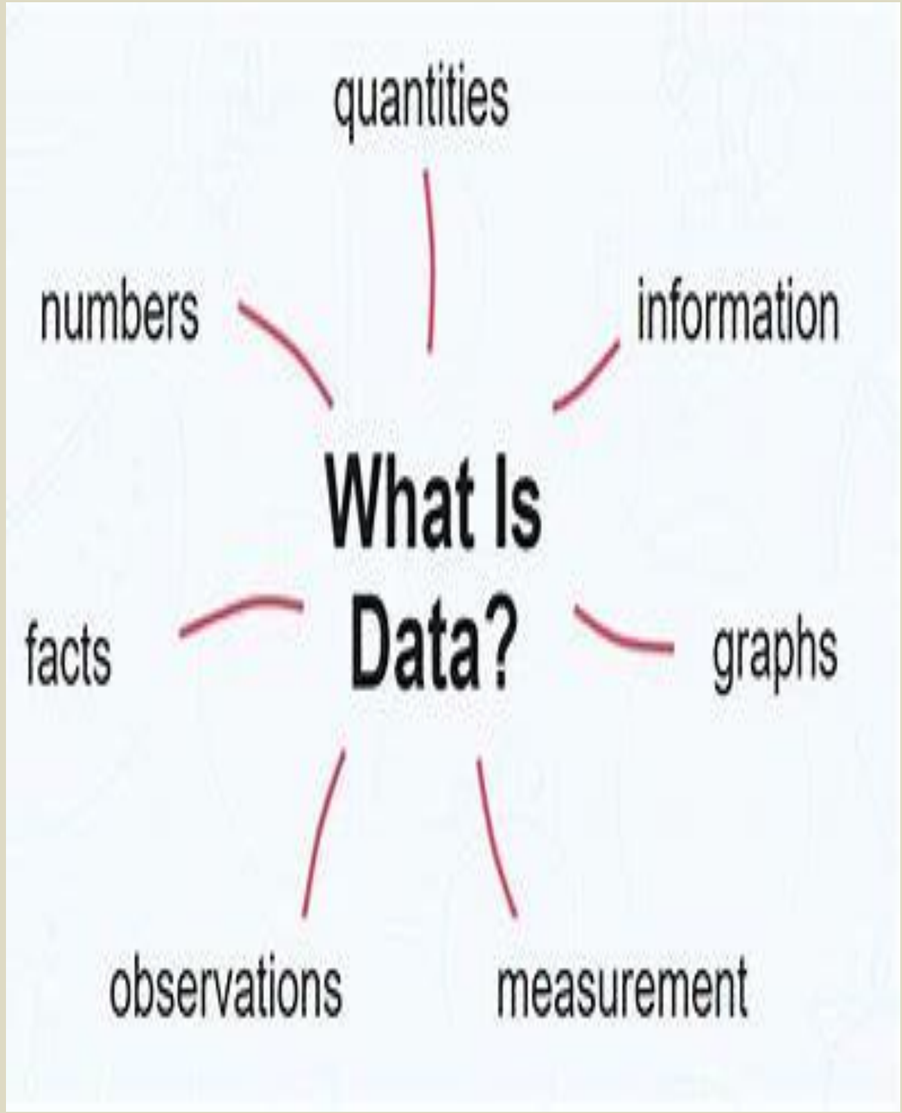
S

Q

L

Data

- Data refers to any information or facts that can be recorded, stored, and processed.
- It can be in various forms, such as text, numbers, images, audio, or video.
- Examples of data include customer names, product prices, temperature readings, and more.



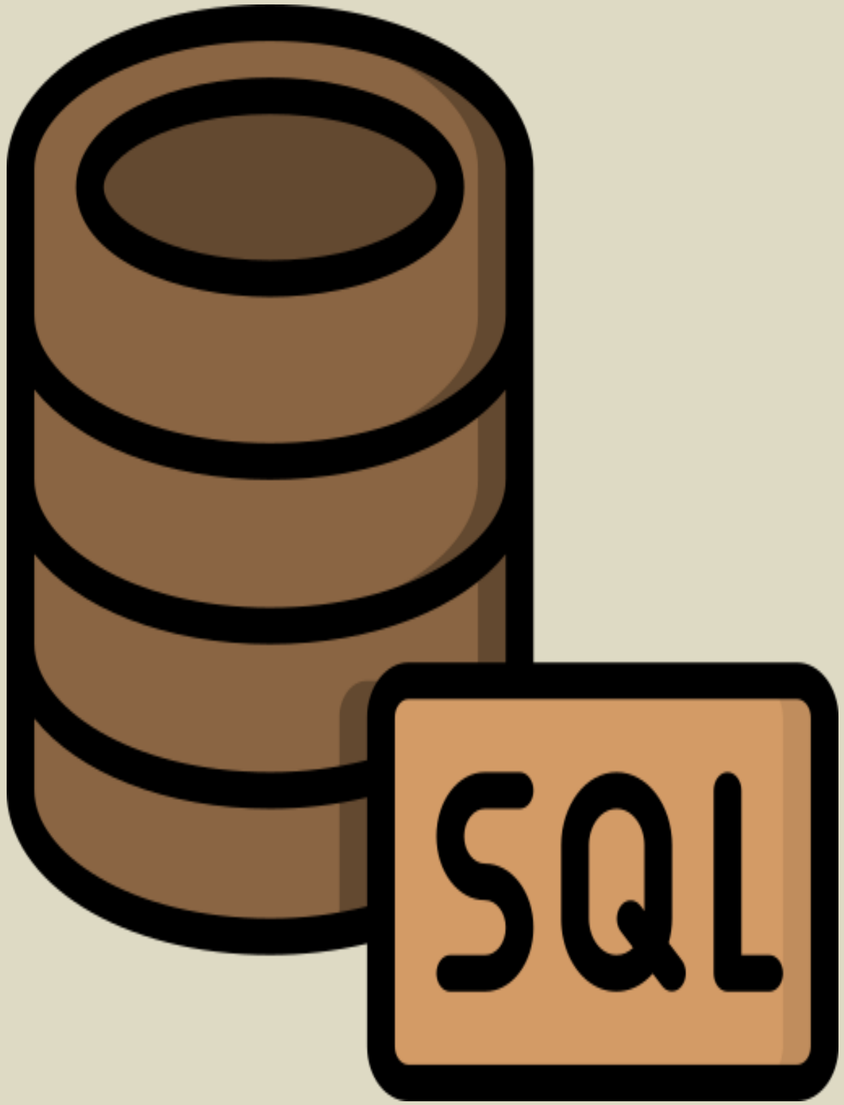
S

Q

L

Database

- A database is a structured collection of data that is organized for efficient storage, retrieval, and management.
- It provides a way to store and manage large amounts of data in a systematic manner.
- Databases can be relational (using tables) or non-relational (using other data models).



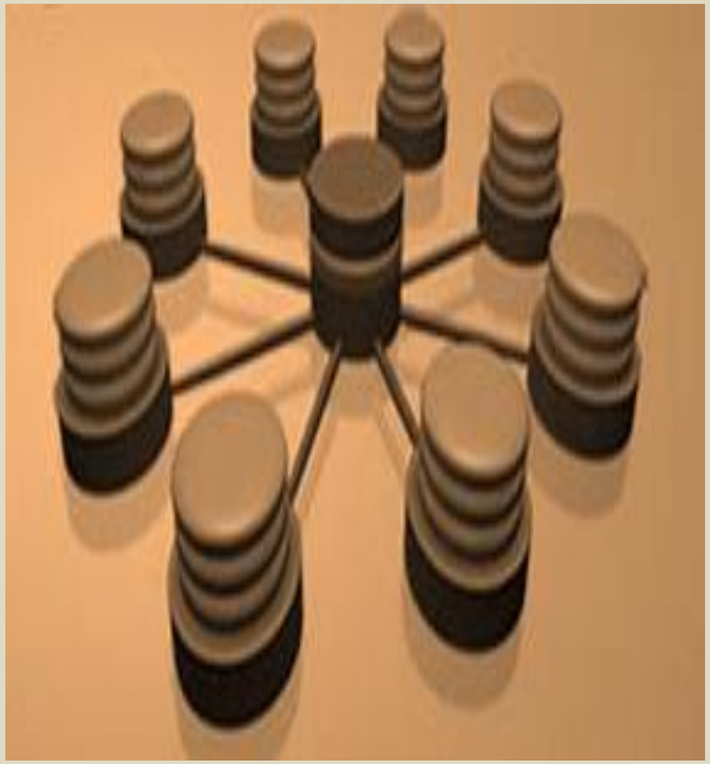
S

Q

L

DBMS (Database Management System):

- A DBMS is software that allows users to interact with databases.
- It provides tools for creating, modifying, querying, and maintaining databases.
- Examples of DBMS include MySQL, Microsoft SQL Server, Oracle, and PostgreSQL.



S

Q

L


Minimum Duplication and Redundancy
Large Database Maintenance
Saves Storage Space and Cost
Features of Database Management System
Provides High Level of Security
Anyone Can Work on it
Multi-User Access

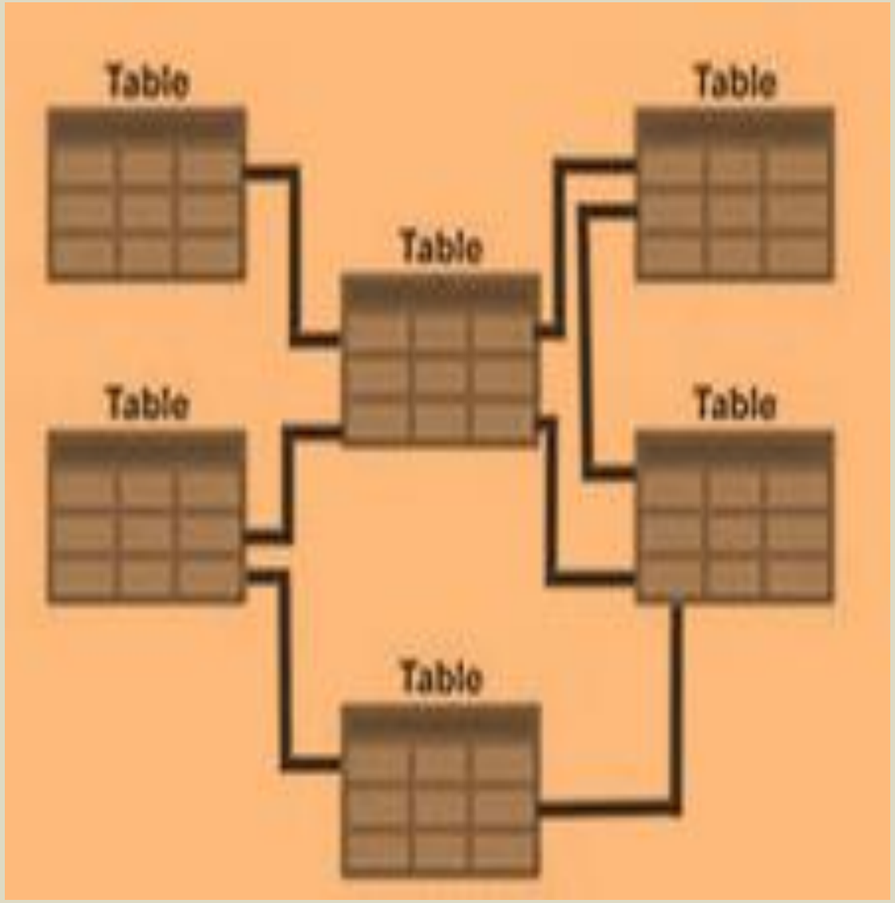
S

Q

L

RDBMS (Relational Database Management System):

- An RDBMS is a type of DBMS that follows the relational model.
- It organizes data into tables with rows (records) and columns (attributes).
- Relationships between tables are defined using keys (e.g., primary keys and foreign keys).
- SQL (Structured Query Language) is used to manipulate data in RDBMS.



S

Q

L

RELATIONAL DATABASE MANAGEMENT SYSTEM

Features

01

Provides data to be stored in tables.

02

Persists data in the form of rows and columns.

03

Provides a primary key to identify the rows in a table uniquely.

06

And it features four types of relationships in the database.

05

Provides multi-user accessibility that individual users can control.

04

Provides a virtual table creation in which sensitive data can be stored.

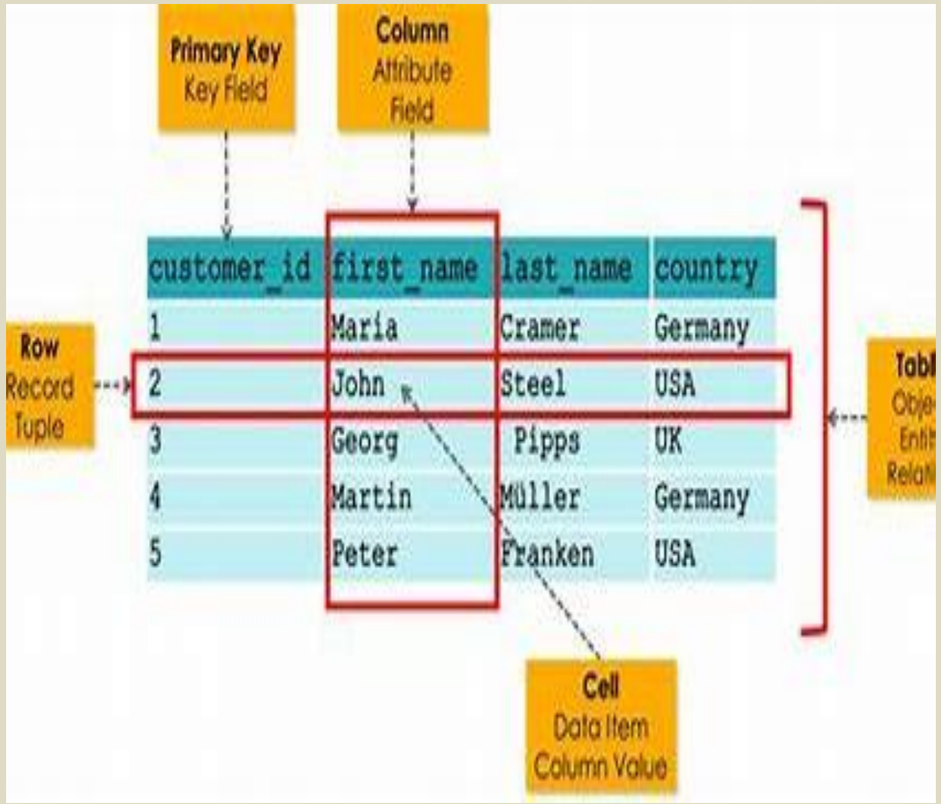
S

Q

L

Tables

- Tables are fundamental components of an RDBMS.
- A table represents a collection of related data entries.
- Each table consists of columns (fields) and rows (records).
- Columns hold specific information about each record, and rows represent individual entries.



