKE**: Removes privileges from users.

REVOKE SELECT ON Students FROM User1;

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

* **COMMIT**: Saves changes made during the current transaction.

COMMIT;

* **ROLLBACK**: Undoes changes made during the current transaction.

ROLLBACK;

* **SAVEPOINT**: Sets a point in the transaction to which you can later roll back.

SAVEPOINT savepoint1;

**6. Differences Between SQL Commands**

|  |  |  |  |
| --- | --- | --- | --- |
| **Command Type** | **Purpose** | **Examples** | **Key Operations** |
| **DDL** | Deals with defining and modifying database structure. | CREATE, ALTER, DROP | Creates tables, modifies structures |
| **DML** | Deals with data manipulation. | SELECT, INSERT, UPDATE, DELETE | Retrieves, inserts, updates, deletes data |
| **DCL** | Deals with database security and permissions. | GRANT, REVOKE | Manages access control |
| **TCL** | Deals with transaction management. | COMMIT, ROLLBACK, SAVEPOINT | Manages transactions (commit/rollback) |

**All Important Question regarding SQL**

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

**What is SQL?**

**SQL (Structured Query Language)** is a programming language used to manage and manipulate data stored in relational databases. It enables users to:

* Retrieve data efficiently.
* Insert, update, and delete records in a database.
* Create and modify database structures (tables, views, etc.).
* Perform advanced operations like joins, filtering, and aggregations.

**Core Functions of SQL:**

1. **Data Retrieval:** Fetch specific data using SELECT statements.
2. **Data Manipulation:** Modify data using INSERT, UPDATE, and DELETE commands.
3. **Data Definition:** Create or alter database structures using CREATE and ALTER.
4. **Data Control:** Manage access to data using permissions and privileges.

**Why is SQL Important in Data Analytics?**

SQL plays a vital role in data analytics because it serves as the bridge between raw data stored in databases and actionable insights. Here’s why it is indispensable:

1. **Efficient Data Handling:**
   * SQL can quickly query large datasets, making it ideal for analytics.
   * It can filter, group, and sort data to identify trends and patterns.

**Example:**

SELECT department, SUM(sales) AS total\_sales

FROM sales\_data

GROUP BY department

ORDER BY total\_sales DESC;

This query identifies departments with the highest sales.

1. **Data Preparation:**
   * In analytics, raw data often requires cleaning, transforming, and preparing. SQL provides tools to achieve this efficiently.
   * Techniques like joins, subqueries, and aggregations can combine and transform data into a usable format.

**Example:**

SELECT e.employee\_name, d.department\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id;

This query combines employee and department data.

1. **Cross-Platform Use:**
   * SQL works with various database management systems (DBMS) like MySQL, PostgreSQL, Oracle, and Microsoft SQL Server, making it versatile.
2. **Foundation of BI Tools:**
   * Business Intelligence (BI) tools like Tableau, Power BI, and Looker rely on SQL for querying data.

**Example in BI:** A Tableau report that shows sales trends over time is often powered by an SQL query running in the background.

1. **Universal Adoption:**
   * Most organizations use SQL to manage and analyze data, making it a critical skill for data professionals.

**SQL in Real-World Data Analytics**

1. **Customer Insights:**
   * SQL helps analyze customer behavior, preferences, and purchase history.

**Example:**

SELECT customer\_id, COUNT(order\_id) AS orders\_count

FROM orders

GROUP BY customer\_id

HAVING orders\_count > 5;

This query identifies frequent customers.

1. **Sales Performance:**
   * Analyzing sales data over time or by region.

**Example:**

SELECT region, AVG(sales) AS avg\_sales

FROM sales\_data

GROUP BY region;

1. **Risk Management:**
   * SQL queries can identify potential risks like unpaid invoices or overdue accounts.

**Example:**

SELECT invoice\_id, due\_date

FROM invoices

WHERE due\_date < CURRENT\_DATE AND status = 'Pending';

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

Joins in SQL are used to combine rows from two or more tables based on a related column.

**Tables for Example**

We will use two example tables:

**Table 1: employees**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **name** | **dept\_id** |
| 1 | John | 10 |
| 2 | Alice | 20 |
| 3 | Bob | NULL |
| 4 | Eve | 30 |

**Table 2: departments**

|  |  |
| --- | --- |
| **dept\_id** | **dept\_name** |
| 10 | HR |
| 20 | Sales |
| 30 | IT |
| 40 | Finance |

**1. INNER JOIN**

* **Definition:** Combines rows where there is a **match in both tables** based on the join condition. Rows without a match are excluded.
* **When to Use:** When you only need rows with matching data in both tables.

**Query:**

SELECT employees.name, departments.dept\_name

FROM employees

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

**Result:**

|  |  |
| --- | --- |
| **name** | **dept\_name** |
| John | HR |
| Alice | Sales |
| Eve | IT |

**Explanation:**

* Only employees whose dept\_id exists in the departments table are included.
* Bob (with NULL dept\_id) and Finance (with no matching dept\_id in employees) are excluded.

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

* **Definition:** Retrieves all rows from the **left table (employees)** and the matching rows from the right table (departments). If no match is found, NULL is returned for columns from the right table.
* **When to Use:** When you need all records from the left table, regardless of matches in the right table.

**Query:**

SELECT employees.name, departments.dept\_name

FROM employees

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

**Result:**

|  |  |
| --- | --- |
| **name** | **dept\_name** |
| John | HR |
| Alice | Sales |
| Bob | NULL |
| Eve | IT |

**Explanation:**

* All employees are included.
* Bob has no matching dept\_id in departments, so dept\_name is NULL.

