**🔍 SQL Differences – Comprehensive Question List**

**🔹 Data Definition & Manipulation**

* **What is the difference between DELETE and TRUNCATE?**

| **Aspect** | **DELETE** | **TRUNCATE** |
| --- | --- | --- |
| **Purpose** | Removes specific rows from a table | Removes all rows from a table |
| **Filtering** | Supports WHERE clause to target specific rows | Does not support WHERE; affects the entire table |
| **Rollback Capability** | Can be rolled back if used within a transaction | Usually cannot be rolled back (depends on the database system) |
| **Trigger Activation** | Activates DELETE triggers | Does not activate triggers |
| **Logging** | Fully logged (row-by-row deletion) | Minimally logged (faster for large datasets) |
| **Identity Column Reset** | Does not reset identity column | Resets identity column (e.g., auto-increment counter) |
| **Performance** | Slower for large datasets due to row-level operations | Faster for bulk deletion |
| **Command Type** | DML (Data Manipulation Language) | DDL (Data Definition Language) |

**🧠 Example Usage**

-- Deletes only HR department employees

DELETE FROM employees WHERE department = 'HR';

-- Removes all rows from the employees table

TRUNCATE TABLE employees;

**✅ Summary**

* Use DELETE when you need to remove specific rows and possibly roll back the operation.
* Use TRUNCATE when you want to quickly and completely clear a table.

----------------------------------------------------------------------------------------------------

* **What distinguishes a TEMPORARY table from a PERMANENT table?**

| * **Aspect** | * **TEMPORARY Table** | * **PERMANENT Table** |
| --- | --- | --- |
| * **Definition** | * Exists temporarily during a session or transaction | * Exists permanently in the database until explicitly dropped |
| * **Scope** | * Accessible only within the session or procedure that created it | * Accessible by any user with appropriate permissions |
| * **Persistence** | * Automatically deleted when the session ends | * Remains in the database until manually deleted |
| * **Storage Location** | * Stored in temporary database or memory | * Stored in the main database on disk |
| * **Use Case** | * Ideal for intermediate results, staging, or session-specific logic | * Used for long-term data storage and persistent business logic |
| * **Naming** | * Often prefixed with # or ## in SQL Server | * Regular table names without special prefixes |
| * **Performance** | * May offer faster access due to in-memory storage | * Optimized for durability and indexing |
| * **DDL Operations** | * Limited support for constraints and indexes | * Full support for constraints, indexes, and relationships |

* **✅ Summary**
* Use **TEMPORARY tables** for short-lived tasks like staging, transformations, or session-specific operations.
* Use **PERMANENT tables** for storing business-critical data that must persist across sessions and users.

---------------------------------------------------------------------------------------------------

* **How does a VIEW differ from a TABLE?**

| * **Aspect** | * **VIEW** | * **TABLE** |
| --- | --- | --- |
| * **Definition** | * A virtual table based on the result of a SQL query | * A physical structure that stores data in rows and columns |
| * **Storage** | * Does not store data physically; only stores the query logic | * Stores actual data on disk |
| * **Data Retrieval** | * Retrieves data dynamically from underlying tables | * Retrieves data directly from its own stored records |
| * **Performance** | * May be slower due to query execution at runtime | * Generally faster for repeated access |
| * **Updatability** | * Can be updatable (with restrictions); depends on DBMS and query structure | * Fully updatable |
| * **Usage** | * Used for abstraction, simplification, and security | * Used for persistent data storage |
| * **Dependency** | * Depends on underlying tables; changes in base tables affect the view | * Independent structure; changes affect only the table itself |
| * **Security** | * Can restrict access to specific columns or rows | * Requires explicit permission control |
| * **DDL Operations** | * Limited support (e.g., cannot add indexes directly) | * Full support for indexes, constraints, and relationships |

* **✅ Summary**
* Use a **VIEW** when you want to simplify complex queries, restrict access, or create reusable logic without duplicating data.
* Use a **TABLE** when you need to store and manage persistent data with full control over indexing, constraints, and relationships.

