**SQL (STRUCTURED QUERY LANGUAGE) LEARNINGS**

**Normalization:**

**What is Normalization?**

Normalization is a process of minimizing redundancy (repetition of data in a table).

**What does it do?**

Normalization divides the large table into smaller tables and links then using relationships like one to one, one to many and many to many that links tables are primary keys and foreign keys.

**Why Do We Use Normalization?**

It helps to:

* Eliminate data redundancy.
* Prevent data anomalies (insert, update, delete).
* Improve data efficiency and consistency.

**Key Normal Forms:**

1. **1NF (First Normal Form)**: splits multi-values into individual cells.
2. **2NF (Second Normal Form)**: ensures no partial dependencies on part of the composite key.
3. **3NF (Third Normal Form)**: Ensures no indirect dependencies on the primary key (non-key attributes depend only on the primary key).
4. **BCNF (Boyce-Codd Normal Form)**: A stricter version of 3NF, ensuring that every determinant is a candidate key.
5. **4NF:** Avids rows having unrelated multi-values.
6. **5NF:** Breakdown complex relationships into smaller independent pieces.

**Example:**

**Real World Example:**

Consider a **Customer Orders** table:

| **CustomerID** | **CustomerName** | **OrderID** | **OrderDate** | **Product** | **Quantity** |
| --- | --- | --- | --- | --- | --- |
| 1 | John Doe | 101 | 2023-01-01 | Laptop | 2 |
| 1 | John Doe | 102 | 2023-01-02 | Mouse | 1 |

This table violates normalization rules, as:

* **Redundancy**: The customer information (John Doe) is repeated for every order.
* **Anomalies**: If John’s address changes, it needs to be updated in multiple rows; deleting an order would also remove customer information.

After normalization, the data could be split into two tables:

1. **Customers Table**:

| **CustomerID** | **CustomerName** |
| --- | --- |
| 1 | John Doe |

1. **Orders Table**:

| **OrderID** | **CustomerID** | **OrderDate** | **Product** | **Quantity** |
| --- | --- | --- | --- | --- |
| 101 | 1 | 2023-01-01 | Laptop | 2 |
| 102 | 1 | 2023-01-02 | Mouse | 1 |

Here, customer information is stored once, and orders are linked to the customer through the CustomerID, reducing redundancy and avoiding anomalies.

**Subqueries and Correlated Subqueries:**

**1. Subqueries:**

**Definition:**

A subquery is a query embedded inside another query. It is executed once, and its result is used by the outer query.

**Example 1: Simple Subquery in WHERE Clause**

Let's assume we have two tables: employees and departments.

SELECT name

FROM employees

WHERE department\_id = (SELECT department\_id

FROM departments

WHERE department\_name = 'Sales').

**2. Correlated Subqueries:**

**Definition:**

A correlated subquery is a subquery that refers to columns from the outer query. Unlike a regular subquery, which is independent and executed once, a correlated subquery is evaluated once for each row processed by the outer query.

**Example 1: Correlated Subquery in WHERE Clause**

SELECT name, salary

FROM employees e

WHERE salary > (SELECT AVG (salary)

FROM employees

WHERE department\_id = e. department\_id).

|  |  |  |
| --- | --- | --- |
| Feature | Subquery | Correlated subquery |
| Execution | Executed once for the entire outer query | Executed once for each row of the outer query |
| Independence | Independent of the outer query | |  | | --- | |  |  |  | | --- | |  | |  |  |  | | --- | | Dependent on the outer query’s row values | |
| References to Outer Query | Does not reference the outer query’s columns | References columns from the outer query |
| Use Case | |  | | --- | |  |  |  | | --- | | Used for fixed values or aggregations | | Used when row-by-row processing is needed |

**When to Use Subqueries vs. Correlated Subqueries:**

* Subqueries are ideal when you need to perform a single, independent query to retrieve a result that can be used in the outer query.
* Correlated Subqueries are more appropriate when the inner query depends on each row from the outer query, often when the values need to change dynamically for each row.