S

Q

L

SQL Vs NOSQL Databases

SQL (Relational Databases)

- Uses structured query language (SQL)
- Has a predefined schema
- Data is stored in tables with rows and columns
- Best suited for complex queries
- Vertically scalable (increase server capacity)
- ACID properties (Atomicity, Consistency, Isolation, Durability)
- Ideal for multi-row transactions
- Examples: MySQL, PostgreSQL, Oracle, SQL Server, SQLite

NoSQL (Non-Relational Databases)

- May support SQL-like queries but not strictly SQL
- Has a dynamic schema for unstructured data
- Data can be stored in various formats like key-value, document, graph, or wide-column stores
- Suited for unstructured data and rapid development
- Horizontally scalable (add more servers)
- Focus on performance and flexibility, may not strictly adhere to ACID
- Good for large sets of distributed data
- Examples: MongoDB, Cassandra, Couchbase, Redis

S

Q

L

Databases vs Excel

Aspect	Databases (SQL/NoSQL)	Excel
Type	Databases can be relational (SQL) or non-relational (NoSQL).	Excel is a spreadsheet application for organizing and analyzing data.
Data Structure	Databases store data in tables, rows, and columns.	Excel uses worksheets with cells arranged in rows and columns.
Schema	SQL databases have a predefined schema; NoSQL databases have a dynamic schema.	Excel does not enforce a strict schema; you can add data freely.
Use Cases	SQL databases are ideal for multi-row transactions (e.g., accounting systems).	Excel is commonly used for small-scale data analysis, calculations, and reporting.
Complexity	SQL databases handle complex queries and large datasets.	Excel is simpler and suited for individual users or small teams.
Structured vs. Unstructured	Databases structure data; NoSQL databases handle unstructured data (e.g., documents).	Excel works with structured and semi-structured data (e.g., tables, charts).
Examples	SQL: MySQL, PostgreSQL, Oracle, SQL Server, SQLite. NoSQL: MongoDB, Cassandra, Redis, Neo4j.	Excel is a standalone application widely used for data manipulation and analysis.

S

Q

L

SQl Data Types

SQL

Data Types	Description	Example
Character/String	◦ CHAR(n) : Fixed-length character strings (e.g., storing employee codes). ◦ VARCHAR(n) : Variable-length character strings (e.g., storing names). ◦ TEXT : Large text data (e.g., storing descriptions).	CREATE TABLE products (product_id INT PRIMARY KEY, product_name **VARCHAR**(50), description **TEXT**, category **CHAR**(20));
Numeric	◦ INT : Stores whole numbers (e.g., employee IDs). ◦ DECIMAL(p, s) : Fixed-point decimal numbers (e.g., salary with precision and scale). ◦ FLOAT or REAL : Approximate floating-point numbers (e.g., stock prices).	CREATE TABLE sample_data (id **INT** PRIMARY KEY, age **INT**, height **FLOAT**, price **DECIMAL**(10, 2));
Date and Time	◦ DATE : Stores dates (e.g., order dates). ◦ TIME : Stores time of day (e.g., appointment times). ◦ DATETIME or TIMESTAMP : Stores both date and time.	CREATE TABLE Orders (order_id INT PRIMARY KEY, order_date **DATE**, delivery_time **TIME**, created_at **DATETIME**);

S

Q

L

SQL CONSTRAINTS

S

Q

L

Constraints	Description	Example

NOT NULL	◦ Ensures a column cannot have NULL values.	`CREATE TABLE Customers (`
		`  customer_id INT **NOT NULL,**`
		`  customer_name VARCHAR(100)`
		`  NOT NULL`
		`);`

UNIQUE	◦ Requires all values in a column to be unique.	`CREATE TABLE Products (`
		`  product_id INT PRIMARY KEY,`
		`  product_code VARCHAR(20)`
		`  **UNIQUE,**`
		`  product_name VARCHAR(100)`
		`);`

PRIMARY KEY	◦ Uniquely identifies each row in a table.
	◦ Combination of NOT NULL and UNIQUE constraints.
		`CREATE TABLE Students (`
		`  student_id INT **PRIMARY KEY,**`
		`  student_name VARCHAR(50)`
		`NOT NULL`
		`);`

S

Q

L

Constraints	Description	Example

FOREIGN KEY	◦ References a record in another table.
	◦ Ensures referential integrity.
		`CREATE TABLE employees (`
		`id INTEGER PRIMARY KEY,`
		`name VARCHAR(50) NOT NULL,`
		`department_id INT,`
		`FOREIGN KEY (department_id)`
		`REFERENCES departments(id)`
		`);`

CHECK	◦ Validates column values against specific conditions.	`CREATE TABLE Employees (`
		`emp_id INT PRIMARY KEY,`
		`salary DECIMAL(10, 2) CHECK (salary >= 30000)`
		`);`

DEFAULT	◦ Sets a default value if no value is specified.	`CREATE TABLE Settings (`
		`setting_id INT PRIMARY KEY,`
		`theme VARCHAR(20) DEFAULT 'light'`
		`);`

SQL Command Types

S

Q

L

DDL

- CREATE
- DROP
- ALTER
- TRUNCATE
- COMMENT
- RENAME

DQL

- SELECT

DML

- INSERT
- DELETE
- UPDATE
- LOCK
- MERGE

DCL

- GRANT
- REVOKE

TCL

- COMMIT
- ROLLBACK
- SAVEPOINT
- SET TRANSACTION

S

Q

L

DATA DEFINITION LANGUAGE (DDL)

DDL commands are used to define and manage the structure of database objects (tables, indexes, views, etc.).

S

Q

L

COMMANDS	Description	Example
CREATE	. Creates a new table.	CREATE TABLE Employees (EmployeeID INT PRIMARY KEY, FirstName VARCHAR(50), LastName VARCHAR(50), Salary DECIMAL(10, 2));
ALTER	• Modifies an existing table (e.g., adding columns).	ALTER TABLE Employees ADD Department VARCHAR(50);
DROP	. Deletes a table.	DROP TABLE Employees;

S

Q

L

COMMANDS	Description	Example
TRUNCATE	• Removes all records from a table while preserving the table's structure.	TRUNCATE TABLE student;
COMMENT	• Adds descriptive comments to the data dictionary for a table.	COMMENT ON TABLE students_info IS 'Stores student information.';
RENAME	• Allows you to change the name of an existing database object (e.g., table, column, index).	RENAME TABLE student TO students_info;

S

Q

L

DATA QUERY LANGUAGE (DQL)

DQL commands allow you to retrieve data from the database.

S

Q

L

COMMANDS	Description	Example
		SELECT name, age FROM employees WHERE department = 'sales';
SELECT	Retrieves data from one or more tables.	SELECT * FROM customers;
		SELECT c.*, co.CountryName FROM customer c JOIN Country co ON c.CountryID = co.CountryID WHERE co.CountryName = 'Germany';

S

Q

L

DATA MANIPULATION LANGUAGE (DML)

DML commands manipulate data within tables.

S

Q

L

COMMANDS	Description	Example
INSERT	• Adds new records to a table.	INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary) VALUES (101, 'John', 'Doe', 60000);
UPDATE	• Modifies existing records.	UPDATE Employees SET Salary = 65000 WHERE EmployeeID = 101;
DELETE	• Removes records from a table.	DELETE FROM Employees WHERE EmployeeID = 101;

S

Q

L

COMMANDS	Description	Example
LOCK	• Ensures data consistency and prevents concurrent access issues.	UPDATE Employees SET Salary = Salary * 1.1 WHERE Department = 'Sales' WITH (TABLOCKX);
MERGE	• Combines `INSERT`, `UPDATE`, and `DELETE` operations based on a condition.	MERGE INTO target_table AS T USING source_table AS S ON T.id = S.id WHEN MATCHED THEN UPDATE SET T.description = S.description WHEN NOT MATCHED THEN INSERT (ID, description) VALUES (S.id, S.description);

S

Q

L

DATA CONTROL LANGUAGE (DCL)

DCL commands manage user access and permissions.

S

Q

L

COMMANDS	Description	Example
GRANT	• Grants specific privileges to users.	GRANT SELECT, INSERT ON Employees TO AnalystUser;
REVOKE	• Removes privileges from users.	REVOKE DELETE ON Employees FROM InternUser;

S

Q

L

TRANSACTION CONTROL LANGUAGE (DCL)

TCL commands manage transactions.

S

Q

L

COMMANDS	Description	Example
COMMIT	• Saves changes made during a transaction.	COMMIT;
ROLLBACK	• Reverts changes if an error occurs.	ROLLBACK;
SAVE POINT	• A point within a transaction where you can roll back to a specific state without affecting the entire transaction.	SAVEPOINT savepoint_name;
SET TRANSACTION	• Allows you to configure transaction properties.	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

S

Q

L

SQL OPERATORS

S

Q

L

COMPARISON OPERATORS

Comparison operators in SQL are used to compare two expressions and return a Boolean value (TRUE, FALSE, or NULL) based on the result of the comparison. These operators are commonly used in SQL queries to filter data based on specific conditions. Here are the main comparison operators in SQL,

1. Equal to (=) : Checks if two values are equal.
2. Not equal to (<> or !=) : Checks if two values are not equal.
3. Greater than (>) : Checks if the value on the left is greater than the value on the right.
4. Less than (<) : Checks if the value on the left is less than the value on the right.
5. Greater than or equal to (>=) : Checks if the value on the left is greater than or equal to the value on the right.
6. Less than or equal to (<=) : Checks if the value on the left is less than or equal to the value on the right.
7. BETWEEN : Checks if a value is within a range (inclusive).
8. IN : Checks if a value matches any value in a list.
9. LIKE : Checks if a value matches a pattern.
10. IS NULL : Checks if a value is NULL.
11. IS NOT NULL : Checks if a value is not NULL.

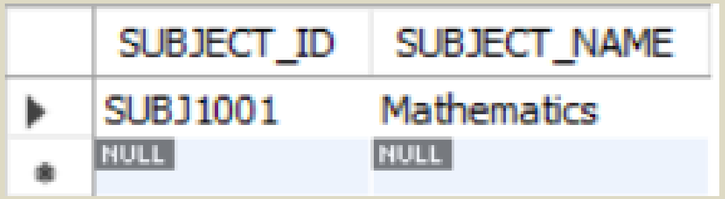
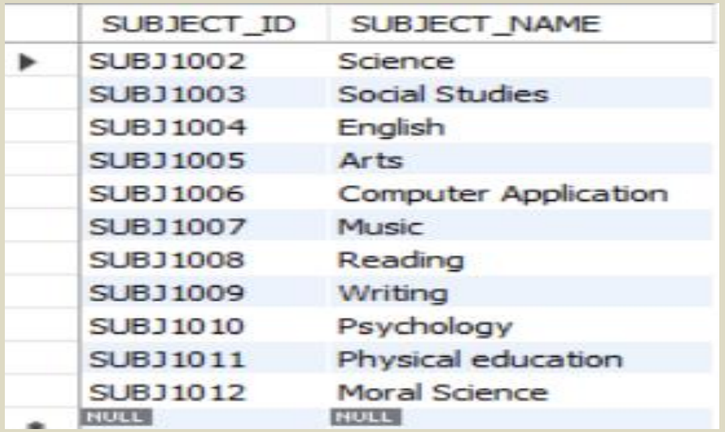
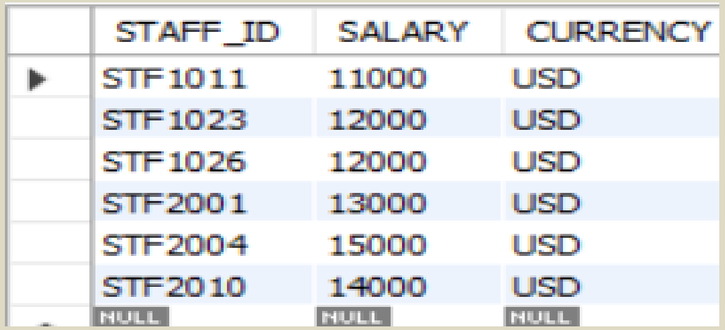
Comparison operators are essential tools for PROBLEMing and managing data in SQL databases, allowing for precise and flexible data retrieval based on specific criteria.