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

* **Definition:** Retrieves all rows from the **right table (departments)** and the matching rows from the left table (employees). If no match is found, NULL is returned for columns from the left table.
* **When to Use:** When you need all records from the right table, regardless of matches in the left table.

**Query:**

SELECT employees.name, departments.dept\_name

FROM employees

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

**Result:**

|  |  |
| --- | --- |
| **name** | **dept\_name** |
| John | HR |
| Alice | Sales |
| Eve | IT |
| NULL | Finance |

**Explanation:**

* All departments are included.
* Finance has no matching dept\_id in employees, so name is NULL.

**4. FULL OUTER JOIN**

* **Definition:** Combines the results of **LEFT JOIN** and **RIGHT JOIN**. Includes all rows from both tables, and fills NULL where there is no match.
* **When to Use:** When you need all records from both tables, regardless of matches.

**Query:**

SELECT employees.name, departments.dept\_name

FROM employees

FULL OUTER JOIN departments ON employees.dept\_id = departments.dept\_id;

**Result:**

|  |  |
| --- | --- |
| **name** | **dept\_name** |
| John | HR |
| Alice | Sales |
| Bob | NULL |
| Eve | IT |
| NULL | Finance |

**Explanation:**

* Includes all employees and all departments.
* Bob has no department (dept\_name is NULL).
* Finance has no employee (name is NULL).

**Summary of Joins**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type of Join** | **Left Table Rows** | **Right Table Rows** | **Matched Rows** | **Non-Matched Rows (NULL)** |
| **INNER JOIN** | Only matching | Only matching | Yes | No |
| **LEFT JOIN** | All | Only matching | Yes | Non-matches from right |
| **RIGHT JOIN** | Only matching | All | Yes | Non-matches from left |
| **FULL OUTER** | All | All | Yes | Non-matches from both |

**Visualization**

**INNER JOIN**

Employees Departments

+----------------+

John 10 <---> 10 HR

Alice 20 <---> 20 Sales

Eve 30 <---> 30 IT

**LEFT JOIN**

Employees Departments

+----------------+

John 10 <---> 10 HR

Alice 20 <---> 20 Sales

Eve 30 <---> 30 IT

Bob NULL ------> NULL

**RIGHT JOIN**

Employees Departments

+----------------+

John 10 <---> 10 HR

Alice 20 <---> 20 Sales

Eve 30 <---> 30 IT

NULL NULL ------> 40 Finance

**FULL OUTER JOIN**

Employees Departments

+----------------+

John 10 <---> 10 HR

Alice 20 <---> 20 Sales

Eve 30 <---> 30 IT

Bob NULL ------> NULL

NULL NULL ------> 40 Finance

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

**Definition:**

* **WHERE Clause**: Filters rows before any grouping or aggregation occurs in a SQL query.
* **HAVING Clause**: Filters rows after grouping or aggregation has been applied.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **WHERE** | **HAVING** |
| **Purpose** | Filters individual rows | Filters aggregated/grouped rows |
| **Execution Stage** | Before grouping/aggregation | After grouping/aggregation |
| **Can Use Aggregate Functions?** | No | Yes |

**Examples:**

1. **WHERE Clause Example**

SELECT name, salary

FROM employees

WHERE salary > 50000;

* **Explanation**: Filters employees whose salary is greater than 50,000.

1. **HAVING Clause Example**

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

HAVING AVG(salary) > 60000;

* **Explanation**:
  + First, the query calculates the average salary for each department (GROUP BY department).
  + Then, it filters departments with an average salary greater than 60,000 using the HAVING clause.

1. **Combined Example: WHERE and HAVING**

SELECT department, COUNT(\*) AS emp\_count

FROM employees

WHERE salary > 30000

GROUP BY department

HAVING COUNT(\*) > 5;

* **Explanation**:
  1. Filters employees with a salary greater than 30,000 (WHERE salary > 30000).
  2. Groups the remaining rows by department (GROUP BY department).
  3. Filters departments with more than 5 employees (HAVING COUNT(\*) > 5).

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

**Definition:**

* **GROUP BY**: Organizes rows into groups based on one or more columns.
* **HAVING**: Filters those groups based on conditions, often involving aggregate functions.

**Steps to Use GROUP BY and HAVING:**

1. Use GROUP BY to define how the rows should be grouped.
2. Apply aggregate functions (like COUNT(), SUM(), AVG(), etc.) on the groups.
3. Use HAVING to filter the groups based on the results of the aggregation.

**Example:**

**Problem**: Find departments with a total salary expense greater than 200,000.

**Table**: employees

|  |  |  |  |
| --- | --- | --- | --- |
| **emp\_id** | **name** | **department** | **salary** |
| 1 | John | HR | 50000 |
| 2 | Alice | IT | 90000 |
| 3 | Bob | IT | 60000 |
| 4 | Eve | HR | 60000 |
| 5 | Sara | Sales | 70000 |

**Query**:

SELECT department, SUM(salary) AS total\_salary

FROM employees

GROUP BY department

HAVING SUM(salary) > 200000;

**Explanation**:

1. Groups the rows by department.
2. Calculates the total salary for each department using SUM(salary).
3. Filters the groups where the total salary is greater than 200,000.