**UNION and INTERSECT in SQL**

**1. UNION**

Definition:

* The UNION operator combines the result sets of two or more SELECT queries, eliminating duplicate rows in the result.
* The columns in each SELECT query must match in both number and data type.

**Use Case:**

* When you need to combine data from multiple tables that have the same structure.

**Syntax:**

SELECT column1, column2

FROM table1

UNION

SELECT column1, column2

FROM table2.

* Key Point: If there are duplicate rows in the result set, UNION will eliminate them.

**2. INTERSECT**

**Definition:**

* The INTERSECT operator returns only the rows that are common to both SELECT queries (i.e., the intersection of both result sets).
* Like UNION, the columns in both queries must match in number and type.

**Use Case:**

* When you need to find the common records between two result sets.

**Syntax:**

SELECT column1, column2

FROM table1

INTERSECT

SELECT column1, column2

FROM table2.

|  |  |  |
| --- | --- | --- |
| Feature | Union | Intersect |
| Operation | |  | | --- | |  |  |  | | --- | | Combines all results from both queries, removing duplicates | | Returns only the common results between two queries |
| Duplicates | Eliminates duplicates | |  | | --- | |  |  |  | | --- | | Only keeps duplicates (common rows) | |
| Use Case | When combining datasets | When finding common records |

**Conclusion:**

* Use UNION to combine datasets and eliminate duplicates.
* Use INTERSECT to find common records between two datasets.

**Aggregate Functions in SQL**

1. SUM ()

**Syntax:**

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

2. AVG ()

**Syntax:**

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

3. MIN ()

**Syntax:**

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

4. MAX ()

**Syntax:**

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

5. COUNT ()

**Syntax:**

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

SELECT department, AVG (salary) FROM employees GROUP BY department.

**Window Functions in SQL**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**1. ROW\_NUMBER():** Assigns a unique number to each row based on the ordering provided.

SELECT emp\_id, emp\_name, salary,

ROW\_NUMBER() OVER (ORDER BY salary DESC) AS row\_num FROM employeetable;

A screenshot of a computer

Description automatically generated

**2. RANK():** Ranks the rows, but if two rows have the same value, they will get the same rank, and the next rank will be skipped.

SELECT emp\_id, emp\_name, salary,

RANK() OVER (ORDER BY salary DESC) AS rank FROM employees;

**A screenshot of a computer

Description automatically generated**

**3. DENSE\_RANK():** Like RANK(), but it does not skip ranks if there are ties.

SELECT emp\_id, emp\_name, salary,

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

A screenshot of a computer

Description automatically generated

**4. LEAD()**: Returns the value of a column for the next row in the result set.

SELECT emp\_id, emp\_name, salary,

LEAD(salary, 1) OVER (ORDER BY salary DESC) AS next\_salary FROM employees;

A screenshot of a computer

Description automatically generated

**5. LAG()**: Returns the value of a column for the previous row in the result set.

SELECT emp\_id, emp\_name, salary,

LAG(salary, 1) OVER (ORDER BY salary DESC) AS prev\_salary

FROM employees;

A screenshot of a computer

Description automatically generated

**Key Differences Between the Functions:**

| **Function** | **Purpose** | **Behaviour With Ties** |
| --- | --- | --- |
| ROW\_NUMBER () | Assigns a unique sequential number to each row. | No ties (unique row numbers). |
| RANK () | Assigns ranks, with gaps in case of ties. | Skips ranks for ties (e.g., two tied rows will both get rank 1, and the next rank will be 3). |
| DENSE\_RANK () | Assigns ranks, without gaps for ties. | No gaps in ranks (e.g., two tied rows will both get rank 1, and the next rank will be 2). |
| LEAD () | Accesses the value from the next row. | Returns NULL for the last row. |
| LAG () | Accesses the value from the previous row. | Returns NULL for the first row. |

**Common Table Expressions (CTEs)**

**What is a CTE?**

A **Common Table Expression (CTE)** is a temporary result set defined within a SELECT, INSERT, UPDATE, or DELETE statement. It helps simplify complex queries, improve readability, and break down complex subqueries.