COMPARISON OPERATORS

S

Q

L

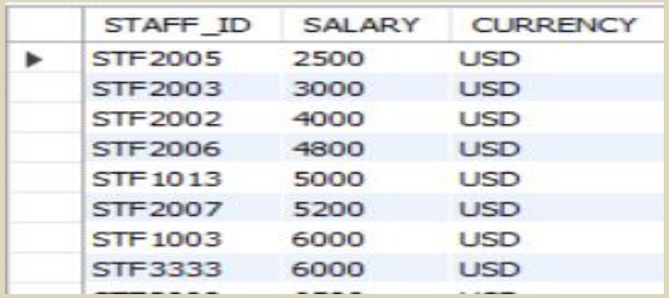
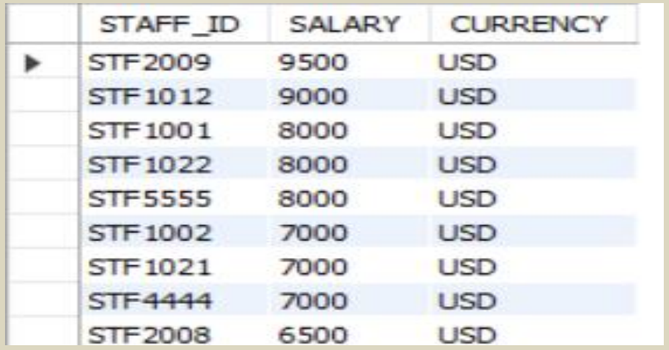
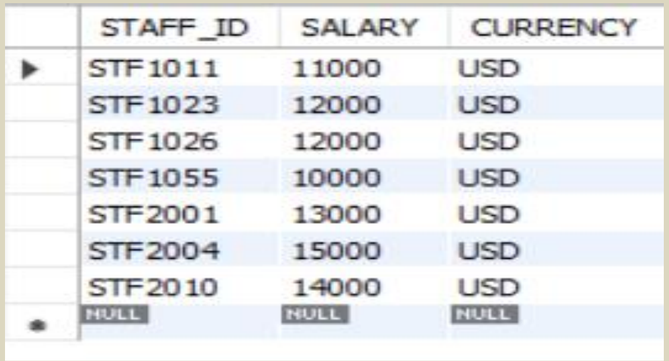
PROBLEM	QUERY	RESULT TABLE
1 . Fetch all records where subject name is Mathematics from SUBJECTS table	SELECT * FROM SUBJECTS WHERE SUBJECT_NAME = 'Mathematics';	
2. Fetch all records where subject name is not Mathematics from SUBJECTS table.	SELECT * FROM SUBJECTS WHERE SUBJECT_NAME <> 'Mathematics';	
3. Fetch all records where salary is greater than 10000 from STAFF_SALARY table.	SELECT * FROM STAFF_SALARY WHERE SALARY > 10000;	

COMPARISON OPERATORS

S

Q

L

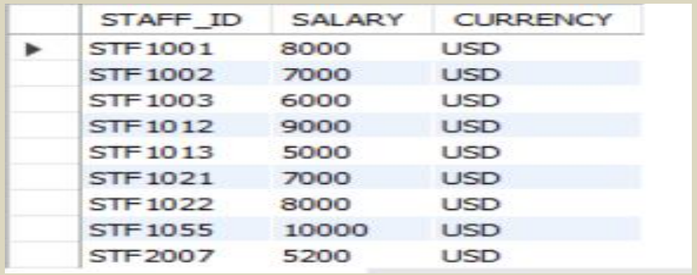
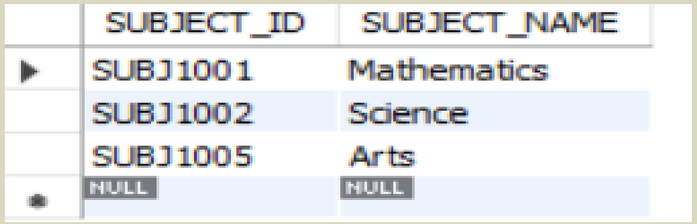
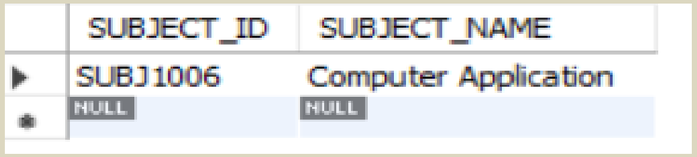
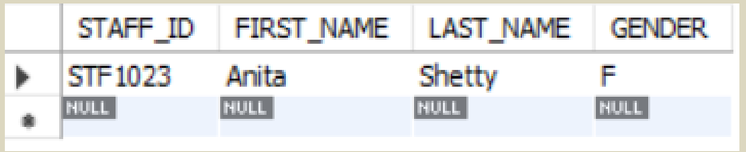
PROBLEM	QUERY	RESULT TABLE
4 . Fetch all records where salary is less than 10000 and the output is sorted in ascending order of salary from STAFF_SALARY table.	SELECT * FROM STAFF_SALARY WHERE SALARY < 10000 ORDER BY SALARY;	
5. Fetch all records where salary is less than 10000 and the output is sorted in descending order of salary from STAFF_SALARY table.	SELECT * FROM STAFF_SALARY WHERE SALARY < 10000 ORDER BY SALARY DESC;	
6. Fetch all records where salary is greater than or equal to 10000.	SELECT * FROM STAFF_SALARY WHERE SALARY >= 10000;	

COMPARISON OPERATORS

S

Q

L

PROBLEM	QUERY	RESULT TABLE
7. Fetch all records where salary is between 5000 and 10000 from STAFF_SALARY table.	SELECT * FROM STAFF_SALARY WHERE SALARY BETWEEN 5000 AND 10000;	
8. Fetch all records where subjects is either Mathematics, Science or Arts from SUBJECTS table.	SELECT * FROM SUBJECTS WHERE SUBJECT_NAME IN ('Mathematics', 'Science', 'Arts');	
9.Fetch records from SUBJECTS table where subject name has Computer as prefixed.	SELECT * FROM SUBJECTS WHERE SUBJECT_NAME LIKE 'Computer%';.	
10. Fetch STAFF_ID, FIRST_NAME, LAST_NAME , GENDER FROM STAFF where first name of staff starts with "A" AND last name starts with "S".	SELECT STAFF_ID, FIRST_NAME, LAST_NAME , GENDER FROM STAFF WHERE FIRST_NAME LIKE 'A%' AND LAST_NAME LIKE 'S%';	

STAFF_ID	SALARY	CURRENCY
STF1001	8000	USD
STF1002	7000	USD
STF1003	6000	USD
STF1012	9000	USD
STF1013	5000	USD
STF1021	7000	USD
STF1022	8000	USD
STF1055	10000	USD
STF2007	5200	USD

SUBJECT_ID	SUBJECT_NAME
SUBJ1001	Mathematics
SUBJ1002	Science
SUBJ1005	Arts
NULL	NULL

SUBJECT_ID	SUBJECT_NAME
SUBJ1006	Computer Application
NULL	NULL

STAFF_ID	FIRST_NAME	LAST_NAME	GENDER
STF1023	Anita	Shetty	F
NULL	NULL	NULL	NULL

S

Q

L

LOGICAL OPERATORS

Logical operators in SQL are used to combine multiple conditions in a SQL statement's WHERE clause. They help refine the selection criteria for queries, allowing more complex and specific data retrieval. The primary logical operators in SQL are

1. AND : Combines two or more conditions and returns true only if all the conditions are true.
2. OR : Combines two or more conditions and returns true if at least one of the conditions is true.
3. NOT : Reverses the result of a condition. It returns true if the condition is false, and vice versa.

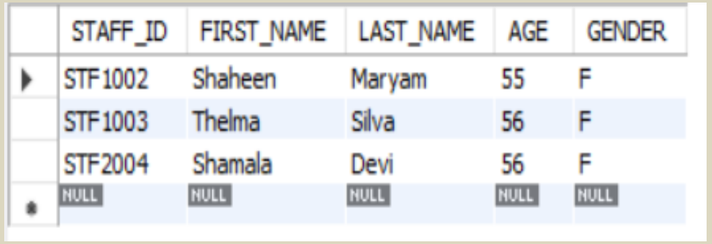
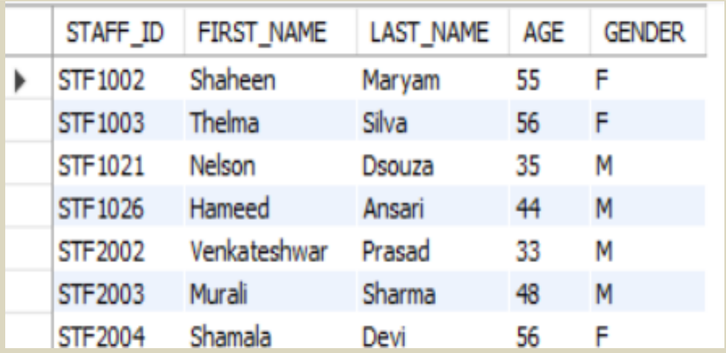
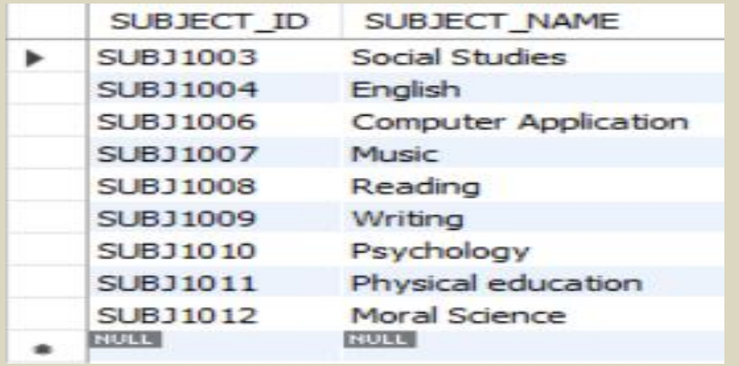
Logical operators are crucial for creating sophisticated queries in SQL, allowing you to filter and retrieve data based on multiple, specific conditions. They enable the development of powerful, detailed, and precise data retrieval strategies.

LOGICAL OPERATORS

S

Q

L

PROBLEM	QUERY	RESULT TABLE
1. Fetch STAFF_ID, FIRST_NAME, LAST_NAME ,AGE, GENDER from STAFF table where staff is female and is over 50 years of age.	SELECT STAFF_ID, FIRST_NAME, LAST_NAME ,AGE, GENDER FROM STAFF WHERE AGE > 50 **AND** GENDER = 'F';	
2. Fetch STAFF_ID, FIRST_NAME, LAST_NAME ,AGE, GENDER from STAFF table where staff is male or is over 50 years of age.	SELECT STAFF_ID, FIRST_NAME, LAST_NAME ,AGE, GENDER FROM STAFF WHERE AGE > 50 **OR** GENDER = 'M';	
3. Fetch all records where subjects is not Mathematics, Science or Arts from SUBJECTS table.	SELECT * FROM SUBJECTS WHERE SUBJECT_NAME **NOT** IN ('Mathematics', 'Science', 'Arts');	

S

Q

L

ARITHMETIC OPERATORS

Arithmetic operators in SQL are used to perform mathematical operations on numerical data. These operators allow you to execute calculations directly within your SQL queries, facilitating tasks like summing values, computing averages, and more. The primary arithmetic operators in SQL are :

1. Addition (+) : Adds two numerical values.
2. Subtraction (-) : Subtracts one numerical value from another.
3. Multiplication (*) : Multiplies two numerical values.
4. Division (/) : Divides one numerical value by another.
5. Modulus (%) : Returns the remainder of one numerical value divided by another.

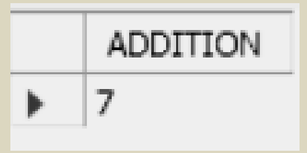
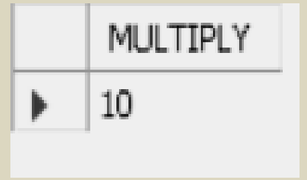
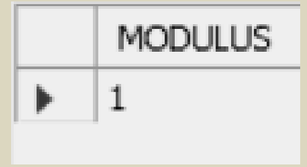
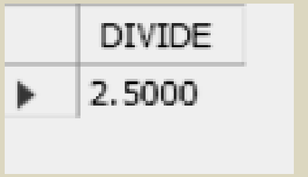
Arithmetic operators are essential tools in SQL for performing calculations directly within your queries, allowing for real-time data analysis and reporting. They enhance the flexibility and power of SQL in managing and manipulating numerical data.

S

Q

L

ARITHMETIC OPERATORS

PROBLEM	QUERY	RESULT TABLE
1. Add 2 numbers	SELECT (5+2) AS ADDITION;	
2. Multiply 2 numbers	SELECT (5*2) AS MULTIPLY;	
3. Divide 2 numbers and returns the remainder.	SELECT (5%2) AS MODULUS;	
4. Divides 2 numbers and returns whole number.	SELECT (5/2) AS DIVIDE;	

S

Q

L

CASE STATEMENT

The CASE statement in SQL is used to implement conditional logic within your queries. It allows you to return different values based on specific conditions, similar to an IF-THEN-ELSE structure in programming languages. This can be particularly useful for generating custom output, transforming data, or performing complex calculations within your SELECT statements.

Syntax of CASE Statement :

There are two forms of the CASE statement in SQL:

- Simple CASE Statement : This compares an expression to a set of simple expressions to determine the result.

```
CASE expression
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    ...
    ELSE default_result
END
```

- Searched CASE Statement : This evaluates a set of Boolean expressions to determine the result.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
END
```

S

Q

L

CASE STATEMENT

PROBLEM:

Fetch STAFF_ID, SALARY from STAFF_SALARY and find the range. in which salary falls. Condition for the range is:

1. If salary>=10000 then range is “High Salary”
2. If salary between 5000 and 10000 then range is “Average Salary”
3. If salary<5000 then range is “Too Low”

QUERY :

```
SELECT STAFF_ID, SALARY,
  CASE
    WHEN SALARY >= 10000 THEN 'High Salary'
    WHEN SALARY BETWEEN 5000 AND 10000 THEN 'Average Salary'
    WHEN SALARY < 5000 THEN 'Too Low’
  END AS SALARY_RANGE
FROM STAFF_SALARY;
```

RESULT TABLE:

	STAFF_ID	SALARY	SALARY_RANGE
▶	STF1001	8000	Average Salary
	STF1002	7000	Average Salary
	STF1003	6000	Average Salary
	STF1011	11000	High Salary
	STF1012	9000	Average Salary
	STF1013	5000	Average Salary
	STF1021	7000	Average Salary
	STF1022	8000	Average Salary
	STF1023	12000	High Salary
	STF1026	12000	High Salary
	STF1055	10000	High Salary
	STF2001	13000	High Salary
	STF2002	4000	Too Low
	STF2003	3000	Too Low

S

Q

L

DIFFERENT WAYS TO WRITE SQL JOINS

When writing SQL joins, there are two primary ways to express them:

1. Implicit joins

2. Explicit joins

Each method has its own style and use cases.

Key Differences:

1. Syntax and Structure:
   - Implicit Join : Combines tables in the FROM clause and specifies conditions in the WHERE clause.
   - Explicit Join : Uses specific JOIN keywords to combine tables and specifies join conditions in the ON clause.
2. Readability:
   - Implicit Join : Can become cluttered and harder to read with complex queries and multiple conditions.
   - Explicit Join : More readable and maintainable, as the join logic is clearly separated from filtering conditions.
3. Functionality:
   - Implicit Join : Limited to INNER JOIN logic. Achieving LEFT JOIN, RIGHT JOIN, or FULL JOIN functionality requires more complex and less intuitive queries.
   - Explicit Join : Supports various types of joins (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN), making it more versatile and powerful.

S

Q

L

IMPLICIT JOINS

Implicit joins use a comma-separated list of tables in the FROM clause and specify the join condition in the WHERE clause. This method was more common in older SQL syntax but is less preferred now due to its potential for ambiguity and decreased readability.

Example :

Fetch all the class name where Music is thought as a subject.

Query :