**Result**:

|  |  |
| --- | --- |
| **department** | **total\_salary** |
| IT | 150000 |
| HR | 110000 |

**Key Notes:**

* **GROUP BY** must come before HAVING.
* Use HAVING for aggregate filtering, not WHERE.

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

**Problem**: Find duplicate records in a table based on one or more columns.

**Example Table: employees**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **name** | **dept\_id** |
| 1 | John | 10 |
| 2 | Alice | 20 |
| 3 | John | 10 |
| 4 | Eve | 30 |

**Query:**

SELECT name, dept\_id, COUNT(\*) AS count

FROM employees

GROUP BY name, dept\_id

HAVING COUNT(\*) > 1;

**Explanation**:

1. GROUP BY name, dept\_id groups records by name and dept\_id.
2. COUNT(\*) counts the number of occurrences in each group.
3. HAVING COUNT(\*) > 1 filters groups with more than one occurrence.

**Result**:

|  |  |  |
| --- | --- | --- |
| **name** | **dept\_id** | **count** |
| John | 10 | 2 |

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

**Query to Retrieve Unique Values:**

Use the DISTINCT keyword to get unique values.

**Example Table: employees**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **name** | **dept\_id** |
| 1 | John | 10 |
| 2 | Alice | 20 |
| 3 | John | 10 |
| 4 | Eve | 30 |

**Query:**

SELECT DISTINCT name

FROM employees;

**Explanation**:

* DISTINCT name ensures that duplicate names are removed, showing only unique names.

**Result**:

|  |
| --- |
| **name** |
| John |
| Alice |
| Eve |

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

Aggregate functions perform calculations on groups of rows and return a single result.

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Description** | **Example Query** | **Example Result** |
| **COUNT()** | Counts rows in a column or result set. | SELECT COUNT(\*) FROM employees; | 4 (number of rows) |
| **SUM()** | Calculates the sum of values in a column. | SELECT SUM(salary) FROM employees; | 200000 (total salary) |
| **AVG()** | Finds the average value in a column. | SELECT AVG(salary) FROM employees; | 50000 (average salary) |
| **MIN()** | Finds the minimum value in a column. | SELECT MIN(salary) FROM employees; | 30000 (lowest salary) |
| **MAX()** | Finds the maximum value in a column. | SELECT MAX(salary) FROM employees; | 70000 (highest salary) |

**Example Table: employees**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **name** | **salary** |
| 1 | John | 50000 |
| 2 | Alice | 70000 |
| 3 | Bob | 30000 |

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

**Definition**:  
The DISTINCT keyword is used to remove duplicate rows from the result set, ensuring only unique rows are returned.

**Example Table: employees**

|  |  |
| --- | --- |
| **emp\_id** | **dept\_id** |
| 1 | 10 |
| 2 | 20 |
| 3 | 10 |

**Query:**

SELECT DISTINCT dept\_id

FROM employees;

**Explanation**:

* Removes duplicates in the dept\_id column, returning only unique department IDs.

**Result**:

|  |
| --- |
| **dept\_id** |
| 10 |
| 20 |

**Key Notes:**

* Use DISTINCT on one or more columns to ensure uniqueness.
* Combining DISTINCT with aggregate functions can summarize data more effectively.

𝐈𝐧𝐭𝐞𝐫𝐦𝐞𝐝𝐢𝐚𝐭𝐞 𝐒𝐐𝐋:

**1. Write a query to find the second-highest salary from an employee table**

**Problem**: Identify the second-highest salary in an employees table.

**Example Table: employees**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **name** | **salary** |
| 1 | John | 50000 |
| 2 | Alice | 70000 |
| 3 | Bob | 60000 |
| 4 | Eve | 90000 |

**Solution 1: Using a Subquery**

SELECT MAX(salary) AS second\_highest\_salary

FROM employees

WHERE salary < (SELECT MAX(salary) FROM employees);

**Explanation**:

1. (SELECT MAX(salary) FROM employees) finds the highest salary (90,000 in this case).
2. WHERE salary < (highest salary) filters out the highest salary.
3. MAX(salary) finds the maximum salary in the remaining rows, which is the second-highest (70,000).

**Result**:

| **second\_highest\_salary** |
| --- |
| 70000 |

**Solution 2: Using LIMIT or OFFSET (If Supported)**

SELECT salary

FROM employees

ORDER BY salary DESC

LIMIT 1 OFFSET 1;

**Explanation**:

1. ORDER BY salary DESC: Sorts salaries in descending order.
2. LIMIT 1 OFFSET 1: Skips the highest salary (offset = 1) and retrieves the next one.

**Result**:

|  |
| --- |
| **salary** |
| 70000 |

**2. What are subqueries, and how do you use them?**

**Definition**:  
A **subquery** is a query nested inside another query. It is used to fetch intermediate results needed for the main query.

**Types of Subqueries:**

1. **Single-row subquery**: Returns a single value.
2. **Multi-row subquery**: Returns multiple rows.
3. **Correlated subquery**: Depends on the outer query for its results.

**Example 1: Single-row Subquery**

**Problem**: Find employees earning more than the average salary.

**Query**:

SELECT name, salary

FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

**Explanation**:

1. (SELECT AVG(salary) FROM employees) calculates the average salary (e.g., 67500).
2. The outer query retrieves employees earning more than this value.