**Why is it Useful?**

* **Improves query readability** by breaking down complex queries into simpler parts.
* **Simplifies repetitive queries** by creating temporary result sets that can be referenced multiple times.
* **Makes the query more maintainable** by avoiding nested subqueries.

**How to Use a CTE?**

You define a CTE using the WITH keyword. For example:

WITH EmployeeSalaries AS (

SELECT name, salary

FROM employees

WHERE salary > 50000

)

SELECT \*

FROM EmployeeSalaries;

**Real-Life Example:**

In an organization, if you want to list employees earning above a certain salary, you can use a CTE to simplify the query, like the one above.

**Recursive CTE**

A **Recursive CTE** is a type of CTE that references itself, useful for hierarchical data (e.g., organizational charts).

Example: Finding all employees under a manager in a hierarchy:

WITH RECURSIVE EmployeeHierarchy AS (

-- Base case: Select the top-level manager

SELECT employee\_id, name, manager\_id

FROM employees

WHERE name = 'Alice'

UNION ALL

-- Recursive case: Select employees managed by someone already in the CTE

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

FROM employees e

INNER JOIN EmployeeHierarchy eh ON e.manager\_id = eh.employee\_id

)

SELECT \*

FROM EmployeeHierarchy;

**Key Points:**

* **Base Case**: The starting point of the CTE, which doesn’t reference itself.
* **Recursive Case**: Repeatedly references the CTE itself to find related data.
* **Termination**: Requires a condition to stop the recursion to avoid infinite loops.

**Pivoting and Unpivoting data**

Pivoting: To transform one unique value from one column to multiple columns

**Example Dataset**

Consider a sales table:

| **product\_id** | **month** | **sales** |
| --- | --- | --- |
| 1 | Jan | 100 |
| 1 | Feb | 150 |
| 2 | Jan | 200 |
| 2 | Feb | 300 |

**Pivoting Query**

To transform this data so each month is represented as a column:

SELECT product\_id,

SUM (CASE WHEN month = 'Jan' THEN sales END) AS Jan,

SUM (CASE WHEN month = 'Feb' THEN sales END) AS Feb

FROM sales

GROUP BY product\_id;

**Output:**

| **product\_id** | **Jan** | **Feb** |
| --- | --- | --- |
| 1 | 100 | 150 |
| 2 | 200 | 300 |

**Unpivoting**: The Reverse of pivoting is Unpivoting converting columns into rows.

**Unpivoting Query**

To convert this data back into rows:

SELECT product\_id,

'Jan' AS month,

Jan AS sales

FROM monthly\_sales

UNION ALL

SELECT product\_id,

'Feb' AS month,

Feb AS sales

FROM monthly\_sales;

**Recursive Queries for hierarchical data**

A computer screen shot of a computer code

Description automatically generated

**Definition:** A recursive query is a type of SQL query that references itself to fetch hierarchical or hierarchical-like data, such as organizational structures, family trees, or folder structures.

**Structure:**

* A **Recursive Query** is built using a **Common Table Expression (CTE)**, specifically with the WITH RECURSIVE clause.
* It consists of two parts:
  1. **Base Case**: This is the initial query that retrieves the starting point (e.g., top-level records).
  2. **Recursive Case**: This part refers back to the CTE itself and retrieves related records (e.g., subordinates, child records, etc.).
* The query continues to call itself until a termination condition is met.

**How it Solves Problems:**

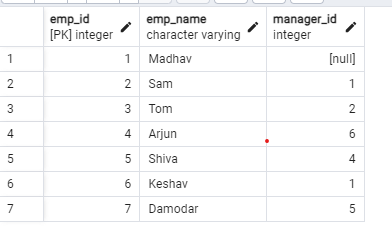
* **Hierarchical Data**: Recursive queries are particularly effective for querying hierarchical structures, where data has relationships that form parent-child hierarchies (e.g., managers and employees, categories and subcategories).
* **Repetitive Tasks**: Recursive queries help in cases where repetitive calculations or data fetching is needed, like navigating through levels of data.