```
SELECT C.CLASS_NAME, S.SUBJECT_NAME
FROM CLASSES C, SUBJECTS S
WHERE C.SUBJECT_ID = S.SUBJECT_ID
AND S.SUBJECT_NAME = "Music";
```

Result table :

	CLASS_NAME	SUBJECT_NAME
▶	Grade 2	Music
	Grade 5	Music
	Grade 6	Music
	Grade 7	Music
	Grade 8	Music
	Grade 9	Music
	Grade 10	Music

S

Q

L

EXPLICIT JOINS

Explicit joins use the JOIN keyword along with ON to specify the joining condition. This approach is clearer and preferred for complex queries as it separates the join conditions from the filtering conditions.

Example :

Fetch all the class name where Music is thought as a subject.

Query :

```
SELECT C.CLASS_NAME, S.SUBJECT_NAME
FROM CLASSES C
JOIN SUBJECTS S
ON C.SUBJECT_ID = S.SUBJECT_ID
WHERE S.SUBJECT_NAME = "Music";
```

Result table :

	CLASS_NAME	SUBJECT_NAME
▶	Grade 2	Music
	Grade 5	Music
	Grade 6	Music
	Grade 7	Music
	Grade 8	Music
	Grade 9	Music
	Grade 10	Music

S

Q

L

TYPES OF EXPLICIT JOINS

1. INNER JOIN : Retrieves records that have matching values in both tables

2. LEFT JOIN : Retrieves all records from the left table and the matched records from the right table. Records from the left table without a match in the right table will have NULL values for the right table columns.

3. RIGHT JOIN : Retrieves all records from the right table and the matched records from the left table. Records from the right table without a match in the left table will have NULL values for the left table columns.

4. FULL JOIN : Retrieves records when there is a match in either left or right table. Records without a match in the other table will have NULL values for the columns of the table without a match.

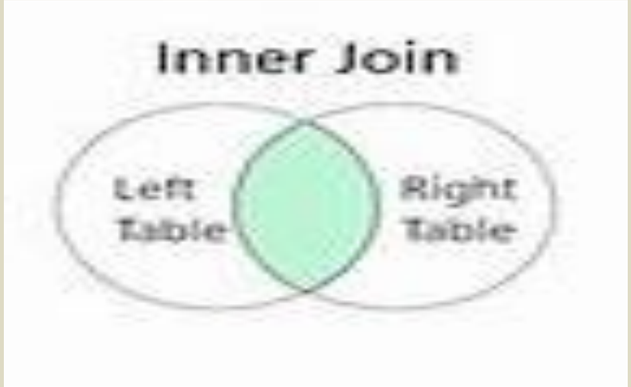
S

Q

L

INNER JOIN

- An INNER JOIN returns only the matching rows from both tables.
- It uses the ON keyword to specify the join condition.
- Inner Join can be represented as either "JOIN" or as "INNER JOIN". Both are correct and mean the same.
- Resulting rows = Intersection of rows from TableA and TableB.



Example :
Fetch the details of the those staff who have matching records in STAFF_SALARY table.

Query :

```
SELECT S.STAFF_ID,
        CONCAT(S.FIRST_NAME," ",S.LAST_NAME) AS FULL_NAME,
         SAL.SALARY
FROM STAFF S
JOIN STAFF_SALARY SAL
ON S.STAFF_ID = SAL.STAFF_ID;
```

STAFF_ID	FULL_NAME	SALARY
STF1001	Violet Mascarenhas	8000
STF1002	Shaheen Maryam	7000
STF1003	Thelma Silva	6000
STF1011	Eugine Rebello	11000
STF1012	Cynthia Sequeira	9000
STF1013	Veena Bhat	5000
STF1021	Nelson Dsouza	7000
STF1022	Mariat Rodrigues	8000
STF1023	Anita Shetty	12000
STF1026	Hameed Ansari	12000
STF1055	Leena Dsouza	10000

Here we fetched the staff_id and name of the staff from STAFF table and salary from the STAFF_SALARY table by joining the 2 tables using the matching rows

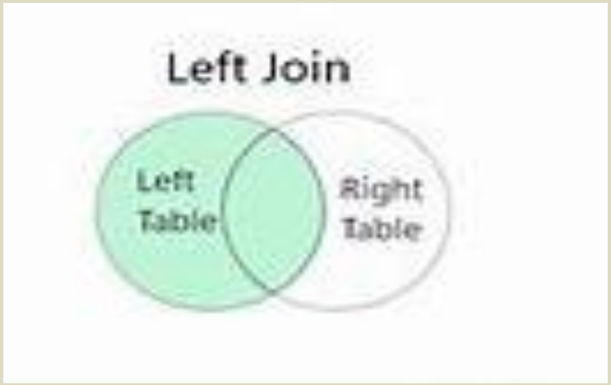
S

Q

L

LEFT JOIN (LEFT OUTER JOIN)

- A LEFT JOIN returns all rows from the left table and matching rows from the right table.
- If there's no match in the right table, it fills with null values.
- Resulting rows = All rows from TableA + Matching rows from TableB.



Example :
Fetch the details of the staff from STAFF table and their matching salary from STAFF_SALARY table.

Query :

```
SELECT S.STAFF_ID,
        CONCAT(S.FIRST_NAME," ",S.LAST_NAME) AS FULL_NAME,
    SAL.SALARY
FROM STAFF S
LEFT JOIN STAFF_SALARY SAL
ON S.STAFF_ID = SAL.STAFF_ID;
```

STAFF_ID	FULL_NAME	SALARY
STF2003	Murali Sharma	3000
STF2004	Shamala Devi	15000
STF2005	Rajesh Kumar	2500
STF2006	Mohammed Yasin	4800
STF2007	Raheem Khan	5200
STF2008	Adarsh Khandelwal	6500
STF2009	Premalatha Kumari	9500
STF2010	Eshal Maryam	14000
STF6006	Shobha Alva	NULL
STF7007	James Harden	NULL

Here for the staffs who don't have matching rows in STAFF_SALARY table was filled with NULL values.

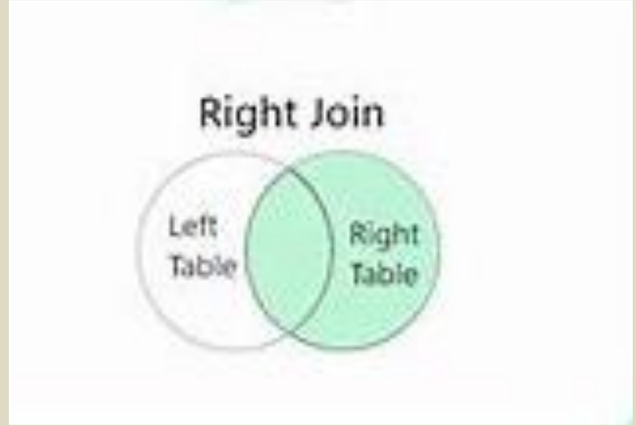
S

Q

L

RIGHT JOIN (RIGHT OUTER JOIN)

- A RIGHT JOIN returns all rows from the right table and matching rows from the left table.
- If there's no match in the left table, it fills with null values.
- Resulting rows = All rows from TableB + Matching rows from TableA.



Example :
Fetch the salary details of the staff from STAFF_SALARY table and their matching records from STAFF table.

Query :

```
SELECT S.STAFF_ID,
        CONCAT(S.FIRST_NAME," ",S.LAST_NAME) AS FULL_NAME,
    SAL.SALARY
FROM STAFF S
RIGHT JOIN STAFF_SALARY SAL
ON S.STAFF_ID = SAL.STAFF_ID;
```

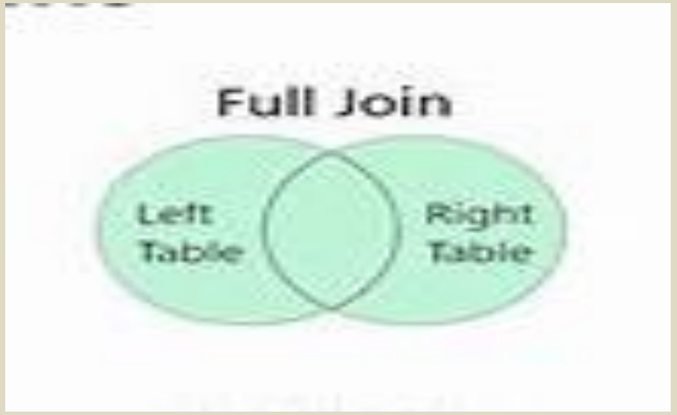
STAFF_ID	FULL_NAME	SALARY
STF2004	Shamala Devi	15000
STF2005	Rajesh Kumar	2500
STF2006	Mohammed Yasin	4800
STF2007	Raheem Khan	5200
STF2008	Adarsh Khandelwal	6500
STF2009	Premalatha Kumari	9500
STF2010	Eshal Maryam	14000
NULL	NULL	6000
NULL	NULL	7000
NULL	NULL	8000

Here the non matching records from the right STAFF table is filled with null values

FULL JOIN (FULL OUTER JOIN)

S Q L

- A FULL JOIN returns all rows from both tables (union of left and right joins).
- If there's no match, it fills with null values.
- Resulting rows = All rows from TableA + All rows from TableB.



Example :
Fetch all the rows from both the STAFF and STAFF_SALARY table.
Query :

```
SELECT S.STAFF_ID,
        CONCAT(S.FIRST_NAME," ",S.LAST_NAME) AS FULL_NAME,
    SAL.SALARY
FROM STAFF S
LEFT JOIN STAFF_SALARY SAL
ON S.STAFF_ID = SAL.STAFF_ID
UNION
SELECT S.STAFF_ID,
        CONCAT(S.FIRST_NAME," ",S.LAST_NAME) AS FULL_NAME,
    SAL.SALARY
FROM STAFF S
RIGHT JOIN STAFF_SALARY SAL
ON S.STAFF_ID = SAL.STAFF_ID;
```

STAFF_ID	FULL_NAME	SALARY
STF2006	Mohammed Yasin	4800
STF2007	Raheem Khan	5200
STF2008	Adarsh Khandelwal	6500
STF2009	Premalatha Kumari	9500
STF2010	Eshal Maryam	14000
STF6006	Shobha Alva	NULL
STF7007	James Harden	NULL
NULL	NULL	6000
NULL	NULL	7000
NULL	NULL	8000

Here we have all the records from both the tables and non matching records are filled with NULL values.

S

Q

L

UNION AND UNION ALL OPERATOR

UNION OPERATOR

- The UNION operator combines the results of two or more SELECT queries into a single result set.
- It removes duplicate rows from the combined result.

UNION ALL OPERATOR

- The UNION ALL operator also combines the results of two or more SELECT queries.
- Unlike UNION, it includes all rows, including duplicates.

Key Differences:

1. *Duplicate Handling*:
   - *`UNION`: Removes duplicates.*
   - *`UNION ALL`: Keeps duplicates.*

2. *Performance*:

   - *`UNION`: Slightly slower due to the need to remove duplicates.*
   - *`UNION ALL`: Faster as it simply concatenates the result sets.*

When to Use:

S

Q

L

UNION AND UNION ALL OPERATOR

Feature	UNION	UNION ALL
Functionality	Combines results from multiple SELECT queries and removes duplicates.	Combines results from multiple SELECT queries and keeps duplicates.
Duplicate Handling	Removes duplicate rows from the final result set.	Includes all rows from each SELECT query, keeping duplicates.
Performance	Slightly slower due to the overhead of removing duplicates.	Faster as it simply concatenates the result sets without removing duplicates.
Use Case	When you need a unique set of results and want to avoid duplicates. Ideal for merging lists of unique identifiers.	When duplicates are acceptable or necessary. Ideal for scenarios where performance is critical, and you know there are no duplicates or you need to keep them for analysis.
Practical Example	Using UNION: Suppose you have two tables, students and alumni, and you want to list all unique names from both tables. SELECT name FROM students UNION SELECT name FROM alumni;	Using UNION ALL: Using the same students and alumni tables, if you want to list all names, including duplicates: SELECT name FROM students UNION ALL SELECT name FROM alumni;
Key Points	Ensures that each row in the result is unique. Useful when merging lists with potentially overlapping entries.	Keeps all duplicates in the final result set. Useful for performance-critical scenarios or when duplicates are required for analysis.

S

Q

L

UNION OPERATOR

Example :
Fetch all staff who teaches grade 8, 9, 10 and also fetch all the non-teaching staff.

Query :

```
SELECT CONCAT(S.FIRST_NAME," ",S.LAST_NAME) AS FULL_NAME,
        S.STAFF_TYPE, S.GENDER, S.AGE
FROM CLASSES C
JOIN STAFF S
ON C.TEACHER_ID = S.STAFF_ID
WHERE C.CLASS_NAME IN ("Grade 8","Grade 9","Grade 10")
AND S.STAFF_TYPE="Teaching“

UNION

SELECT  CONCAT(FIRST_NAME," ",LAST_NAME) AS FULL_NAME,
        STAFF_TYPE, GENDER,AGE
FROM STAFF
WHERE STAFF_TYPE="Non-Teaching";
```

Here the query returned 16 rows removing the duplicate values from the table

Syntax:

```
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

Result Table :

FULL_NAME	STAFF_TYPE	GENDER	AGE
Thelma Silva	Teaching	F	56
Nelson Dsou	aching	M	35
Mariat Rodrigues	Teaching	F	40
Anita Shetty	Teaching	F	32
Hameed Ansari	Teaching	M	44
Leena Dsouza	Teaching	F	28
Laxmi Narayana	Non-Teaching	F	40
Venkateshwar Prasad	Non-Teaching	M	33

S

Q

L

UNION ALL OPERATOR

Example :
Fetch all staff who teaches grade 8, 9, 10 and also fetch all the non-teaching staff.