**Result**:

| **name** | **salary** |
| --- | --- |
| Eve | 90000 |
| Alice | 70000 |

**Example 2: Multi-row Subquery**

**Problem**: Find employees working in departments where the department size is more than 5.

**Query**:

SELECT name

FROM employees

WHERE dept\_id IN (

SELECT dept\_id

FROM departments

WHERE employee\_count > 5

);

**Explanation**:

1. The subquery retrieves dept\_id values where employee\_count > 5.
2. The outer query filters employees belonging to these departments.

**Example 3: Correlated Subquery**

**Problem**: Find employees whose salaries are greater than the average salary of their department.

**Query**:

SELECT name, salary

FROM employees e1

WHERE salary > (

SELECT AVG(salary)

FROM employees e2

WHERE e1.dept\_id = e2.dept\_id

);

**Explanation**:

1. For each employee in the outer query (e1), the subquery (e2) calculates the average salary in their department.
2. Filters employees whose salary exceeds this department average.

**Key Benefits of Subqueries:**

1. Simplifies complex queries by breaking them into smaller parts.
2. Allows dynamic filtering based on intermediate results.
3. Useful for solving problems that involve comparisons, conditions, and calculations.

**3. What is a Common Table Expression (CTE)? Give an example of when to use it.**

**Definition**:  
A **Common Table Expression (CTE)** is a temporary result set defined within the execution scope of a SELECT, INSERT, UPDATE, or DELETE statement. It is defined using the WITH keyword and can be referred to multiple times within the query.

**Key Benefits**:

* Improves query readability and organization.
* Enables recursion (useful for hierarchical data).
* Can be referenced multiple times within the main query.

**Example 1: Basic CTE**

**Problem**: Find the average salary per department, and then list the employees who earn above that average.

WITH DepartmentAvg AS (

SELECT dept\_id, AVG(salary) AS avg\_salary

FROM employees

GROUP BY dept\_id

)

SELECT e.name, e.salary, e.dept\_id

FROM employees e

JOIN DepartmentAvg da ON e.dept\_id = da.dept\_id

WHERE e.salary > da.avg\_salary;

**Expected Output**:

|  |  |  |
| --- | --- | --- |
| **name** | **salary** | **dept\_id** |
| Alice | 70000 | 20 |
| Eve | 90000 | 30 |

**Explanation**:

1. The CTE DepartmentAvg calculates the average salary for each department.
2. The main query then joins the employees table with the CTE to list employees earning more than their department's average salary.

**Result**:

|  |  |  |
| --- | --- | --- |
| **name** | **salary** | **dept\_id** |
| Alice | 70000 | 20 |

**Example 2: Recursive CTE (Hierarchical Data)**

**Problem**: Find all employees and their managers in a company (recursive query).

WITH RECURSIVE EmployeeHierarchy AS (

SELECT emp\_id, name, manager\_id

FROM employees

WHERE manager\_id IS NULL -- Start with top-level managers

UNION ALL

SELECT e.emp\_id, e.name, e.manager\_id

FROM employees e

JOIN EmployeeHierarchy eh ON e.manager\_id = eh.emp\_id

)

SELECT \* FROM EmployeeHierarchy;

**Expected Output**:

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **name** | **manager\_id** |
| 1 | Eve | NULL |
| 2 | Alice | 1 |
| 3 | John | 2 |
| 4 | Bob | 2 |

**Explanation**:

1. The CTE starts with employees who do not have a manager (manager\_id IS NULL).
2. It recursively joins with the same employees table to find each employee’s direct reportees.
3. This helps create a hierarchy of employees and managers.

**4. Explain window functions like ROW\_NUMBER(), RANK(), and DENSE\_RANK().**

**Definition**:  
**Window functions** perform calculations across a set of rows related to the current row, but unlike regular aggregate functions, they do not collapse the result set into a single row.

**Key Window Functions:**

1. **ROW\_NUMBER()**:
   * Assigns a unique sequential integer to rows within a partition of the result set, starting from 1.

**Example**:

SELECT name, salary, ROW\_NUMBER() OVER (ORDER BY salary DESC) AS row\_num

FROM employees;

**Explanation**:

* + This query assigns a unique row number to each employee ordered by salary in descending order.

**Result**:

|  |  |  |
| --- | --- | --- |
| **name** | **salary** | **row\_num** |
| Eve | 90000 | 1 |
| Alice | 70000 | 2 |
| John | 50000 | 3 |

1. **RANK()**:
   * Similar to ROW\_NUMBER(), but it assigns the same rank to rows with equal values. It leaves gaps in the ranking when there are ties.

**Example**:

SELECT name, salary, RANK() OVER (ORDER BY salary DESC) AS rank

FROM employees;

**Explanation**:

* + Employees with the same salary will have the same rank, but the next rank will be skipped.

**Result**:

|  |  |  |
| --- | --- | --- |
| **name** | **salary** | **rank** |
| Eve | 90000 | 1 |
| Alice | 70000 | 2 |
| John | 50000 | 3 |

1. **DENSE\_RANK()**:
   * Similar to RANK(), but does not leave gaps in the ranking when there are ties.

**Example**:

SELECT name, salary, DENSE\_RANK() OVER (ORDER BY salary DESC) AS dense\_rank

FROM employees;

**Explanation**:

* + Employees with the same salary will have the same rank, but the next rank will not be skipped.

**Result**:

|  |  |  |
| --- | --- | --- |
| **name** | **salary** | **dense\_rank** |
| Eve | 90000 | 1 |
| Alice | 70000 | 2 |
| John | 50000 | 3 |

**5. How do you combine results of two queries using UNION and UNION ALL?**

**Definition**:

* **UNION** and **UNION ALL** are used to combine results from two or more SELECT queries into a single result set.
* **UNION** removes duplicates, while **UNION ALL** includes all rows, including duplicates.