-----------------------------------------------------------------------------------------------------

**🔹 Keys & Constraints**

* **What is the difference between a PRIMARY KEY and a UNIQUE constraint?**

| **Aspect** | **PRIMARY KEY** | **UNIQUE Constraint** |
| --- | --- | --- |
| **Purpose** | Uniquely identifies each row in a table | Ensures all values in a column or group of columns are unique |
| **Null Handling** | Does **not** allow NULL values | Allows **multiple NULLs** (depending on the database system) |
| **Number per Table** | Only **one** PRIMARY KEY allowed per table | Can have **multiple** UNIQUE constraints per table |
| **Default Index** | Automatically creates a **clustered index** (in most DBMS) | Creates a **non-clustered index** (in most DBMS) |
| **Usage in Relationships** | Commonly used in **foreign key** references | Not typically used for foreign key relationships |
| **Column Requirement** | Must be defined on **non-nullable** column(s) | Can be defined on **nullable** column(s) |
| **Constraint Type** | Combination of **NOT NULL + UNIQUE** | Only enforces **uniqueness**, not nullability |

**✅ Summary**

* Use a **PRIMARY KEY** when you need a strict, unique identifier for each row—no duplicates, no nulls.
* Use a **UNIQUE constraint** when you want to enforce uniqueness but allow flexibility with null values or multiple unique columns.

----------------------------------------------------------------------------------------------------

* **What is the difference between a PRIMARY KEY and a FOREIGN KEY?**

| * **Aspect** | * **PRIMARY KEY** | * **FOREIGN KEY** |
| --- | --- | --- |
| * **Purpose** | * Uniquely identifies each row in a table | * Establishes a link between two tables |
| * **Data Ownership** | * Defined in the parent table | * Defined in the child table |
| * **Uniqueness** | * Must be unique and not null | * Can contain duplicate values |
| * **Null Handling** | * Does **not** allow NULL values | * May allow NULL values (depends on design) |
| * **Number per Table** | * Only **one** PRIMARY KEY allowed per table | * Can have **multiple** FOREIGN KEYs per table |
| * **Referential Integrity** | * Ensures uniqueness within the table | * Ensures consistency across related tables |
| * **Index Creation** | * Automatically creates an index | * May or may not create an index (depends on DBMS) |
| * **Relationship Role** | * Acts as the **source** of reference | * Acts as the **target** referencing the PRIMARY KEY |

* **✅ Summary**
* Use a **PRIMARY KEY** to uniquely identify records within a table.
* Use a **FOREIGN KEY** to enforce relationships between tables and maintain referential integrity.

----------------------------------------------------------------------------------------------------

* **How does a CHECK constraint differ from a DEFAULT constraint?**

| * **Aspect** | * **CHECK Constraint** | * **DEFAULT Constraint** |
| --- | --- | --- |
| * **Purpose** | * Validates that column values meet a specified condition | * Assigns a default value to a column when no value is provided |
| * **Execution Time** | * Evaluated during INSERT and UPDATE operations | * Applied only during INSERT if no explicit value is given |
| * **Condition Type** | * Can enforce complex logical expressions (e.g., salary > 0) | * Provides a static or function-based default (e.g., GETDATE()) |
| * **Error Handling** | * Raises an error if the condition is violated | * Automatically fills in the default value without raising an error |
| * **Flexibility** | * Can reference other columns in the same row | * Cannot reference other columns; must be a constant or function |
| * **Use Case** | * Ensures data integrity by enforcing business rules | * Simplifies data entry and ensures consistency for missing values |

* **🧠 Example**
* -- CHECK constraint
* CREATE TABLE employees (
* salary INT CHECK (salary > 0)
* );
* -- DEFAULT constraint
* CREATE TABLE orders (
* order\_date DATE DEFAULT GETDATE()
* );
* **✅ Summary**
* Use a **CHECK constraint** to enforce rules on data values.
* Use a **DEFAULT constraint** to automatically populate missing values with predefined defaults.