Query :

```
SELECT CONCAT(S.FIRST_NAME," ",S.LAST_NAME) AS FULL_NAME,
        S.STAFF_TYPE, S.GENDER, S.AGE
FROM CLASSES C
JOIN STAFF S
ON C.TEACHER_ID = S.STAFF_ID
WHERE C.CLASS_NAME IN ("Grade 8","Grade 9","Grade 10")
AND S.STAFF_TYPE="Teaching“

UNION ALL

SELECT  CONCAT(FIRST_NAME," ",LAST_NAME) AS FULL_NAME,
        STAFF_TYPE, GENDER,AGE
FROM STAFF
WHERE STAFF_TYPE="Non-Teaching";
```

Here the query returned 28 rows including the duplicate values from both the table

Syntax:

```
SELECT column1, column2, ...
FROM table1
UNION ALL
SELECT column1, column2, ...
FROM table2;
```

Result Table :

FULL_NAME	STAFF_TYPE	GENDER	AGE
Thelma Silva	Teaching	F	56
Thelma Silva	Teaching	F	56
Thelma Silva	Teaching	F	56
Nelson Dsouza	Teaching	M	35
Nelson Dsouza	Teaching	M	35
Nelson Dsouza	Teaching	M	35
Mariat Rodrigues	Teaching	F	40
Mariat Rodrigues	Teaching	F	40
Mariat Rodrigues	Teaching	F	40

S

Q

L

GROUP BY CLAUSE

- The GROUP BY clause groups rows that have the same values in specified columns into summary rows. It’s typically used with aggregate functions like COUNT, SUM, AVG, MAX, and MIN to perform operations on each group.

Syntax:

```
SELECT column1,
    AGGREGATE_FUNCTION
    (column2)
FROM table_name
GROUP BY column1;
```

Example :
Count the no of students in each class.

Query :

```
SELECT CLASS_ID, COUNT(CLASS_ID) AS NO_OF_STUDENTS
FROM CLASSES
GROUP BY CLASS_ID;
```

Here the query grouped same class_id together and counted the total no.of students.

Result Table :

CLASS_ID	NO_OF_STUDENTS
CLS1001	3
CLS1002	4
CLS1003	4
CLS1004	4
CLS1005	6
CLS1006	6
CLS1007	6
CLS1008	6
CLS1009	6
CLS1010	6

S

Q

L

HAVING CLAUSE

The HAVING clause filters the results of GROUP BY based on a specified condition. Unlike the WHERE clause, which filters rows before grouping, HAVING filters groups after the aggregation.

```
SELECT column1,
    AGGREGATE_FUNCTION
    (column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Example :
Return only those records where there are more than 100 students in each class.

Query :

```
SELECT CLASS_ID,COUNT(1) AS NO_OF_STUDENTS
FROM STUDENT_CLASSES
GROUP BY CLASS_ID
HAVING  NO_OF_STUDENTS >100;
```

Result Table :

CLASS_ID	NO_OF_STUDENTS
CLS1002	120
CLS1007	125

Here the query returned only the details of those class where no.of students >100.

S

Q

L

AGGREGATE FUNCTIONS

Aggregate functions perform a calculation on a set of values and return a single value. Common aggregate functions include:

- COUNT() : Counts the number of rows in a group.
- SUM() : Calculates the sum of a numeric column.
- AVG() : Computes the average of a numeric column.
- MIN() : Finds the minimum value in a column.
- MAX() : Retrieves the maximum value in a column.

```
SELECT column1,
    AGGREGATE_FUNCTION
    (column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Example :
FIND THE AVERAGE SALARY OF STAFFS.

Query :

```
SELECT S.STAFF_TYPE, ROUND(AVG(SAL.SALARY),2) AS AVG_SALARY
FROM STAFF S
JOIN STAFF_SALARY SAL
ON S.STAFF_ID = SAL.STAFF_ID
GROUP BY S.STAFF_TYPE;
```

Result Table :

STAFF_TYPE	AVG_SALARY
Teaching	8636.36
Non-Teaching	7750

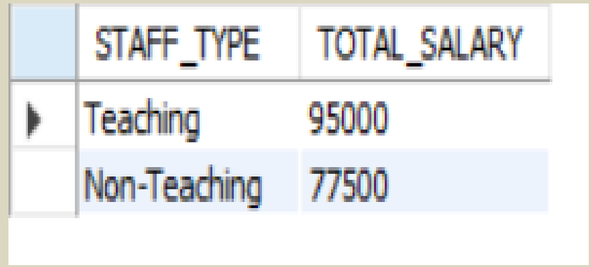
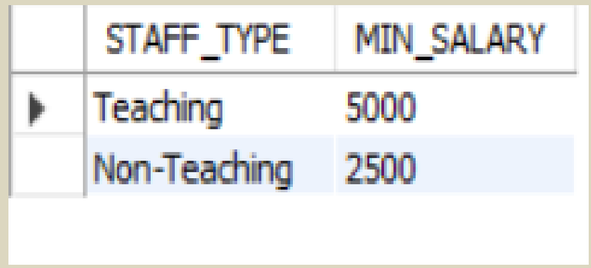
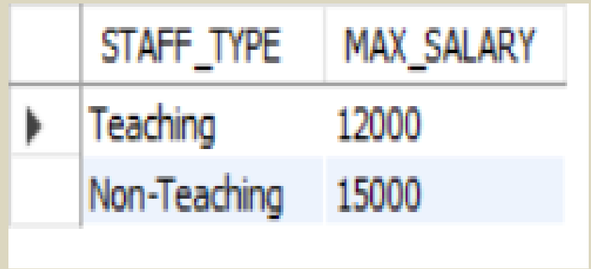
Here the query returned the average salary of staffs based on the staff type.

AGGREGATE FUNCTIONS

S

Q

L

PROBLEM	QUERY	RESULT TABLE
Calculates the total salary paid to each staff type.	SELECT S.STAFF_TYPE, SUM(SAL.SALARY) AS TOTAL_SALARY FROM STAFF S J OIN STAFF_SALARY SAL ON S.STAFF_ID = SAL.STAFF_ID GROUP BY S.STAFF_TYPE;	
Calculate the minimum salary paid to each staff type.	SELECT S.STAFF_TYPE, MIN(SAL.SALARY) AS MIN_SALARY FROM STAFF S JOIN STAFF_SALARY SAL ON S.STAFF_ID = SAL.STAFF_ID GROUP BY S.STAFF_TYPE;	
Calculate the maximum salary paid to each staff type.	SELECT S.STAFF_TYPE, MAX(SAL.SALARY) AS MAX_SALARY FROM STAFF S JOIN STAFF_SALARY SAL ON S.STAFF_ID = SAL.STAFF_ID GROUP BY S.STAFF_TYPE;	

S

Q

L

SQL SUBQUERIES

SUBQUERY

A subquery, also known as an inner query or nested query, is a query embedded within another SQL query. The subquery executes first, and its result is used by the outer query.

Subqueries can be used in various parts of an SQL statement, including the `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses.

EXAMPLE :

Find the employees who's salary is more than the average salary earned by all employees.

```
SELECT * FROM EMPLOYEE
WHERE SALARY> (SELECT AVG(salary) AS
                    avg_salary
               FROM employee);
```

EMP_ID	EMP_NAME	DEPT_NAME	SALARY
104	Dorvin	Finance	6500
107	Preet	HR	7000
109	Sanjay	IT	6500
110	Vasudha	IT	7000
111	Melinda	IT	8000
112	Komal	IT	10000
116	Satya	Finance	6500
119	Cory	HR	8000

- Here the subquery calculates the average salary.
- The outer query selects the employee details from employees where the salary is greater than the average salary.

DIFFERENT TYPES OF SUBQUERIES

Subqueries can be classified into different types based on their usage and return values:

- Single-Row Subquery : Returns a single row and column.
- Multi-Row Subquery : Returns multiple rows and columns.
- Correlated Subquery : References columns from the outer query and is executed once for each row processed by the outer query.
- Nested Subquery : A subquery within another subquery.

SCALAR SUBQUERY

- A scalar subquery returns a single value (one row and one column).
- Commonly used in the WHERE clause to filter results based on a specific condition.

EXAMPLE :

Find the employees who's salary is more than the average salary earned by all employees.

```
SELECT *FROM EMPLOYEE E
JOIN (SELECT AVG(salary) AS avg_salary
        FROM employee) AVERAGE
ON E.SALARY > AVERAGE.avg_salary;
```

EMP_ID	EMP_NAME	DEPT_NAME	SALARY	avg_salary
104	Dorvin	Finance	6500	5791.6667
107	Preet	HR	7000	5791.6667
109	Sanjay	IT	6500	5791.6667
110	Vasudha	IT	7000	5791.6667
111	Melinda	IT	8000	5791.6667
112	Komal	IT	10000	5791.6667
116	Satya	Finance	6500	5791.6667
119	Cory	HR	8000	5791.6667

- First, the subquery is executed. This subquery calculates the average salary of all employees in the EMPLOYEE table
- After calculating the average salary, the main query performs a join between the EMPLOYEE table (E) and the result of the subquery (AVERAGE).

  The join condition ON E.SALARY > AVERAGE.avg_salary ensures that only employees whose salary is greater than the average salary are selected.

MULTIPLE ROW SINGLE COLUMN SUB-QUERIES

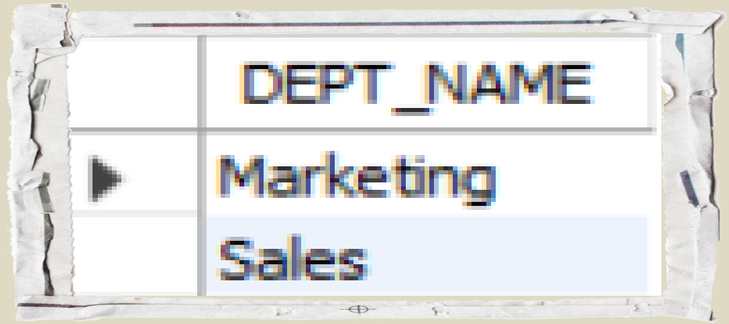
- A multiple-row subquery returns a set of rows.
- Useful when you need to compare values across multiple rows.

EXAMPLE :

-- Find department who do not have any employees

.

```
SELECT DEPT_NAME
FROM DEPARTMENT
WHERE DEPT_NAME NOT IN (SELECT  DISTINCT
                                DEPT_NAME
                        FROM EMPLOYEE);
```



The output table has a single column with multiple rows

1. First the subquery will find out the departments where the employees are present.
2. Then the outer query will filter out the results of above query from the department table.

MULTIPLE ROW MULTIPLE COLUMN SUB-QUERIES

- Often used in complex queries where you need to retrieve additional columns from a subquery.

EXAMPLE:

-- Find the employees who earn the highest salary in each department.

```
SELECT *
FROM EMPLOYEE
WHERE (DEPT_NAME,SALARY) IN (SELECT
                    DEPT_NAME,
                    MAX(SALARY) AS MAX_SALARY
                    FROM EMPLOYEE
                    GROUP BY DEPT_NAME);
```

EMP_ID	EMP_NAME	DEPT_NAME	SALARY
104	Dorvin	Finance	6500
116	Satya	Finance	6500
119	Cory	HR	8000
120	Monica	Admin	5000
124	Dheeraj	IT	11000
NULL	NULL	NULL	NULL

The output table has multiple columns with multiple rows

1. **First the subquery will return the highest salary in each department.**
2. **Then the outer query will filter the employees based on above result.**

CORRELATED SUBQUERIES

- A correlated subquery references columns from the outer query.
- It executes once for each row in the outer query.

EXAMPLE :

-- Find the employees in each department who earn more than the average salary in that department. .

```
SELECT * FROM EMPLOYEE E1
WHERE SALARY > (SELECT AVG(SALARY)
        FROM EMPLOYEE E2
        WHERE E1.DEPT_NAME = E2.DEPT_NAME);
```

EMP_ID	EMP_NAME	DEPT_NAME	SALARY
101	Mohan	Admin	4000
104	Dorvin	Finance	6500
107	Preet	HR	7000
108	Maryam	Admin	4000
111	Melinda	IT	8000
112	Komal	IT	10000
116	Satya	Finance	6500
119	Cory	HR	8000

1. First the subquery will return average salary of every department
2. Then the outer query filter data from employee tables based on avg salary from above result.

NESTED SUBQUERIES

- A nested subquery contains another subquery within it.
- These can be scalar, multiple-row, or multiple-column subqueries.

EXAMPLE :

-- Find stores who's sales where better than the average sales accross all stores .

```
SELECT *
FROM (SELECT store_name, SUM(price) AS total_sales
      FROM sales
      GROUP BY store_name) sales
      JOIN (SELECT AVG(total.total_sales) AS avg_sales
            FROM (SELECT store_name, SUM(price) AS total_sales
                  FROM sales
                  GROUP BY store_name) total) average
      ON sales.total_sales > average.avg_sales;
```

store_name	total_sales	avg_sales
Apple Store 1	7500	6050.0000
Apple Store 3	9700	6050.0000

1. First the total subquery will be calculated to find the total sales of all stores
2. Then the average query will return the average sales of all stores based on above result.
3. Then the result from both the above queries will be compared.

CLAUSES WHERE SUBQUERY CAN BE USED

Subqueries in SQL can be used in various clauses to perform complex operations. Here's a brief overview of where subqueries can be used:

- SELECT Clause : To return values that are included as part of the result set.
- FROM Clause : To create a temporary or derived table that the outer query can use.
- WHERE Clause : To filter records based on conditions evaluated by the subquery.
- HAVING Clause : To filter groups based on aggregate values calculated by the subquery.

In the previous examples we have seen how to use subquery with FROM clause and WHERE clause.

SUBQUERY IN SELECT CLAUSE

Subqueries which return only 1 row and 1 column (scalar or correlated) is allowed with SELECT clause.