**1. UNION (removes duplicates)**

**Example**:

SELECT name FROM employees WHERE dept\_id = 10

UNION

SELECT name FROM employees WHERE dept\_id = 20;

**Explanation**:

* This query combines the names of employees from department 10 and 20, but duplicates (same name in both departments) will be removed.

**Result**:

|  |
| --- |
| **name** |
| John |
| Alice |
| Bob |

**2. UNION ALL (keeps duplicates)**

**Example**:

SELECT name FROM employees WHERE dept\_id = 10

UNION ALL

SELECT name FROM employees WHERE dept\_id = 20;

**Explanation**:

* This query also combines the names of employees from department 10 and 20, but it keeps duplicate names.

**Result**:

|  |
| --- |
| **name** |
| John |
| Alice |
| John |
| Bob |

**Key Differences:**

* **UNION**: Removes duplicates from the result set.
* **UNION ALL**: Includes all duplicates, providing a faster operation (because it does not perform duplicate elimination).

**6. What are indexes in SQL, and how do they improve query performance?**

**Definition**: An **index** in SQL is a database object that improves the speed of data retrieval operations on a table at the cost of additional space and time for updates, insertions, and deletions. An index is created on one or more columns of a table to make searching and querying more efficient.

**How Indexes Improve Query Performance**:

* **Faster Search**: Indexes help in faster data retrieval because the database engine can search through an index (which is often structured as a B-tree) rather than scanning the entire table.
* **Reduced Data Access Time**: By storing pointers to the actual data in the table, an index reduces the amount of data that needs to be read.
* **Efficient Sorting and Filtering**: Indexes are particularly useful for ORDER BY, GROUP BY, and WHERE clauses, as the database engine can leverage the index to perform these operations faster.

**Example of creating an index:**

CREATE INDEX idx\_employee\_name ON employees(name);

**Explanation**:

* This creates an index idx\_employee\_name on the name column of the employees table. When you query the name column, the database will use this index to find matching rows faster.

**Impact on Query Performance:**

* **Without an Index**:

SELECT \* FROM employees WHERE name = 'John';

The database would need to scan the entire employees table to find rows where name = 'John'.

* **With an Index**:

SELECT \* FROM employees WHERE name = 'John';

With the index, the database can quickly locate the rows by looking up the value in the index, avoiding a full table scan.

**Expected Output (Query Result)**:

|  |  |  |  |
| --- | --- | --- | --- |
| **emp\_id** | **name** | **dept\_id** | **salary** |
| 3 | John | 10 | 50000 |

**7. Write a query to calculate the total sales for each month using GROUP BY.**

**Problem**: You have a sales table with columns sale\_date and sale\_amount, and you want to calculate the total sales for each month.

**Query:**

SELECT

YEAR(sale\_date) AS year,

MONTH(sale\_date) AS month,

SUM(sale\_amount) AS total\_sales

FROM sales

GROUP BY YEAR(sale\_date), MONTH(sale\_date)

ORDER BY year, month;

**Explanation**:

* YEAR(sale\_date) extracts the year from the sale\_date.
* MONTH(sale\_date) extracts the month from the sale\_date.
* SUM(sale\_amount) calculates the total sales amount for each month.
* GROUP BY groups the results by year and month.
* ORDER BY year, month sorts the results by year and month in ascending order.

**Expected Output:**

|  |  |  |
| --- | --- | --- |
| **year** | **month** | **total\_sales** |
| 2023 | 1 | 50000 |
| 2023 | 2 | 62000 |
| 2023 | 3 | 48000 |
| 2023 | 4 | 55000 |

**Explanation of Output**:

* For January 2023, the total sales is 50,000.
* For February 2023, the total sales is 62,000.
* And so on for each month.

**Key Points:**

* **GROUP BY**: This clause groups the rows in the table that have the same values for the specified columns (in this case, year and month).
* **SUM()**: This is an aggregate function that adds up all the sale\_amount values for each group (each month).
* **Ordering**: ORDER BY is used to sort the result by year and month, ensuring that the data is presented chronologically.

### ****1. How do you optimize a slow-running SQL query?****

Optimizing SQL queries is crucial for improving performance, especially when working with large datasets. Below are several techniques to optimize slow-running SQL queries:

#### ****1.1. Use Indexes****

* **Indexes** speed up data retrieval by allowing the database to quickly locate rows matching a condition. You should create indexes on columns frequently used in WHERE, JOIN, ORDER BY, or GROUP BY clauses.
* **Example**: If you frequently query the employees table by dept\_id, create an index on the dept\_id column:

CREATE INDEX idx\_dept\_id ON employees(dept\_id);

#### \*\*1.2. Avoid SELECT \*\*\*

* Instead of using SELECT \*, specify only the columns you need. This reduces the amount of data being retrieved and sent over the network.
* **Example**:

SELECT name, salary FROM employees WHERE dept\_id = 10;

**Output**:

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **dept\_id** | **salary** |
| 101 | Alice | 10 | 50000 |
| 102 | Bob | 10 | 45000 |
| 103 | Charlie | 10 | 55000 |

#### ****1.3. Use LIMIT/OFFSET****

* If you only need a subset of data, use LIMIT (or TOP in SQL Server) to reduce the number of rows returned. This is especially useful in pagination scenarios.
* **Example**:

SELECT \* FROM employees LIMIT 100; -- Retrieve only the first 100 rows

#### ****1.4. Optimize JOINs****

* **Minimize the number of joins**: Try to limit the number of JOIN operations in your query.
* **Use appropriate join types**: For example, use INNER JOIN instead of OUTER JOIN unless necessary.
* **Join on indexed columns**: If you're joining large tables, make sure the columns used in the JOIN condition are indexed.