**Benefits:**

* **Solves Hierarchical Queries**: Recursive queries are ideal for dealing with data that has levels, like finding all subordinates under a manager.
* **Improves Readability**: Helps break complex queries into manageable parts, making it easier to write and understand.
* **Efficient**: Can simplify what would otherwise be complex queries with multiple joins or nested subqueries.

In summary, **recursive queries** help manage hierarchical data by allowing you to repeatedly query the same data until the condition is satisfied, making them an efficient solution for managing relationships like managers and employees or parent-child structures.

**String Manipulation (CONCAT, SUBSTRING, REPLACE, TRIM, etc.)**

****

String Manipulation is used for transforming and handling data.

**1. CONCAT**

* **Goal**: Concatenate employee names with a title.

SELECT CONCAT('Employee: ', emp\_name) AS employee\_title FROM employees;

A screenshot of a computer

Description automatically generated

**2. SUBSTRING**

* **Goal**: Extract a part of the employee’s name (e.g., the first three letters).

SELECT emp\_name, SUBSTRING(emp\_name FROM 1 FOR 3) AS short\_name FROM employees;

A screenshot of a computer

Description automatically generated

**3. REPLACE**

* **Goal**: Replace part of an employee’s name.

SELECT emp\_name, REPLACE(emp\_name, 'a', 'A') AS modified\_name FROM employees;

A screenshot of a computer

Description automatically generated

**4. TRIM**

* **Goal**: Remove extra spaces from employee names (for example, if they have trailing spaces).

SELECT emp\_name, TRIM(emp\_name) AS trimmed\_name FROM employees;

A screenshot of a computer

Description automatically generated

**5. LENGTH**

* **Goal**: Find the length of each employee's name.

SELECT emp\_name, LENGTH(emp\_name) AS name\_length FROM employees;

A screenshot of a computer

Description automatically generated

**6. LOWER and UPPER**

* **Goal**: Convert employee names to all uppercase or lowercase.

SELECT emp\_name, LOWER(emp\_name) AS lowercase\_name, UPPER(emp\_name) AS uppercase\_name FROM employees;

A screenshot of a computer

Description automatically generated

**7. POSITION**

* **Goal**: Find the position of a specific letter (e.g., "a") in each employee’s name.

SELECT emp\_name, POSITION('a' IN emp\_name) AS position\_of\_a FROM employees;

A screenshot of a computer

Description automatically generated

**Date and Time Functions (DATEADD, DATEDIFF, DATEPART, etc.)**

**A screenshot of a phone

Description automatically generated**

**1. DATEADD (Using INTERVAL)**

* **Goal**: Add a specific time interval to a date.
* **Example**: Extend each project's end\_date by 30 days.

SELECT project\_name, end\_date, end\_date + INTERVAL '30 days' AS extended\_end\_date FROM projects;

A screenshot of a computer

Description automatically generated

**2. DATEDIFF (Using AGE or - Operator)**

* **Goal**: Calculate the difference between start\_date and end\_date.
* **Example**: Find the total number of days each project ran.

SELECT project\_name, start\_date, end\_date, end\_date - start\_date AS project\_duration FROM projects;

A screenshot of a computer

Description automatically generated

**3. DATEPART (Extract specific parts of a date, e.g., year, month, day)**

* **Goal**: Extract and display specific parts of start\_date.
* **Example**: Retrieve the year, month, and day of start\_date for each project.

SELECT project\_name,

start\_date,

EXTRACT(YEAR FROM start\_date) AS start\_year,

EXTRACT(MONTH FROM start\_date) AS start\_month,

EXTRACT(DAY FROM start\_date) AS start\_day

FROM projects;

A screenshot of a computer

Description automatically generated

**4. CURRENT\_DATE and NOW()**

* **Goal**: Show the current date and time.
* **Example**: List all projects with their end\_date and compare to today’s date to see if the project has ended.

SELECT project\_name, end\_date, CURRENT\_DATE AS today\_date,

CASE WHEN end\_date < CURRENT\_DATE THEN 'Completed' ELSE 'In Progress' END AS project\_status FROM projects;

A screenshot of a computer

Description automatically generated

**5. DATE\_TRUNC**

* **Goal**: Truncate a date to the start of a specified time unit (e.g., year, month).
* **Example**: Get each project’s start date truncated to the beginning of the month.