EXAMPLE :

Fetch all employee details and add remarks to those employees who earn more than the average pay.

```
SELECT E.*,
        CASE WHEN E.SALARY > (SELECT
                              AVG(SALARY)
                              FROM EMPLOYEE)
             THEN 'Above average salary'
             ELSE NULL
        END REMARKS
FROM EMPLOYEE E;
```

EMP_ID	EMP_NAME	DEPT_NAME	SALARY	REMARKS
102	Rajkumar	HR	3000	NULL
103	Akbar	IT	4000	NULL
104	Dorvin	Finance	6500	Above average salary
105	Rohit	HR	3000	NULL
106	Rajesh	Finance	5000	NULL
107	Preet	HR	7000	Above average salary
108	Maryam	Admin	4000	NULL

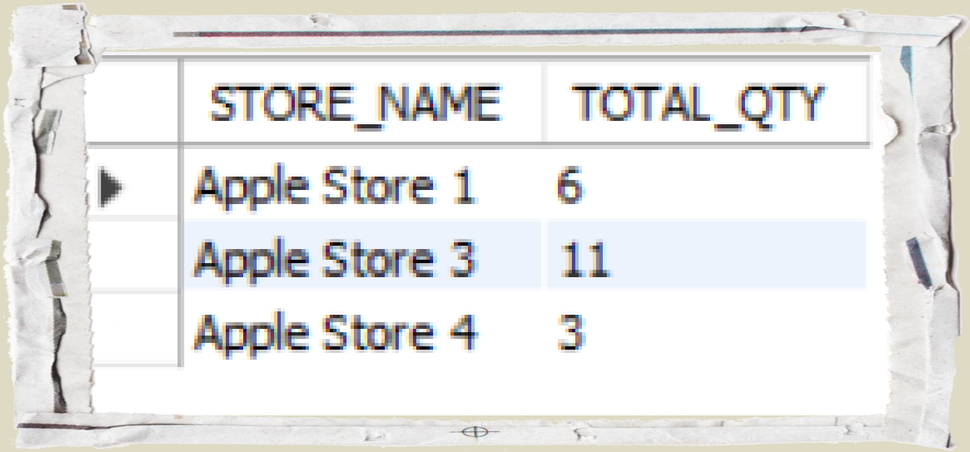
- The subquery calculates the average salary of all employees in the EMPLOYEE table.
- For each row in the EMPLOYEE table, the main query compares the employee's salary with the average salary obtained from the subquery.
- If the salary is greater than the average, 'Above average salary' is assigned to the REMARKS column otherwise, NULL is assigned to the REMARKS column.

SUBQUERY IN HAVING CLAUSE

EXAMPLE :

Find the stores who have sold more units than the average units sold by all stores.

```
SELECT STORE_NAME,
        SUM(QUANTITY) AS TOTAL_QTY
FROM SALES
GROUP BY STORE_NAME
HAVING TOTAL_QTY> (SELECT AVG(QUANTITY)
                   FROM SALES);
```



STORE_NAME	TOTAL_QTY
Apple Store 1	6
Apple Store 3	11
Apple Store 4	3

- The query groups the sales data by STORE_NAME and calculates the total quantity sold for each store.
- The subquery calculates the average quantity sold across all records in the SALES table.
- The HAVING clause filters the groups to include only those where the total quantity sold is greater than the average quantity sold across all sales, as computed by the subquery.

SQL COMMANDS WHICH ALLOW A SUBQUERY

SQL commands that allow subqueries enable complex data retrieval and manipulation by incorporating the results of one query within another. Here's an overview of the primary SQL commands that can incorporate subqueries

- INSERT COMMAND : To insert rows into a table based on the result of the subquery.
- UPDATE COMMAND : To update rows in a table based on conditions evaluated by the subquery.
- DELETE COMMAND : To delete rows from a table based on conditions evaluated by the subquery.

SUBQUERY WITH INSERT STATEMENT

- Subqueries can be used to insert rows based on the result of another query.

EXAMPLE :

-- Insert data to employee history table. Make sure not insert duplicate records

```
INSERT INTO EMPLOYEE_HISTORY
SELECT E.EMP_ID, E.EMP_NAME, D.DEPT_NAME,
        E.SALARY, D.LOCATION
FROM EMPLOYEE E
JOIN DEPARTMENT D
ON E.DEPT_NAME = D.DEPT_NAME
WHERE NOT EXISTS (SELECT 1
                  FROM EMPLOYEE_HISTORY EH
                  WHERE EH.EMP_ID = E.EMP_ID);
```

```
209 •  SELECT * FROM EMPLOYEE_HISTORY;
```

Result Grid | Filter Rows: | Edit:

emp_id	emp_name	dept_name	salary	location
101	Mohan	Admin	4000	Bangalore
102	Rajkumar	HR	3000	Bangalore
103	Akbar	IT	4000	Bangalore
104	Dorvin	Finance	6500	Mumbai
105	Rohit	HR	3000	Bangalore

- The main query joins the EMPLOYEE and DEPARTMENT tables based on the DEPT_NAME column.
- The NOT EXISTS subquery checks if an employee with the same EMP_ID already exists in the EMPLOYEE_HISTORY table. If the employee exists, the condition is false, and that employee is excluded from the insert operation.
- Only those rows from the EMPLOYEE table that do not have a corresponding EMP_ID in the EMPLOYEE_HISTORY table are inserted.

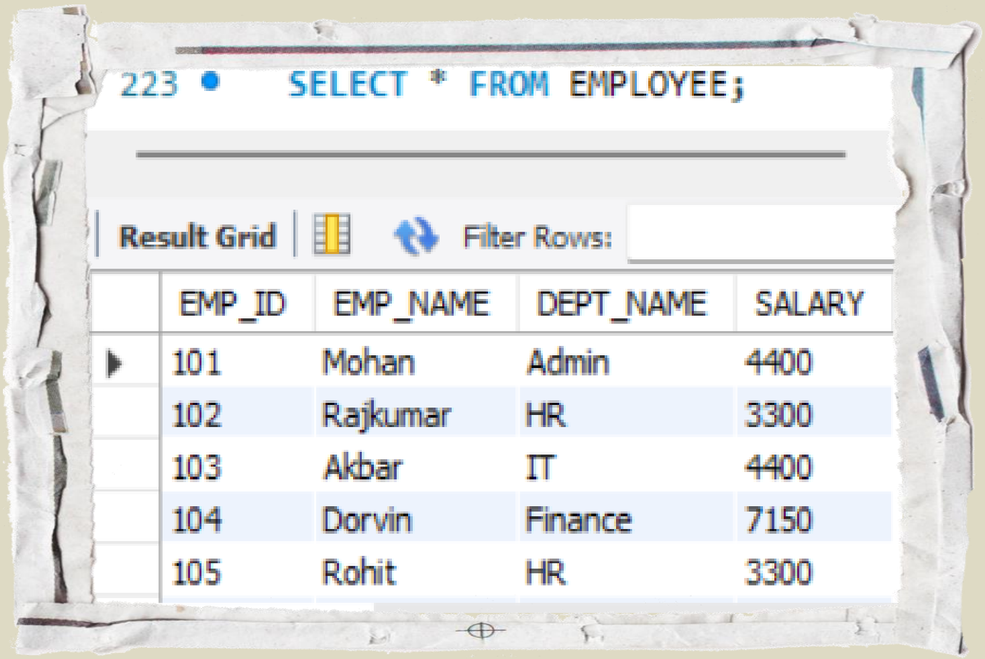
SUBQUERY WITH UPDATE STATEMENT

- Subqueries can be used to update rows based on conditions evaluated by another query.

EXAMPLE :

-- Give 10% increment to all employees in Bangalore location based on the maximum salary earned by an employee in each dept. Only consider employees in employee_history table.-

```
UPDATE EMPLOYEE E
SET SALARY = (SELECT MAX(SALARY)+ (MAX(SALARY)* 0.1)
              FROM EMPLOYEE_HISTORY EH
                   WHERE EH.EMP_ID = E.EMP_ID)
WHERE DEPT_NAME IN (SELECT DEPT_NAME
                    FROM DEPARTMENT)
AND E.EMP_ID IN (SELECT EMP_ID
                 FROM EMPLOYEE_HISTORY);
```



EMP_ID	EMP_NAME	DEPT_NAME	SALARY
101	Mohan	Admin	4400
102	Rajkumar	HR	3300
103	Akbar	IT	4400
104	Dorvin	Finance	7150
105	Rohit	HR	3300

- For each employee in the EMPLOYEE table, the subquery calculates the new salary.
- Where clause ensures that the employees belong to valid departments and employees exist in the EMPLOYEE_HISTORY table.
- The SALARY of each employee meeting the criteria is updated to the calculated new salary.

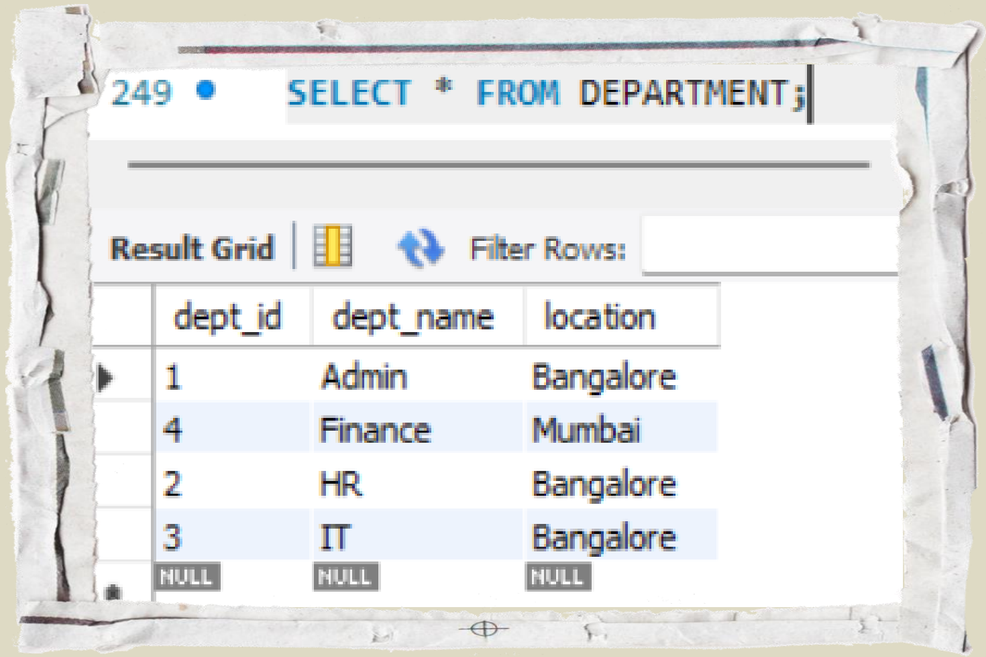
SUBQUERY WITH DELETE STATEMENT

- Subqueries can be used to delete rows based on conditions evaluated by another query.

EXAMPLE :

Delete all departments who do not have any employees.

```
DELETE FROM department
WHERE dept_name IN
        ( SELECT dept_name FROM
                ( SELECT dept_name FROM department d
                  WHERE NOT EXISTS
                        ( SELECT 1 FROM employee e
                          WHERE e.dept_name = d.dept_name
                        )) AS derived_table);
```



- The inner subquery identifies departments without employees
- The outer query then deletes all rows from the department table where dept_name matches any of the department names in the derived table.

S

Q

L

COMMON TABLE EXPRESSION

What is a CTE?

The SQL WITH clause, also known as Common Table Expressions (CTEs), allows you to define temporary result sets that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. It helps in organizing and structuring complex queries, making them more readable and maintainable.

Key Features of CTEs :

Temporary Scope : CTEs exist only for the duration of the query in which they are defined.
Readable Syntax : Using the WITH clause helps to simplify and clarify complex query logic.
Support for Recursion : Recursive CTEs are useful for hierarchical or recursive data structures.
Reusable Within a Query : You can reference a CTE multiple times within the same query.

Advantages of Using CTEs

- Improved Readability : Breaking down complex queries into simpler parts makes them easier to understand and maintain.
- Modularity : Each CTE can be written, debugged, and tested independently.
- Reusability : Avoid redundancy by referencing the same CTE multiple times in the main query.
- Hierarchical Data Handling : Recursive CTEs provide an elegant solution for querying hierarchical data.

When to Use CTEs

- Simplifying Complex Queries : Break down complicated queries into smaller, more manageable parts.
- Reusing Subqueries : Define a subquery once and reuse it multiple times within the main query.
- Handling Hierarchical Data : Recursive CTEs are ideal for working with hierarchical data like organizational charts.
- Staging Data : Use CTEs to store intermediate results temporarily during multi-step transformations.
- Improving Readability and Maintainability : Enhance the structure and clarity of your queries.