--------------------------------------------------------------------------------------------

**🔹 Joins & Subqueries**

* How does an INNER JOIN differ from a LEFT JOIN?

**🔗 Difference Between INNER JOIN and LEFT JOIN in SQL**

| **Aspect** | **INNER JOIN** | **LEFT JOIN** |
| --- | --- | --- |
| **Definition** | Returns only matching rows from both tables | Returns all rows from the left table and matching rows from the right table |
| **Matching Requirement** | Requires matching values in both tables | Includes unmatched rows from the left table with NULL values from the right |
| **Result Set** | Contains only rows where the join condition is satisfied | Contains all rows from the left table, regardless of matches |
| **Use Case** | When you need only intersecting data | When you need all data from the left table, even if there's no match |
| **Performance** | Typically faster due to smaller result set | May be slower due to larger result set and NULL handling |
| **Nulls in Output** | No NULL values from unmatched rows | NULL values appear for columns from the right table when no match exists |

**🧠 Example**

-- INNER JOIN: returns only employees with matching departments

SELECT e.name, d.department\_name

FROM employees e

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

-- LEFT JOIN: returns all employees, even if they don't belong to a department

SELECT e.name, d.department\_name

FROM employees e

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

**✅ Summary**

* Use **INNER JOIN** when you want only matched records from both tables.
* Use **LEFT JOIN** when you want all records from the left table, including those without matches in the right table.

-----------------------------------------------------------------------------------------------------

**How does a CROSS JOIN differ from an INNER JOIN?**

| **Aspect** | **CROSS JOIN** | **INNER JOIN** |
| --- | --- | --- |
| **Definition** | Returns the Cartesian product of two tables (all possible row combinations) | Returns only rows with matching values based on a specified condition |
| **Join Condition** | Does **not** require a join condition | Requires a join condition using ON or USING clause |
| **Result Size** | Number of rows = rows in Table A × rows in Table B | Number of rows = only those that satisfy the join condition |
| **Use Case** | Used when all combinations are needed (e.g., pairing every item with every option) | Used when you want to match related records between tables |
| **Null Handling** | Includes all combinations regardless of nulls | Excludes rows where join condition fails due to nulls or mismatches |
| **Performance** | Can be expensive for large tables due to exponential row growth | More efficient for relational queries with defined conditions |

**🧠 Example**

-- CROSS JOIN: returns all combinations of employees and departments

SELECT e.name, d.department\_name

FROM employees e

CROSS JOIN departments d;

-- INNER JOIN: returns only employees with matching departments

SELECT e.name, d.department\_name

FROM employees e

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

**✅ Summary**

* Use **CROSS JOIN** when you need every possible combination of rows from two tables.
* Use **INNER JOIN** when you want only the rows that match based on a condition.

-----------------------------------------------------------------------------------------------------

**What is the difference between a subquery and a JOIN?**

**🔍 Difference Between Subquery and JOIN in SQL**

| **Aspect** | **Subquery** | **JOIN** |
| --- | --- | --- |
| **Definition** | A query nested inside another query | Combines rows from two or more tables based on a related column |
| **Execution Style** | Executed independently, then its result is used by the outer query | Executes by matching rows across tables simultaneously |
| **Readability** | Can be harder to read in complex scenarios | Often more readable and efficient for multi-table relationships |
| **Performance** | May be slower, especially if not optimized | Generally faster due to direct row matching and indexing |
| **Use Case** | Ideal for filtering, aggregation, or conditional logic | Ideal for retrieving related data across multiple tables |
| **Types** | Correlated and non-correlated subqueries | INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN |
| **Reusability** | Cannot be reused within the same query | Joined tables can be reused in multiple clauses |
| **Complexity** | Better for single-table logic or nested filtering | Better for relational data and combining multiple datasets |