#### ****1.5. Use WHERE Clause Efficiently****

* **Filter as early as possible**: Always filter data early with WHERE clauses to minimize the dataset before applying other operations like GROUP BY or ORDER BY.
* **Avoid functions on columns in WHERE**: Using functions on columns (e.g., WHERE YEAR(date\_column) = 2020) disables index usage. It's better to transform the column before querying or use a range condition.

#### ****1.6. Avoid Using Subqueries in SELECT****

* Subqueries in the SELECT clause can be slow. Consider using JOINs or WITH statements (Common Table Expressions) instead.
* **Example**: Instead of:

SELECT name, (SELECT MAX(salary) FROM employees WHERE dept\_id = e.dept\_id) AS max\_salary

FROM employees e;

Use:

SELECT e.name, MAX(e.salary) AS max\_salary

FROM employees e

GROUP BY e.dept\_id, e.name;

#### ****1.7. Use Proper Data Types****

* Ensure that your columns use appropriate data types to minimize storage and optimize query performance. For example, using INT for small numbers and DATE for date-related columns instead of VARCHAR.

#### ****1.8. Avoid Using Wildcards in LIKE Clauses****

* Using LIKE with a wildcard at the beginning (e.g., LIKE '%abc') can result in a full table scan. Try to avoid leading wildcards or use them only when necessary.

#### ****1.9. Avoid Complex Calculations in Queries****

* Avoid performing calculations inside the query when possible, especially on large datasets. For instance, calculating sums, averages, or other aggregates on the fly can be slow.

### ****2. What are views in SQL, and when would you use them?****

**Definition**: A **view** in SQL is a virtual table based on the result of a query. A view does not store data itself but provides a way to present data in a specific format by encapsulating a complex query.

#### ****2.1. Benefits of Using Views****:

* **Simplify Complex Queries**: Views simplify complex queries by encapsulating them into a single virtual table. This allows you to reuse the same logic without rewriting the query each time.
* **Data Security**: Views can be used to provide users access to only a specific subset of data without giving them direct access to the base tables.
* **Reusability**: Once a view is created, you can use it in multiple queries like a normal table.

#### ****2.2. Creating a View****:

You can create a view using the CREATE VIEW statement.

* **Syntax**:

CREATE VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

#### ****2.3. Example of Creating and Using a View****:

**Problem**: Suppose you frequently need to query employees' names and their respective department names. Instead of writing the query every time, you can create a view.

CREATE VIEW employee\_dept\_view AS

SELECT e.name, d.dept\_name

FROM employees e

JOIN departments d ON e.dept\_id = d.dept\_id;

Now, instead of writing the entire JOIN query every time, you can query the view:

SELECT \* FROM employee\_dept\_view;

**Expected Output**:

|  |  |
| --- | --- |
| **name** | **dept\_name** |
| Alice | HR |
| Bob | IT |
| John | Finance |

#### ****2.4. Modifying a View****:

* If you need to modify a view, use the CREATE OR REPLACE statement:

**SQL Query to Create/Replace the View:**

CREATE OR REPLACE VIEW employee\_dept\_view AS

SELECT e.name, d.dept\_name, e.salary

FROM employees e

JOIN departments d ON e.dept\_id = d.dept\_id;

**Step 1: What the Query Does**

This query will:

1. **Create or replace** a view named employee\_dept\_view.
2. The view will contain the following columns:
   * e.name: The name of the employee from the employees table.
   * d.dept\_name: The department name from the departments table.
   * e.salary: The employee's salary from the employees table.
3. The view is based on an **INNER JOIN** between the employees and departments tables, where the dept\_id column from the employees table matches the dept\_id column in the departments table.

**Step 2: Example Data in employees and departments Tables**

Let's assume we have the following data:

**employees Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **dept\_id** | **salary** |
| 101 | Alice | 10 | 50000 |
| 102 | Bob | 10 | 45000 |
| 103 | John | 20 | 55000 |
| 104 | Sarah | 30 | 60000 |

**departments Table:**

|  |  |
| --- | --- |
| **dept\_id** | **dept\_name** |
| 10 | HR |
| 20 | IT |
| 30 | Marketing |

**Step 3: Querying the View**

Once the view employee\_dept\_view is created or replaced, querying it will give us the following result.

**SQL Query to View Data**:

SELECT \* FROM employee\_dept\_view;

**Expected Output:**

|  |  |  |
| --- | --- | --- |
| **name** | **dept\_name** | **salary** |
| Alice | HR | 50000 |
| Bob | HR | 45000 |
| John | IT | 55000 |
| Sarah | Marketing | 60000 |

#### ****2.5. Updating Data Through a View****:

* **Updatable Views**: Some views are updatable, meaning you can use them to INSERT, UPDATE, or DELETE rows in the underlying table.
* However, views that involve JOINs, GROUP BY, DISTINCT, or aggregation are typically not updatable.