Syntax of a CTE

```
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT column1, column2
FROM cte_name
WHERE another_condition;
```

Fetch employees who earn more than average salary of all employees

```
16
17    /* STEP 1 : FIND THE AVERAGE SALARY OF ALL EMPLOYEES
18       STEP 2: FIND WHETHER THE SALARY OF AN EMPLOYEE > AVERAGE SALARY OF ALL EMPLOYEES. */
19
20    WITH AVERAGE AS (
21        SELECT AVG(SALARY) AS AVG_SALARY
22        FROM EMPLOYEE)
23    SELECT E.*
24    FROM EMPLOYEE E, AVERAGE AV
25    WHERE E.SALARY > AV.AVG_SALARY;
26
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

employee_ID	employee_NAME	SALARY
104	Carol	70000
105	Alice	80000
106	Jimmy	90000

Find stores whose sales were better than the average sales across all the stores

```
50    /*  STEP 1: FIND THE TOTAL SALES OF ALL STORES
51        STEP 2: FIND THE AVERAGE SALES WITH RESPECT TO ALL STORES.
52        STEP 3: FIND THE STORES WHERE TOTAL SALES > AVERAGE SALES. */
53
54    WITH TOTAL_SALES AS (
55            SELECT STORE_NAME,SUM(COST)AS TOTAL
56            FROM SALES
57            GROUP BY STORE_NAME),
58        AVERAGE_SALES AS (
59            SELECT AVG(TOTAL) AS AVG_SALES
60            FROM TOTAL_SALES)
61    SELECT *
62    FROM TOTAL_SALES TS
63    JOIN AVERAGE_SALES AV
64    ON TS.TOTAL > AV.AVG_SALES;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

STORE_NAME	TOTAL	AVG_SALES
Apple Originals 3	5380	3290.0000
Apple Originals 4	3500	3290.0000

S

Q

L

WINDOW FUNCTIONS

AGGREGATE FUNCTIONS

FIND THE MAXIMUM SALARY IN EACH DEPARTMENT

WITHOUT WINDOW FUNCTIONS

```
42 •  SELECT DEPT_NAME , MAX(SALARY) AS MAX_SALARY
43    FROM EMPLOYEE
44    GROUP BY DEPT_NAME;
45
```

Result Grid | Filter Rows: | Export: | Wrap Cel

DEPT_NAME	MAX_SALARY
Admin	5000
HR	8000
IT	11000
Finance	6500

WITH WINDOW FUNCTIONS

```
46 •  SELECT E.*,
47    MAX(SALARY) OVER (PARTITION BY DEPT_NAME) AS MAX_SALARY
48    FROM EMPLOYEE E;
49
50
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

emp_ID	emp_NAME	DEPT_NAME	SALARY	MAX_SALARY
101	Mohan	Admin	4000	5000
108	Maryam	Admin	4000	5000
113	Gautham	Admin	2000	5000
120	Monica	Admin	5000	5000
104	Dorvin	Finance	6500	6500
106	Rajesh	Finance	5000	6500
116	Satya	Finance	6500	6500

QUERY WITHOUT WINDOW FUNCTIONS:

Purpose : This query calculates the maximum salary for each department and returns one row per department.

Usage : It is typically used when you need a summary of data grouped by a specific column.

QUERY WITH WINDOW FUNCTIONS:

Purpose : This query calculates the maximum salary for each department but returns all rows from the EMPLOYEE table along with the maximum salary as an additional column.

Usage : It is used when you need detailed row-level data along with an aggregated value computed over a specified partition.

RANKING FUNCTIONS

```sql
/* DIFFERENCE BETWEEN ROW_NUMBER(), RANK() AND DENSE_RANK() WINDOW FUNCTIONS WITH EXAMPLE */

SELECT E.*,
    ROW_NUMBER() OVER (PARTITION BY DEPT_NAME ORDER BY SALARY) AS ROW_NO,
    RANK() OVER (PARTITION BY DEPT_NAME ORDER BY SALARY) AS RANK_NO,
    DENSE_RANK() OVER (PARTITION BY DEPT_NAME ORDER BY SALARY) AS DENSE_RANK_NO
FROM EMPLOYEE E;

/* Explanation of Ranking Functions in the above Example :
    ROW_NUMBER()    : Generates a unique identifier for each row within the department, regardless of ties in the salary.
    RANK()          : Generates a ranking number within each department, but skips numbers if there are ties. For example, if two employees share the same salary,
                      they will receive the same rank, and the next rank will skip a number.
    DENSE_RANK()    : Similar to RANK(), but does not skip numbers after ties. Tied employees receive the same rank, and the next rank continues sequentially.
Practical Use Cases :
    ROW_NUMBER()    : Useful for uniquely identifying rows within a partition, such as when paginating results.
    RANK()          : Useful for generating a ranking system where gaps indicate tied rankings, which can highlight the presence of ties within groups.
    DENSE_RANK()        : Useful for scenarios where a continuous sequence of ranks is needed despite ties, such as in scoring systems or hierarchical data analysis. */
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

emp_ID	emp_NAME	DEPT_NAME	SALARY	ROW_NO	RANK_NO	DENSE_RANK_NO
113	Gautham	Admin	2000	1	1	1
101	Mohan	Admin	4000	2	2	2
108	Maryam	Admin	4000	3	2	2
120	Monica	Admin	5000	4	4	3
106	Rajesh	Finance	5000	1	1	1
118	Tejaswi	Finance	5500	2	2	2
104	Dorvin	Finance	6500	3	3	3

RANKING FUNCTIONS

```sql
-- NTILE
-- WRITE A QUERY TO SEGREGATE ALL THE EXPENSIVE PHONES, MID RANGE PHONES AND CHEAPER PHONES

SELECT X.PRODUCT_NAME, PRICE,
    CASE WHEN X.BUCKETS = 1 THEN "EXPENSIVE PHONES"
         WHEN X.BUCKETS = 2 THEN "MID RANGE PHONES"
         WHEN X.BUCKETS = 3 THEN "CHEAPER PHONE"
         END AS PHONE_CATEGORY
FROM
    (SELECT *,
        NTILE(3) OVER (ORDER BY PRICE DESC) AS BUCKETS
        FROM PRODUCT
        WHERE PRODUCT_CATEGORY = "PHONE") X;

/* 1. The subquery first filters the products to include only those that belong to the "PHONE" category.
      Then, it orders these products by PRICE in descending order.
   2. The NTILE(3) function is applied to divide these ordered products into three buckets. The most expensive phones will be in the first bucket (BUCKETS = 1),
      the mid-range priced phones in the second bucket (BUCKETS = 2), and the cheaper phones in the third bucket (BUCKETS = 3).
   3. The outer query then uses a CASE statement to categorize each phone into "EXPENSIVE PHONES", "MID RANGE PHONES", or "CHEAPER PHONE" based on the value of BUCKETS.
*/
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

PRODUCT_NAME	PRICE	PHONE_CATEGORY
iPhone 12 Pro	1100	EXPENSIVE PHONES
iPhone 12	1000	MID RANGE PHONES
Galaxy Z Flip 3	1000	MID RANGE PHONES
Galaxy S21	1000	MID RANGE PHONES
OnePlus 9	800	CHEAPER PHONE

Result 58

VALUE FUNCTIONS

```sql
/* DIFFERENCE BETWEEN LEAD() AND LAG() WINDOW FUNCTIONS WITH EXAMPLE */

SELECT E.*,
    LAG(SALARY) OVER (PARTITION BY DEPT_NAME ORDER BY EMP_ID) AS "PREV_EMP_SALARY",
    LEAD(SALARY) OVER (PARTITION BY DEPT_NAME ORDER BY EMP_ID) AS "NEXT_EMP_SALARY"
FROM EMPLOYEE E;

/* Explanation of LAG and LEAD Functions in the above Example:
    LAG(SALARY)     : This function provides access to the salary value of the previous employee within the same department. If there is no previous row, it returns NULL.
    LEAD(SALARY)    : This function provides access to the salary value of the next employee within the same department. If there is no next row, it returns NULL.
Practical Use Cases :
    LAG(SALARY)     : Useful for comparing an employee's salary with that of the previous employee in the same department, aiding in trend analysis or discrepancy checks.
    LEAD(SALARY)    : Useful for comparing an employee's salary with that of the next employee in the same department, helping to understand future salary trends
                      or to check salary progression. */
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

emp_ID	emp_NAME	DEPT_NAME	SALARY	PREV_EMP_SALARY	NEXT_EMP_SALARY
101	Mohan	Admin	4000	NULL	4000
108	Maryam	Admin	4000	4000	2000
113	Gautham	Admin	2000	4000	5000
120	Monica	Admin	5000	2000	NULL
104	Dorvin	Finance	6500	NULL	5000
106	Rajesh	Finance	5000	6500	6500
116	Satya	Finance	6500	5000	5500
118	Tejaswi	Finance	5500	6500	NULL

Result 24

VALUE FUNCTIONS

```sql
/* Write query to display the most and least expensive product under each category (corresponding to each record) */

SELECT P.*,
    FIRST_VALUE(PRODUCT_NAME) OVER (PARTITION BY PRODUCT_CATEGORY ORDER BY PRICE DESC) AS MOST_EXPENSIVE,
    LAST_VALUE(PRODUCT_NAME) OVER (PARTITION BY PRODUCT_CATEGORY ORDER BY PRICE DESC
                    RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS LEAST_EXPENSIVE
FROM PRODUCT P;

/* Explanation of FIRST_VALUE and LAST_VALUE Functions
    FIRST_VALUE(PRODUCT_NAME)   : This function returns the PRODUCT_NAME of the first row within each product category when ordered by PRICE. Since it orders
                                  by price in ascending order, it effectively returns the name of the product with the lowest price within each category.
    LAST_VALUE(PRODUCT_NAME)    : This function returns the PRODUCT_NAME of the last row within each product category when ordered by PRICE. The window frame
                                  RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ensures that it looks at the entire partition, thus returning the
                                  name of the product with the highest price within each category.
Practical Use Cases
    FIRST_VALUE(PRODUCT_NAME)   : Useful for identifying the most affordable product within each category, aiding in pricing strategy and competitive analysis.
    LAST_VALUE(PRODUCT_NAME)    : Useful for identifying the most expensive product within each category, helping in understanding the top-tier pricing and luxury
                                  segment offerings.*/
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

product_category	brand	product_name	price	MOST_EXPENSIVE	LEAST_EXPENSIVE
Earphone	Apple	AirPods Pro	280	AirPods Pro	Galaxy Buds Live
Earphone	Sony	WF-1000XM4	250	AirPods Pro	Galaxy Buds Live
Earphone	Samsung	Galaxy Buds Pro	220	AirPods Pro	Galaxy Buds Live
Earphone	Samsung	Galaxy Buds Live	170	AirPods Pro	Galaxy Buds Live
Headphone	Apple	AirPods Max	550	AirPods Max	Surface Headphones 2
Headphone	Sony	WH-1000XM4	400	AirPods Max	Surface Headphones 2
Headphone	Microsoft	Surface Headphones 2	250	AirPods Max	Surface Headphones 2

ANALYTIC FUNCTIONS

```sql
-- CUME_DIST()
--  QUERY TO FETCH ALL PRODUCTS WHICH ARE CONSTITUTING THE FIRST 30% OF THE DATA IN PRODUCTS TABLE BASED ON PRICE

SELECT PRODUCT_NAME, CUME_DIST_PERCENTAGE
FROM (
    SELECT *,
        CUME_DIST() OVER (ORDER BY PRICE DESC) AS CUME_DISTRIBUTION,
        CONCAT(ROUND(CUME_DIST() OVER (ORDER BY PRICE DESC)*100,2),'%') AS CUME_DIST_PERCENTAGE
        FROM PRODUCT) X
WHERE X.CUME_DISTRIBUTION <= 0.3;

/* Purpose and Usage :
    The query is designed to identify and display products that fall within the lowest 30% of prices within the dataset.
    This can be useful for various business analyses, such as:

        Identifying Affordable Products : Highlighting products that are in the lower price range for promotions or discounts.
        Market Segmentation             : Understanding the distribution of product prices and identifying budget-friendly options for certain customer segments.
        Inventory Management            : Assessing which lower-priced products might require different stocking strategies.
*/
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

PRODUCT_NAME	CUME_DIST_PERCENTAGE
XPS 17	3.7%
XPS 15	7.41%
Surface Laptop 4	11.11%
MacBook Pro 13	18.52%
XPS 13	18.52%
Galaxy Z Fold 3	22.22%
iPhone 12 Pro Max	25.93%

Result 44

ANALYTIC FUNCTIONS

```sql
-- PERCENT_RANK()
-- QUERY TO IDENTIFY HOW MANY PERCENTAGE MORE EXPENSIVE IS "GALAXY Z FOLD 3 " WHEN COMPARED TO ALL PRODUCTS.

SELECT PRODUCT_NAME, PERCENTAGE_RANK
FROM (
    SELECT *,
        PERCENT_RANK() OVER (ORDER BY PRICE),
        CONCAT(ROUND(PERCENT_RANK() OVER (ORDER BY PRICE)*100,2),' %') AS PERCENTAGE_RANK
        FROM PRODUCT) X
WHERE X.PRODUCT_NAME = "GALAXY Z FOLD 3";

/* Purpose and Usage
    The purpose of this query is to determine and display the percentile rank of the product "GALAXY Z FOLD 3" in terms of its price.
    The percentile rank helps in understanding the product's relative price position compared to other products.

Practical Use Cases
    Pricing Strategy    : Understanding where a product stands in the price spectrum relative to others.
    Market Positioning  : Assessing how a product is positioned price-wise within the entire product range.
    Sales and Marketing : Identifying premium or budget products based on their price percentile.
*/
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

PRODUCT_NAME	PERCENTAGE_RANK
Galaxy Z Fold 3	80.77 %

S

Q

L

SQL VIEWS

CREATING A VIEW

```
64
65      -- Fetch the order summary (to be given to client/vendor)
66
67  •   CREATE VIEW ORDER_SUMMARY
68          AS (SELECT od.prod_id, od.date, c.cust_name, p.prod_name,
69              round((p.price * od.quantity) - (p.price * od.quantity )* (od.disc_percent/100),2) as cost
70              FROM order_details od
71              JOIN customer_data c
72              ON c.cust_id = od.cust_id
73              JOIN product_info p
74              ON p.prod_id = od.prod_id
75              ORDER BY prod_id, cust_name);
76
77  •   SELECT * FROM ORDER_SUMMARY;
78
79      /* HERE I HAVE CREATED A VIEW ORDER_SUMMARY BY JOINING DATAS FROM 3 DIFFERENT TABLES. */
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

prod_id	date	cust_name	prod_name	cost
P1	2020-01-01	Mohan Kumar	Samsung S22	1440.00
P2	2020-01-01	James Xavier	Google Pixel 6 Pro	900.00
P2	2020-02-01	Priyanka Verma	Google Pixel 6 Pro	2160.00
P3	2020-03-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-02-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-05-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-04-01	Mohan Kumar	Sony Bravia TV	1800.00

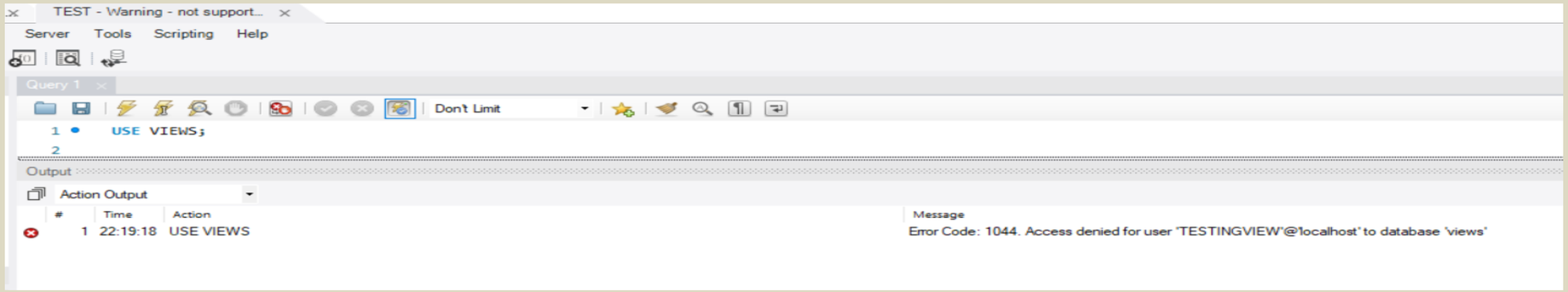
ORDER_SUMMARY 17

CREATING A NEW USER TO TEST FOR THE VIEW ACCESS

CREATING USER

```
81      -- Creating a new user to test for the view access
82
83  •   CREATE USER 'TESTVIEW'@'localhost'
84      IDENTIFIED BY 'TESTPWD';
```