**🧠 Example**

-- Subquery: Get employees in departments with more than 5 people

SELECT name FROM employees

WHERE department\_id IN (

SELECT department\_id FROM employees

GROUP BY department\_id

HAVING COUNT(\*) > 5

);

-- JOIN: Get employee names with their department names

SELECT e.name, d.department\_name

FROM employees e

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

**✅ Summary**

* Use a **subquery** when you need to filter or compute values before passing them to the outer query.
* Use a **JOIN** when you want to combine data from multiple tables based on relationships.

--------------------------------------------------------------------------------------------

**What is the difference between a correlated subquery and a non-correlated subquery?**

**🔍 Difference Between Correlated and Non-Correlated Subqueries in SQL**

| **Aspect** | **Correlated Subquery** | **Non-Correlated Subquery** |
| --- | --- | --- |
| **Definition** | A subquery that references columns from the outer query | A subquery that runs independently of the outer query |
| **Execution Style** | Executed **once per row** of the outer query | Executed **once** and its result is passed to the outer query |
| **Dependency** | Depends on values from the outer query | Self-contained; does not rely on outer query values |
| **Performance** | May be slower due to repeated execution | Typically faster due to single execution |
| **Use Case** | Used when filtering or comparing row-by-row | Used for static filtering, aggregation, or existence checks |
| **Complexity** | More complex and harder to optimize | Simpler and easier to understand |
| **Example Scenario** | Comparing each employee’s salary to the average salary of their department | Filtering employees based on a fixed list of department IDs |

**🧠 Example**

-- Correlated Subquery: compares each employee's salary to their department's average

SELECT name

FROM employees e

WHERE salary > (

SELECT AVG(salary)

FROM employees

WHERE department\_id = e.department\_id

);

-- Non-Correlated Subquery: filters employees in departments with more than 5 people

SELECT name

FROM employees

WHERE department\_id IN (

SELECT department\_id

FROM employees

GROUP BY department\_id

HAVING COUNT(\*) > 5

);

**✅ Summary**

* Use a **correlated subquery** when the subquery needs data from the outer query to evaluate each row.
* Use a **non-correlated subquery** when the subquery can run independently and return a static result.

-----------------------------------------------------------------------------------------------------

* How does a CTE differ from a subquery?

**🧮 Difference Between CTE and Subquery in SQL**

| **Aspect** | **CTE (Common Table Expression)** | **Subquery** |
| --- | --- | --- |
| **Definition** | A temporary named result set defined using WITH clause | A query nested inside another query |
| **Readability** | More readable and modular, especially for complex logic | Can become hard to read in deeply nested or repeated queries |
| **Reusability** | Can be referenced multiple times within the same query | Cannot be reused; must be repeated if needed again |
| **Debugging & Maintenance** | Easier to debug and maintain due to named structure | Harder to isolate and troubleshoot |
| **Recursion Support** | Supports recursive queries (e.g., hierarchical data) | Does not support recursion |
| **Performance** | May be optimized better in some databases | Performance varies; may be less efficient for repeated logic |
| **Scope** | Exists only during execution of the query | Exists only within the query block where it's defined |
| **Use Case** | Ideal for breaking down complex queries into readable blocks | Ideal for filtering, aggregation, or conditional logic |

**🧠 Example**

-- CTE: Get departments with more than 5 employees

WITH dept\_counts AS (

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

FROM employees

GROUP BY department\_id

)

SELECT e.name, d.department\_id

FROM employees e

JOIN dept\_counts d ON e.department\_id = d.department\_id

WHERE d.emp\_count > 5;

-- Subquery: Same logic using nested query

SELECT e.name, e.department\_id

FROM employees e

WHERE e.department\_id IN (

SELECT department\_id

FROM employees

GROUP BY department\_id

HAVING COUNT(\*) > 5

);

**✅ Summary**

* Use a **CTE** when you need modular, reusable logic—especially for complex or recursive queries.
* Use a **subquery** for simpler, one-off filtering or aggregation tasks.