**Example**: If your view is simple and only selects data from one table, it can be updatable:

CREATE VIEW simple\_employee\_view AS

SELECT name, salary FROM employees;

his creates a view named simple\_employee\_view that selects only the name and salary columns from the employees table.

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **dept\_id** | **salary** |
| 101 | Alice | 10 | 50000 |
| 102 | Bob | 10 | 45000 |
| 103 | John | 20 | 50000 |
| 104 | Sarah | 30 | 60000 |

Now you can update the data in the underlying table via the view:

UPDATE simple\_employee\_view SET salary = 55000 WHERE name = 'John';

**Output**:

|  |  |
| --- | --- |
| **name** | **salary** |
| Alice | 50000 |
| Bob | 45000 |
| John | 50000 |
| Sarah | 60000 |

### ****Key Takeaways****:

* **Optimizing Queries**: Use indexes, avoid unnecessary computations, and properly structure your queries to improve performance.
* **Views**: These are virtual tables that simplify complex queries, provide security, and allow for data abstraction. Views help you present data in a simplified and reusable manner.

**3. What is the difference between a stored procedure and a function in SQL?**

A **stored procedure** and a **function** are both used to encapsulate logic in SQL, but they have distinct characteristics and purposes. Let's break down their differences in detail:

**Stored Procedure:**

* **Definition**: A stored procedure is a precompiled collection of one or more SQL statements that can be executed by a user or an application. It allows you to group a set of SQL queries together for easy reuse.
* **Characteristics**:
  + Stored procedures do not return a value directly (though they can return multiple results via OUT parameters or by performing tasks like modifying data).
  + They can perform actions like modifying the database (e.g., inserting, updating, deleting records).
  + You can call a stored procedure without expecting any value to be returned.
  + Stored procedures can have input, output, or both types of parameters.
  + They are generally used for operations that involve multiple SQL statements and logic.

**Function:**

* **Definition**: A function is a piece of code that returns a single value and is typically used to compute and return a result (e.g., a calculation, a transformation, or an aggregation).
* **Characteristics**:
  + A function must return a single value (it can return scalar values, such as integers, strings, dates, etc.).
  + Functions cannot modify the database (e.g., you can't insert or delete data from within a function).
  + Functions can only have input parameters (no output parameters).
  + They are generally used for computations and returning values to the caller.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Stored Procedure** | **Function** |
| **Return Value** | Can return multiple result sets or no value. | Must return exactly one value. |
| **Database Modification** | Can modify the database (INSERT, UPDATE, DELETE). | Cannot modify the database (no data modification). |
| **Use Case** | Used for performing operations like updates, inserts, or deletions. | Used for computations, retrieving calculated values. |
| **Parameters** | Can have input, output, or both types of parameters. | Can only have input parameters. |
| **Invocation** | Can be called with the CALL statement. | Can be called within a SQL expression. |

**Example:**

1. **Stored Procedure**:

CREATE PROCEDURE UpdateSalary(IN emp\_id INT, IN new\_salary DECIMAL)

BEGIN

UPDATE employees

SET salary = new\_salary

WHERE employee\_id = emp\_id;

END;

You would call the stored procedure like this:

CALL UpdateSalary(101, 60000);

1. **Function**:

CREATE FUNCTION CalculateSalaryIncrease(old\_salary DECIMAL)

RETURNS DECIMAL

BEGIN

RETURN old\_salary \* 0.10;

END;

You would call the function like this:

SELECT CalculateSalaryIncrease(50000);

**Expected Output for the function**:

5000

**4. Explain the difference between TRUNCATE, DELETE, and DROP commands.**

The TRUNCATE, DELETE, and DROP commands are all used to remove data or objects in SQL, but they operate in very different ways and serve different purposes. Here’s a detailed explanation of each:

**1. DELETE Command:**

* **Purpose**: The DELETE command is used to remove rows from a table based on a condition specified in the WHERE clause.
* **Characteristics**:
  + Deletes records one by one.
  + It is **slower** because it logs each row deletion.
  + You can **use a WHERE clause** to delete specific rows (e.g., DELETE FROM employees WHERE salary < 50000).
  + It **does not reset the identity** (auto-increment column).
  + Can be rolled back if wrapped in a transaction.

**Example:**

sql

Copy code

DELETE FROM employees WHERE salary < 50000;

**Expected Output**: Deletes all employees with a salary less than 50000.

**2. TRUNCATE Command:**

* **Purpose**: The TRUNCATE command removes all rows from a table quickly, but unlike DELETE, it does not log individual row deletions.
* **Characteristics**:
  + Removes all rows from a table without any condition (no WHERE clause).
  + It is **faster** than DELETE because it doesn't generate individual row deletion logs.
  + Cannot be rolled back in most databases (depends on the DBMS and whether a transaction is used).
  + **Resets the identity** (auto-increment value) to its initial value.
  + It **cannot be used** with a WHERE clause.

**Example:**

TRUNCATE TABLE employees;

**Expected Output**: All rows in the employees table are removed, and the identity (if any) is reset.

**3. DROP Command:**

* **Purpose**: The DROP command is used to completely remove a table, database, or other database objects like views or indexes.
* **Characteristics**:
  + Completely removes the table or object, including its structure and data.
  + Once dropped, the table cannot be rolled back.
  + **Faster** than both DELETE and TRUNCATE because it removes the entire structure and data.
  + **Cannot be undone**, so be very careful when using it.

**Example:**

DROP TABLE employees;

**Expected Output**: The employees table is completely removed, including all data, structure, and any constraints related to it.