SELECT project\_name, start\_date, DATE\_TRUNC('month', start\_date) AS month\_start\_date FROM projects;

A screenshot of a computer

Description automatically generated

**Case Statements and Conditional Expressions**

A screenshot of a table

Description automatically generated

**CASE Statements**

**Goal**: Provide a bonus based on department and salary.

SELECT emp\_name,

department,

salary,

CASE

WHEN department = 'IT' AND salary >= 70000 THEN salary \* 0.10

WHEN department = 'HR' AND salary >= 50000 THEN salary \* 0.05

ELSE 0

END AS bonus

FROM employees;

A screenshot of a computer

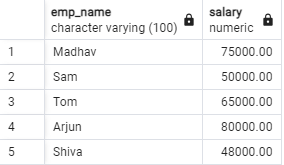
Description automatically generated

**Conditional Expressions**

**1. COALESCE**

COALESCE returns the first non-null value in a list of expressions.

SELECT emp\_name, COALESCE(salary, 0) AS salary FROM employees;



**2. NULLIF**

NULLIF returns NULL if two expressions are equal; otherwise, it returns the first expression.

SELECT emp\_name, NULLIF(salary, 75000) AS adjusted\_salary FROM employees;

A screenshot of a computer

Description automatically generated

**3. Using Logical Expressions in WHERE Clauses**

You can use logical conditions within WHERE to filter data dynamically.

SELECT \* FROM employees WHERE (department = 'IT' AND salary > 70000) OR (department = 'HR' AND salary < 50000);

A screenshot of a computer

Description automatically generated

**User-Defined Functions (scalar and table-valued)**

**Scalar Functions**: Return a single value for each row based on input parameters and can be used for calculations or transformations.

**Table-Valued Functions**: Return a set of rows (a table) and can be used to return a result set that can be queried or joined with other tables.

**Temporary Tables** and **Table Variables**

* Temporary tables exist for the session, while table variables exist only in the procedure or function.
* Temporary tables are more suited for larger, session-wide data, whereas table variables are better for smaller, localized data handling.

**Query Optimization Techniques (indexing strategies, execution plans, and performance tuning)**

**Indexing Strategies**

Indexes speed up data retrieval by creating a structure that helps the database find rows faster. Use single-column indexes for common queries and composite indexes for queries involving multiple columns. However, avoid too many indexes, as they use memory and can slow down data updates.

CREATE INDEX index\_name ON table\_name (column\_name);

**Execution Plans**

Execution plans show how the database processes a query, helping spot inefficiencies. Use EXPLAIN to see if the database is scanning the whole table (slow) or using indexes (faster).

EXPLAIN SELECT \* FROM table\_name WHERE condition;

**Performance Tuning**

Performance tuning means adjusting queries and settings to make the database faster. Tips include simplifying queries, using joins carefully, and using caching. Regularly check and optimize slow queries by improving indexes, restructuring schemas, or rewriting queries.

**External Query Filters**

**External Query Filters** are when filtering is done outside of the SQL query, usually in the application code.

**Key Points:**

* **Definition**: Filtering data in the application, not directly in SQL.
* **Use Case**: Useful for complex logic that’s hard to do in SQL or when more flexibility is needed.
* **Examples**:
  + Fetching all employees and filtering by department in the application.
  + Getting all orders and filtering by date in the app.

**Pros and Cons:**

* **Pros**: Gives more flexibility in filtering.
* **Cons**: Less efficient, especially for large datasets, since more data is fetched.

This method is best avoided for large data, as SQL filtering is usually faster.

**Dynamic SQL for generating flexible queries**

When the requirement needs to get from the client and then need to be executed then we can use dynamic query. We can use these when we want to filter or sort dynamically.