-----------------------------------------------------------------------------------------------------

**🔹 Filtering & Grouping**

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

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

| **Aspect** | **WHERE Clause** | **HAVING Clause** |
| --- | --- | --- |
| **Purpose** | Filters rows **before** grouping or aggregation | Filters groups or aggregated results **after** GROUP BY |
| **Applicable To** | Individual rows | Grouped rows or aggregate functions |
| **Used With** | SELECT, UPDATE, DELETE | Typically used with GROUP BY and aggregate functions (SUM, AVG) |
| **Aggregate Filtering** | Cannot filter on aggregate functions directly | Can filter using aggregate functions like COUNT(), SUM() |
| **Execution Order** | Evaluated **before** GROUP BY | Evaluated **after** GROUP BY |
| **Performance Impact** | Faster for row-level filtering | Used for post-aggregation filtering |

**🧠 Example**

-- WHERE: filters rows before grouping

SELECT department, salary

FROM employees

WHERE salary > 50000;

-- HAVING: filters groups after aggregation

SELECT department, AVG(salary) AS avg\_salary

FROM employees

GROUP BY department

HAVING AVG(salary) > 50000;

**✅ Summary**

* Use **WHERE** to filter raw data before any grouping or aggregation.
* Use **HAVING** to filter grouped or aggregated results after GROUP BY.

-----------------------------------------------------------------------------------------------------

**How does GROUP BY differ from ORDER BY?**

**📊 Difference Between GROUP BY and ORDER BY in SQL**

| **Aspect** | **GROUP BY** | **ORDER BY** |
| --- | --- | --- |
| **Purpose** | Groups rows that have the same values in specified columns | Sorts the result set based on one or more columns |
| **Usage Context** | Typically used with aggregate functions like SUM(), COUNT(), AVG() | Used to arrange query results in ascending or descending order |
| **Effect on Data** | Reduces rows by grouping them | Does not reduce rows; only changes their order |
| **Execution Order** | Executed **before** ORDER BY | Executed **after** GROUP BY |
| **Required With Aggregates** | Required when using aggregate functions across grouped data | Optional; used for presentation purposes |
| **Sorting Capability** | Does **not** sort data unless combined with ORDER BY | Explicitly sorts data based on specified columns |

**🧠 Example**

-- GROUP BY: groups employees by department and calculates average salary

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

FROM employees

GROUP BY department\_id;

-- ORDER BY: sorts employees by salary in descending order

SELECT name, salary

FROM employees

ORDER BY salary DESC;

**✅ Summary**

* Use **GROUP BY** to aggregate and group data based on shared column values.
* Use **ORDER BY** to sort query results for better readability or reporting.

-----------------------------------------------------------------------------------------------------

Absolutely, Sriram! Let’s enrich each comparison with **multiple dimensions**—covering **definition, behavior, performance, use cases, and examples**—so you get a deeper understanding. Here's a **Word-friendly**, expanded version of the key SQL differences you asked for:

**🔹 Data Types**

**1. CHAR vs VARCHAR**

| **Aspect** | **CHAR** | **VARCHAR** |
| --- | --- | --- |
| **Storage Type** | Fixed-length string | Variable-length string |
| **Padding Behavior** | Pads with spaces to match defined length | Stores only actual characters |
| **Performance** | Slightly faster for fixed-size fields | More efficient for varying-length fields |
| **Use Case** | Codes like country abbreviations ('IN', 'US') | Names, addresses, descriptions |
| **Storage Overhead** | May waste space if values are shorter than defined length | Optimized for space |

**Example:**

CREATE TABLE countries (

code CHAR(2),

name VARCHAR(50)