**Key Differences:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Command** | **Purpose** | **Data Affected** | **Can Rollback** | **Resets Identity** | **Conditions for Deletion** |
| **DELETE** | Removes specific rows from a table | Removes specific rows | Yes | No | Can use WHERE clause |
| **TRUNCATE** | Removes all rows from a table | Removes all rows | No | Yes | No WHERE clause |
| **DROP** | Removes a table or database object | Removes the entire table | No | N/A | No WHERE clause |

**5. What are Windowing Functions, and How Are They Used in Analytics?**

**Windowing functions** (also known as **window functions**) are special types of functions in SQL that perform calculations across a set of rows that are somehow related to the current row. These functions allow you to calculate values over a window (or range) of rows, instead of just on a single row.

In analytics, window functions are useful for performing calculations like running totals, moving averages, rank calculations, and more—without needing to group rows together as you would with GROUP BY.

**Key Points About Window Functions:**

* They operate on a set of rows related to the current row, which is defined by an **OVER()** clause.
* Unlike aggregate functions (e.g., COUNT(), SUM()), window functions **do not collapse rows** but instead provide a result for each row, while considering the related rows.
* Common window functions include ROW\_NUMBER(), RANK(), DENSE\_RANK(), NTILE(), and aggregate functions like SUM() or AVG() when used in a window context.

**Example of a Window Function:**

Consider a table sales with columns: sales\_id, employee\_id, sale\_amount, and sale\_date.

CREATE TABLE sales (

sales\_id INT,

employee\_id INT,

sale\_amount DECIMAL,

sale\_date DATE

);

A simple query using a **window function** might look like this:

SELECT employee\_id,

sale\_amount,

SUM(sale\_amount) OVER (PARTITION BY employee\_id ORDER BY sale\_date) AS running\_total

FROM sales;

This query calculates a **running total** of the sale\_amount for each employee, ordered by sale\_date.

**Expected Output:**

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **sale\_amount** | **running\_total** |
| 1 | 100 | 100 |
| 1 | 200 | 300 |
| 2 | 150 | 150 |
| 2 | 250 | 400 |

Here, running\_total gives a cumulative sum for each employee\_id, calculated in the order of sale\_date.

**6. How Do You Use PARTITION BY and ORDER BY in Window Functions?**

The **PARTITION BY** and **ORDER BY** clauses are used in window functions to define the **window** of rows over which the function operates.

* **PARTITION BY**: This divides the result set into partitions (groups of rows), and the window function is computed separately for each partition.
* **ORDER BY**: This defines the order in which the rows are processed within each partition.

**Example:**

Consider the same sales table, and you want to calculate the **rank** of each employee based on the sale\_amount in descending order.

SELECT employee\_id,

sale\_amount,

RANK() OVER (PARTITION BY employee\_id ORDER BY sale\_amount DESC) AS sale\_rank

FROM sales;

**Explanation:**

* **PARTITION BY employee\_id**: This ensures that the ranking is done for each employee separately.
* **ORDER BY sale\_amount DESC**: This ranks the sales in descending order, so the highest sale gets rank 1.

**Expected Output:**

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **sale\_amount** | **sale\_rank** |
| 1 | 100 | 2 |
| 1 | 200 | 1 |
| 2 | 150 | 2 |
| 2 | 250 | 1 |

In this example:

* For each employee, the sales are ranked in descending order of sale\_amount.

**7. How Do You Handle NULL Values in SQL, and What Functions Help with That (e.g., COALESCE, ISNULL)?**

In SQL, NULL represents a missing or unknown value. When dealing with NULL values, you may need to handle them explicitly in your queries, as many operations (such as comparisons or aggregations) don't behave as expected with NULL. SQL provides several functions to handle NULL values.

**COALESCE Function:**

The COALESCE function returns the first non-NULL value in a list of arguments. It is commonly used to replace NULL values with a default value.

**Syntax**:

COALESCE(expression\_1, expression\_2, ..., expression\_n)

* It evaluates the expressions from left to right and returns the first non-NULL value.
* If all arguments are NULL, it returns NULL.

**Example:**

SELECT employee\_id,

COALESCE(sale\_amount, 0) AS sale\_amount

FROM sales;

**Explanation**: This query replaces any NULL values in the sale\_amount column with 0.

**Expected Output:**

|  |  |
| --- | --- |
| **employee\_id** | **sale\_amount** |
| 1 | 100 |
| 2 | 0 |
| 3 | 150 |

**ISNULL Function (SQL Server Specific):**

The ISNULL function is similar to COALESCE, but it takes exactly two arguments and returns the first argument if it is not NULL; otherwise, it returns the second argument.

**Syntax**:

ISNULL(expression, replacement\_value)

**Example:**

SELECT employee\_id,

ISNULL(sale\_amount, 0) AS sale\_amount

FROM sales;

**Explanation**: If sale\_amount is NULL, it is replaced with 0.

**Expected Output:**

|  |  |
| --- | --- |
| **employee\_id** | **sale\_amount** |
| 1 | 100 |
| 2 | 0 |
| 3 | 150 |

**Other Functions to Handle NULL:**

* **IFNULL** (MySQL Specific): Similar to COALESCE but only takes two arguments.
* **NULLIF**: Compares two expressions and returns NULL if they are equal, otherwise returns the first expression.

**Example**:

SELECT employee\_id,

NULLIF(sale\_amount, 0) AS sale\_amount

FROM sales;

This will return NULL if sale\_amount is 0.