Example

DO $$

DECLARE

dept\_name TEXT := 'IT'; -- user input

min\_salary DECIMAL := 50000; -- user input

max\_salary DECIMAL := 80000; -- user input

query TEXT;

BEGIN

query := 'SELECT \* FROM employees WHERE department = $1 AND salary BETWEEN $2 AND $3';

EXECUTE query USING dept\_name, min\_salary, max\_salary;

END $$;

**Advanced Indexing (covering indexes, filtered indexes)**

To improve the speed of retrieving the data we use indexing

Indexing is a technique that helps speed up data retrieval in a database by creating a special structure that links data values to their locations, much like a directory in a book. This allows the database to quickly find the data it needs, instead of scanning every row. It improves query performance, especially with large datasets, by reducing the amount of data the database needs to process, making everything faster and more efficient.

**Partitioning and Sharding for performance improvement on large datasets**

Partitioning and sharding are both techniques for organizing large data sets into smaller subsets, but they differ in how data is distributed:

**Partitioning**:  
Partitioning is like dividing a big table of data into smaller, more manageable pieces. These smaller pieces, called partitions, are still part of the same table, but they are stored separately based on certain criteria (like a range of values, categories, or hash values). This helps improve performance when working with large datasets because the database only needs to look at the relevant partition instead of scanning the entire table.

For example, imagine a table with employee data. If you partition the table by salary ranges, the database will only need to search within the relevant salary range partition instead of going through all the employees.

**Sharding:**  
Sharding is a way of distributing data across multiple databases or servers. It’s similar to partitioning, but while partitioning happens within a single database, sharding splits the data across different servers. Each "shard" is an independent piece of the dataset, and the system can direct queries to the correct shard based on the data being requested.

For instance, if you have a global database of customer records, sharding could divide the data based on regions (e.g., customers in the U.S. on one server, customers in Europe on another). This helps with performance and scaling because each server handles a smaller portion of the data.

Simple Summary:

* **Partitioning**: Breaking large tables into smaller parts within the same database for faster queries.
* **Sharding:** Distributing data across multiple databases or servers to handle large datasets efficiently.

**Joins (including inner, outer, cross, and self joins)**

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

**1. Inner Join**

**Definition:** Returns rows with matching values in both tables.

SELECT products.product\_name, categories.category\_name

FROM products

INNER JOIN categories ON products.category\_id = categories.category\_id;

A screenshot of a phone

Description automatically generated

**2. Left Outer Join**

**Definition:** Returns all rows from the left table (products), with matching rows from the right table (categories). If no match is found, it fills with NULLs.

SELECT products.product\_name, categories.category\_name

FROM products

LEFT JOIN categories ON products.category\_id = categories.category\_id;

A screenshot of a computer

Description automatically generated

**3. Right Outer Join**

**Definition:** Returns all rows from the right table (categories), with matching rows from the left table (products). Fills with NULL if no match.

SELECT products.product\_name, categories.category\_name FROM products

RIGHT JOIN categories ON products.category\_id = categories.category\_id;

A screenshot of a computer

Description automatically generated

**4. Full Outer Join**

**Definition:** Combines left and right outer joins, returning all rows from both tables. If no match is found, it fills with NULLs.

SELECT products.product\_name, categories.category\_name FROM products

FULL OUTER JOIN categories ON products.category\_id = categories.category\_id;

A screenshot of a computer

Description automatically generated

**5. Cross Join**

**Definition:** Returns a Cartesian product, i.e., all possible combinations of rows from both tables.

**Query:**

SELECT products.product\_name, categories.category\_name

FROM products

CROSS JOIN categories;

A screenshot of a computer

Description automatically generated

**6. Self Join**

**Definition:** Used when a table needs to be joined to itself. Common for hierarchical data, duplicate detection, and pairing related data.

**Example Table:** Suppose products has a column related\_product\_id that refers to another product.

**Query:**

SELECT p1.product\_name AS Product\_1, p2.product\_name AS Product\_2, p1.category\_id

FROM products p1

JOIN products p2 ON p1.category\_id = p2.category\_id

WHERE p1.product\_id < p2.product\_id AND p1.category\_id IS NOT NULL;A screenshot of a computer

Description automatically generated