);

**2. TEXT vs VARCHAR(MAX)**

| **Aspect** | **TEXT** | **VARCHAR(MAX)** |
| --- | --- | --- |
| **Function Support** | Limited support for string functions | Full support for string functions |
| **Deprecation** | Deprecated in many modern databases | Recommended for large text fields |
| **Size Limit** | Up to 2GB (varies by DBMS) | Up to 2GB |
| **Use Case** | Legacy systems | Modern applications requiring large text |
| **Indexing** | Cannot be indexed directly | Can be indexed with full-text indexes |

**Example:**

CREATE TABLE articles (

legacy\_content TEXT,

new\_content VARCHAR(MAX)

);

**3. DATE vs DATETIME vs TIMESTAMP**

| **Aspect** | **DATE** | **DATETIME** | **TIMESTAMP** |
| --- | --- | --- | --- |
| **Precision** | Stores only date | Stores date and time | Stores date, time, and time zone info |
| **Format** | YYYY-MM-DD | YYYY-MM-DD HH:MM:SS | Varies by DBMS |
| **Time Zone** | No | No | Yes (in some databases) |
| **Use Case** | Birthdates, deadlines | Event logs, timestamps | Auditing, replication tracking |
| **Auto Update** | No | No | Often auto-updated on row change |

**Example:**

CREATE TABLE logs (

created\_date DATE,

created\_time DATETIME,

updated\_time TIMESTAMP

);

**🔹 Set Operations**

**4. UNION vs UNION ALL**

| **Aspect** | **UNION** | **UNION ALL** |
| --- | --- | --- |
| **Duplicates** | Removes duplicates | Keeps duplicates |
| **Performance** | Slower due to deduplication | Faster due to no deduplication |
| **Sorting** | May sort internally | No sorting unless explicitly specified |
| **Use Case** | When unique results are needed | When all results including duplicates are needed |

**Example:**

SELECT name FROM employees

UNION

SELECT name FROM managers;

SELECT name FROM employees

UNION ALL

SELECT name FROM managers;

**5. EXCEPT vs NOT IN**

| **Aspect** | **EXCEPT** | **NOT IN** |
| --- | --- | --- |
| **Functionality** | Returns rows from first query not in second | Filters rows not matching values in a list |
| **Null Handling** | Ignores NULLs | May behave unpredictably with NULLs |
| **Performance** | Set-based, often faster | May be slower with large subqueries |
| **Use Case** | Set exclusion | Conditional filtering |

**Example:**

-- EXCEPT

SELECT id FROM employees

EXCEPT

SELECT id FROM terminated\_employees;

-- NOT IN

SELECT id FROM employees

WHERE id NOT IN (SELECT id FROM terminated\_employees);

**6. INTERSECT vs INNER JOIN**

| **Aspect** | **INTERSECT** | **INNER JOIN** |
| --- | --- | --- |
| **Purpose** | Returns common rows from two queries | Combines rows based on matching column values |
| **Column Match** | Requires same columns and data types | Can join on different columns |
| **Use Case** | Set-based comparison | Relational data retrieval |
| **Flexibility** | Less flexible | Highly flexible with join conditions |

**Example:**

-- INTERSECT

SELECT id FROM employees

INTERSECT

SELECT id FROM managers;

-- INNER JOIN

SELECT e.id, m.department

FROM employees e

INNER JOIN managers m ON e.id = m.id;

**🔹 Indexing**

**7. Clustered vs Non-Clustered Index**

| **Aspect** | **Clustered Index** | **Non-Clustered Index** |
| --- | --- | --- |
| **Storage** | Sorts and stores data physically | Stores pointers to data rows |
| **Count per Table** | Only one per table | Multiple allowed |
| **Performance** | Faster for range queries | Faster for point lookups |
| **Use Case** | Primary key indexing | Secondary column indexing |

**Example:**

CREATE CLUSTERED INDEX idx\_emp\_id ON employees(id);

CREATE NONCLUSTERED INDEX idx\_emp\_name ON employees(name);

**8. Composite vs Covering Index**

| **Aspect** | **Composite Index** | **Covering Index** |
| --- | --- | --- |
| **Definition** | Index on multiple columns | Index that includes all columns used in a query |
| **Purpose** | Improves multi-column filtering | Avoids accessing base table |
| **Use Case** | WHERE clause with multiple columns | SELECT queries with full index coverage |

**Example:**

-- Composite

CREATE INDEX idx\_emp\_dept ON employees(department\_id, hire\_date);

-- Covering

-- SELECT name, hire\_date WHERE department\_id = 10;

-- Covered by index on (department\_id, name, hire\_date)

**🔹 Ranking & Window Functions**

**9. RANK() vs DENSE\_RANK()**

| **Aspect** | **RANK()** | **DENSE\_RANK()** |
| --- | --- | --- |
| **Ranking Gaps** | Skips ranks after ties | No gaps; ranks are sequential |
| **Use Case** | When gaps in ranking are acceptable | When continuous ranking is needed |

**Example:**

SELECT name, salary,

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

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

FROM employees;

**10. ROW\_NUMBER() vs RANK()**

| **Aspect** | **ROW\_NUMBER()** | **RANK()** |
| --- | --- | --- |
| **Uniqueness** | Always assigns unique row numbers | Assigns same rank to tied values |
| **Use Case** | Pagination, deduplication | Ranking with tie handling |

**Example:**

SELECT name, salary,

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

RANK() OVER (ORDER BY salary DESC) AS rank

FROM employees;

**🔹 Stored Logic**

**11. Stored Procedure vs Function**

| **Aspect** | **Stored Procedure** | **Function** |
| --- | --- | --- |
| **Return Type** | Can return multiple values or none | Must return a single value |
| **Usage** | Executed independently | Used within SQL expressions |
| **Side Effects** | Can modify data | Typically read-only |

**Example:**

-- Procedure

CREATE PROCEDURE GetEmployees AS

BEGIN

SELECT \* FROM employees;

END;

-- Function

CREATE FUNCTION GetTotalSalary() RETURNS INT AS

BEGIN

RETURN (SELECT SUM(salary) FROM employees);

END;

**12. Trigger vs Stored Procedure**

| **Aspect** | **Trigger** | **Stored Procedure** |
| --- | --- | --- |
| **Execution** | Auto-invoked on DML events | Manually executed |
| **Use Case** | Enforcing business rules, auditing | Encapsulating logic for reuse |

**Example:**

-- Trigger

CREATE TRIGGER trg\_after\_insert

ON employees

AFTER INSERT

AS

BEGIN

PRINT 'New employee added';

END;

**🔹 Null Handling & Expressions**

**1️⃣ Difference Between NVL and COALESCE**

| **Aspect** | **NVL() (Oracle-specific)** | **COALESCE() (ANSI SQL standard)** |
| --- | --- | --- |
| **Argument Count** | Accepts exactly **2 arguments** | Accepts **2 or more arguments** |
| **Return Logic** | Returns second value if first is NULL | Returns the **first non-null** value from the list |
| **Portability** | Limited to Oracle | Supported across most SQL databases (PostgreSQL, SQL Server, etc.) |
| **Use Case** | Simple null replacement | Multi-level fallback for null values |
| **Function Nesting** | Cannot handle multiple fallback layers | Ideal for cascading fallback logic |

**Example:**

-- NVL: Oracle only

SELECT NVL(manager\_name, 'No Manager') FROM employees;

-- COALESCE: Cross-platform

SELECT COALESCE(manager\_name, department\_head, 'No Manager') FROM employees;

**2️⃣ Difference Between IS NULL and = NULL**

| **Aspect** | **IS NULL** | **= NULL** |
| --- | --- | --- |
| **Functionality** | Correct way to check for NULL values | Invalid comparison; always returns false |
| **SQL Behavior** | Recognized by SQL engines | Violates SQL's three-valued logic (TRUE, FALSE, UNKNOWN) |
| **Use Case** | Filtering rows with missing values | Should be avoided |
| **Portability** | Works across all SQL dialects | Not supported anywhere |

**Example:**

-- Correct usage

SELECT \* FROM employees WHERE manager\_id IS NULL;

-- Incorrect usage

SELECT \* FROM employees WHERE manager\_id = NULL; -- Always false

**3️⃣ Difference Between CASE and IF in SQL**

| **Aspect** | **CASE (ANSI SQL standard)** | **IF (MySQL-specific)** |
| --- | --- | --- |
| **Syntax Type** | Expression-based conditional logic | Function-style conditional logic |
| **Portability** | Works in all major SQL databases | Limited to MySQL and some procedural SQL dialects |
| **Flexibility** | Supports multiple WHEN conditions | Supports simple IF(condition, true\_value, false\_value) |
| **Use Case** | Complex branching logic in queries | Simple binary decisions |

**Example:**

-- CASE: portable across SQL engines

SELECT name,

CASE

WHEN salary > 50000 THEN 'High'

WHEN salary > 30000 THEN 'Medium'

ELSE 'Low'

END AS salary\_band

FROM employees;

-- IF: MySQL only

SELECT name,

IF(salary > 50000, 'High', 'Low') AS salary\_band

FROM employees;

**🧠 Essential SQL Functions to Know**

**🔹 Aggregate Functions**

Used for summarizing data:

* COUNT() – Counts rows
* SUM() – Adds numeric values
* AVG() – Calculates average
* MIN() / MAX() – Finds smallest/largest value

**🔹 String Functions**

Useful for text manipulation:

* UPPER() / LOWER() – Converts case
* TRIM() / LTRIM() / RTRIM() – Removes whitespace
* SUBSTRING() / LEFT() / RIGHT() – Extracts parts of strings
* REPLACE() – Substitutes characters
* CHARINDEX() / INSTR() – Finds position of substring
* CONCAT() – Joins strings

**🔹 Date & Time Functions**

Critical for time-based analysis:

* GETDATE() / CURRENT\_DATE – Returns current date
* DATEADD() – Adds interval to date
* DATEDIFF() – Calculates difference between dates
* YEAR() / MONTH() / DAY() – Extracts components
* FORMAT() – Formats date/time output

**🔹 Conversion Functions**

For type casting and formatting:

* CAST() – Converts data type
* CONVERT() – Converts with formatting (SQL Server)
* TO\_CHAR() / TO\_DATE() – Oracle-specific conversions

**🔹 Conditional Functions**

For logic and branching:

* CASE – Conditional logic (like if-else)
* COALESCE() – Returns first non-null value
* NULLIF() – Returns NULL if two expressions are equal
* IF() – MySQL-specific conditional

**🔹 Window Functions**

Powerful for ranking and analytics:

* ROW\_NUMBER() – Unique row ID per partition
* RANK() / DENSE\_RANK() – Ranking with/without gaps
* NTILE() – Divides rows into buckets
* LEAD() / LAG() – Access next/previous row
* FIRST\_VALUE() / LAST\_VALUE() – Gets boundary values

**🔹 Mathematical Functions**

For numeric operations:

* ROUND() – Rounds numbers
* CEILING() / FLOOR() – Rounds up/down
* ABS() – Absolute value
* POWER() / SQRT() – Exponents and roots
* MOD() – Modulo operation

**🔹 JSON & Array Functions (Modern SQL)**

Useful in semi-structured data:

* JSON\_VALUE() / JSON\_EXTRACT() – Extracts JSON fields
* ARRAY\_AGG() – Aggregates values into arrays
* UNNEST() – Expands arrays into rows