CHECKING ACCESS



HERE FOR THE USER “TESTVIEW” THE ACCESS TO DATABASE AND VIEWS WAS NOT GRANTED , SO THE QUERY RETURNED AN ERROR SAYING ACCESS DENIED

GRANTING ACCESS

```
86  •   GRANT SELECT ON
87      VIEWS.ORDER_SUMMARY TO 'TESTINGVIEW';
```

CHECKING AFTER GRANTING ACCESS

```
1  USE VIEWS;
2
3  SELECT * FROM ORDER_SUMMARY;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

prod_id	date	cust_name	prod_name	cost
P1	2020-01-01	Mohan Kumar	Samsung S22	1440.00
P2	2020-01-01	James Xavier	Google Pixel 6 Pro	900.00
P2	2020-02-01	Priyanka Verma	Google Pixel 6 Pro	2160.00
P3	2020-05-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-04-01	Mohan Kumar	Sony Bravia TV	1800.00
P3	2020-03-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-02-01	Mohan Kumar	Sony Bravia TV	600.00
P5	2020-02-01	James Xavier	iPhone 13	800.00
P5	2020-03-01	Priyanka Verma	iPhone 13	800.00
P6	2020-05-01	James Xavier	Macbook Pro 16	5000.00
P6	2020-07-01	Mohan Kumar	Macbook Pro 16	17500.00
P6	2020-06-01	Priyanka Verma	Macbook Pro 16	5000.00

Here after granting access to user “TESTINGVIEW” for using the view “ORDER_SUMMARY” , it could fetch all the records from it.

CHECKING THE ACCESS FOR OTHER TABLES FROM THE DATABASE “VIEWS”

```
1  USE VIEWS;
2
3  SELECT * FROM ORDER_SUMMARY;    -- VIEW ON THE DATABASE VIEWS
4  SELECT * FROM PRODUCT_INFO;     -- TABLE ON THE DATABASE VIEWS
```

Output

Action Output

#	Time	Action	Message
1	22:19:18	USE VIEWS	Error Code: 1044. Access denied for user 'TESTINGVIEW'@'localhost' to database 'views'
2	22:20:50	USE VIEWS	0 row(s) affected
3	22:21:23	SELECT * FROM ORDER_SUMMARY	12 row(s) returned
4	22:23:09	SELECT * FROM PRODUCT_INFO	Error Code: 1142. SELECT command denied to user 'TESTINGVIEW'@'localhost' for table 'views'.'product_info'

Here since the user was not given access to the table “PRODUCT_INFO”, his query to fetch the records from the table was denied.

CREATE OR REPLACE

When you create a view using the `CREATE VIEW` command, you define its structure (columns and their data types) and the query that populates it. Suppose you want to modify the view's definition without dropping and recreating it. This is where `CREATE OR REPLACE VIEW` comes in handy. When you use `CREATE OR REPLACE VIEW`, it checks if the view already exists. If it does, it updates the view definition based on the specified query. This preserves any view privileges granted to users or roles. Different database systems (such as Oracle, MySQL, or SQL Server) may have slight variations in syntax, but the concept remains consistent.

```
93      -- UPDATING A COLUMN NAME.
94  •   CREATE OR REPLACE VIEW ORDER_SUMMARY
95          AS (SELECT od.prod_id, od.date AS order_date, c.cust_name, p.prod_name,
96              round((p.price * od.quantity) - (p.price * od.quantity )* (od.disc_percent/100),2) as cost
97              FROM order_details od
98              JOIN customer_data c
99              ON c.cust_id = od.cust_id
100             JOIN product_info p
101             ON p.prod_id = od.prod_id
102             ORDER BY prod_id, cust_name);
103
104 •   SELECT * FROM ORDER_SUMMARY;
105
106     -- HERE I HAVE USED "CREATE OR REPLACE" COMMAND TO RENAME THE COLUMN DATE TO ORDER_DATE
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

prod_id	order_date	cust_name	prod_name	cost
P1	2020-01-01	Mohan Kumar	Samsung S22	1440.00
P2	2020-01-01	James Xavier	Google Pixel 6 Pro	900.00
P2	2020-02-01	Priyanka Verma	Google Pixel 6 Pro	2160.00
P3	2020-05-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-04-01	Mohan Kumar	Sony Bravia TV	1800.00
P3	2020-03-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-02-01	Mohan Kumar	Sony Bravia TV	600.00
P5	2020-02-01	James Xavier	iPhone 13	800.00
P5	2020-03-01	Priyanka Verma	iPhone 13	800.00

ORDER_SUMMARY 19

RENAMING A VIEW

```
110     -- RENAME THE VIEW
111
112 •   CREATE OR REPLACE VIEW CUSTOMER_ORDER_SUMMARY
113     AS
114     SELECT * FROM ORDER_SUMMARY;
115
116 •   SELECT * FROM CUSTOMER_ORDER_SUMMARY;
117 •   DROP VIEW ORDER_SUMMARY;                    -- DELETING THE OLD VIEW.
118
119     /*  Note : In MYSQL we cannot rename view by using ALTER command. we have to create a new view from the existing one and delete the previous.
120         In some DBMS, renaming a view use the same syntax as renaming a table
121
122         ALTER VIEW order_summary RENAME TO customer_order_summary;
123     */
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

prod_id	order_date	cust_name	prod_name	cost
P1	2020-01-01	Mohan Kumar	Samsung S22	1440.00
P3	2020-02-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-03-01	Mohan Kumar	Sony Bravia TV	600.00
P3	2020-04-01	Mohan Kumar	Sony Bravia TV	1800.00
P3	2020-05-01	Mohan Kumar	Sony Bravia TV	600.00

CUSTOMER_ORDER_SUMMARY...

UPDATING A VIEW

UPDATING A VIEW CREATED USING MULTIPLE TABLES

```
130     -- UPDATE COMMAND
131 •       UPDATE CUSTOMER_ORDER_SUMMARY
132         SET COST = 2000
133         WHERE PROD_ID = "P1";
134
135     /* YOU CAN UPDATE ONLY THOSE VIEWS WHICH ARE CREATED BY USING 1 TABLE/VIEW. SINCE THE VIEW "CUSTOMER_ORDER_HISTORY' IS CREATED USING JOINING MULTIPLE TABLES
136            IT RETURNED AN ERROR */
137
```

Output

Action Output

#	Time	Action	Message	Duration / Fetch
74	00:01:13	UPDATE CUSTOMER_ORDER_SUMMARY SET COST = 2000 WHERE PROD_ID = "P1"	Error Code: 1348. Column 'cost' is not updatable	0.000 sec

UPDATING A VIEW CREATED USING SINGLE TABLE

```
135 •   CREATE OR REPLACE VIEW EXPENSIVE_PRODUCTS
136     AS
137     SELECT * FROM PRODUCT_INFO WHERE PRICE>1000;
138
139 •   SELECT * FROM EXPENSIVE_PRODUCTS;
140
141 •   UPDATE EXPENSIVE_PRODUCTS
142     SET PRICE=2500
143     WHERE PROD_ID="P4";
144
145 •   SELECT * FROM PRODUCT_INFO;
146     /* NOTE : 1.HERE THE VIEW EXPENSIVE_PRODUCTS IS CREATED USING A SINGLE TABLE, THE VIEW COULD BE UPDATED.
147               2. WHEN YOU UPDATE A VIEW CREATED USING A SINGLE TABLE / VIEW, THE ORIGIN TABLE ALSO WILL GET UPDATED. HERE THE TABLE PRODUCT_INFO ALSO
148                  GOT UPDATED WHILE UPDATING THE VIEW.
149     */
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

prod_id	prod_name	brand	price
P1	Samsung S22	Samsung	800
P2	Google Pixel 6 Pro	Google	900
P25	NOTE 20	SAMSUNG	2700
P3	Sony Bravia TV	Sony	600
P4	Dell XPS 17	Dell	2500

PRODUCT_INFO 28

RULES FOR UPDATING A VIEW

1. While updating a view ,the column list along with its name and data type should be same as used when creation of the view.

2. New columns can be added only to end of the column list

3. Query with JOINS, table list, Order by clause can be changed.

4. View created using just one table or view can be updated.

5. The query while creating view should not contain distinct clause, or group by clause

6. It Should not also contain window functions and WITH clause.

7. Have no subqueries in the SELECT statement.

WITH CHECK OPTION

The WITH CHECK OPTION clause in SQL views ensures the consistency of data when inserting or updating rows through the view. The WITH CHECK OPTION clause restricts modifications (inserts or updates) to rows that conform to the original query used to create the view.

```
178     -- WITH CHECK OPTION
179 •      CREATE OR REPLACE VIEW APPLE_PRODUCTS
180        AS
181        SELECT * FROM PRODUCT_INFO
182        WHERE BRAND = "APPLE"
183        WITH CHECK OPTION;
184
185 •      INSERT INTO APPLE_PRODUCTS
186        VALUES ("P25", "NOTE 20","SAMSUNG",2700);
187
188 •       SELECT * FROM APPLE_PRODUCTS;
189
190        /*  IF YOU DONT USE WITH CHECK OPTION THE VIEW WILL BE UPDATED EVENTHOUGH IT WONT SHOW TH RECORD
191           .THIS IS BECAUSE THE WITH CHECK OPTION WILL CHECK FOR THE CONDITION BRAND ="APPLE
192           */
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

prod_id	prod_name	brand	price
P5	iPhone 13	Apple	800
P6	Macbook Pro 16	Apple	5000

APPLE_PRODUCTS 32

S

Q

L

RECURSIVE SQL QUERIES

PROBLEM 1:
Display number from 1 to 10 without using any in built functions.

```sql
WITH RECURSIVE numbers AS
    (SELECT 1 AS n
     UNION
     SELECT n+1
     FROM numbers
     WHERE n<10
    )
SELECT * FROM numbers;

/* The query generates the numbers 1 through 10 recursively. It starts with the anchor member 1, then iteratively adds 1 to the
   previous value until reaching 10, as specified by the termination condition.
   Finally, the SELECT statement retrieves all the generated numbers.*/
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

n
1
2
3
4
5
6
7
8
9
10

PROBLEM 2:

Find the Hierarchy of employees under a given employee. Also displaying the manager name.

```sql
WITH RECURSIVE emp_Hierarchy AS
    (SELECT id, name, manager_id, designation , 1 AS lvl
    FROM emp_details WHERE name = "Asha"
    UNION
    SELECT E.id, E.name, E.manager_id, E.designation , H.lvl+1 AS lvl
    FROM emp_Hierarchy H
    JOIN emp_details E
    ON H.id = E.manager_id )
SELECT H2.id AS emp_id, H2.name AS emp_name, E2.name AS manager_name,H2.lvl AS level
FROM emp_Hierarchy H2
JOIN emp_details E2
ON E2.id = H2.manager_id
ORDER BY level;

/* 1. The query starts with an anchor member selecting the details of the employee named "Asha" from the emp_details table and assigns a level (lvl) of 1.
   2. The recursive member then joins the emp_Hierarchy CTE with the emp_details table to retrieve the details of employees who directly
      report to the employees already present in the CTE. It increments the level (lvl) by 1 for each level down the hierarchy.
   3. After the recursive CTE (emp_Hierarchy) has fetched all the hierarchical information, the final SELECT statement retrieves specific details
      of employees and their managers in the hierarchy.
   4. It joins the emp_Hierarchy CTE with the emp_details table to retrieve the manager's name for each employee.*/
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

emp_id	emp_name	manager_name	level
7	Asha	Shripadh	1
5	Michael	Asha	2
6	Arvind	Asha	2
4	David	Michael	3
2	Satya	Michael	3
3	Jia	Michael	3

Result 23

PROBLEM 3:

Find the Hierarchy of managers for a given employee.

```sql
WITH RECURSIVE managers AS
    (SELECT id, name, manager_id, designation, 1 AS lvl
    FROM emp_details WHERE name="Jia"
    UNION
    SELECT E.id, E.name, E.manager_id, E.designation , M.lvl+1 AS lvl
    FROM managers M
    JOIN emp_details E
    ON M.manager_id = E.id)
SELECT M2.id AS emp_id, M2.name AS emp_name, E2.name AS manager_name, M2.lvl AS level
FROM managers M2
JOIN emp_details E2
ON E2.id = M2.manager_id
ORDER BY level;

/* 1. This query begins by selecting the details of the employee named "Jia" from the emp_details table and assigns a level of 1 to this initial employee.
   2. Then, it recursively joins the managers CTE with the emp_details table to retrieve the details of employees who are managed by those already present in the CTE.
   3. Each time a new level of employees is added, the level (lvl) is incremented by 1. This process continues until all levels of the hierarchy have been retrieved.
   4. In the final result, each employee's ID, name, and their manager's name are displayed, along with the corresponding level in the hierarchy. */
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

emp_id	emp_name	manager_name	level
3	Jia	Michael	1
5	Michael	Asha	2
7	Asha	Shripadh	3

Result 24

S

Q

L

PIVOTING IN SQL

PIVOTING

ORIGINAL TABLE :

```
30 •    SELECT * FROM SALES_DATA;
31
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

sales_date	customer_id	amount
2021-01-01	Cust-1	50$
2021-01-02	Cust-1	50$
2021-01-03	Cust-1	50$
2021-01-01	Cust-2	100$
2021-01-02	Cust-2	100$
2021-01-03	Cust-2	100$
2021-02-01	Cust-2	-100$
2021-02-02	Cust-2	-100$
2021-02-03	Cust-2	-100$
2021-03-01	Cust-3	1$
2021-04-01	Cust-3	1$
2021-05-01	Cust-3	1$
2021-06-01	Cust-3	1$
2021-07-01	Cust-3	-1$
2021-08-01	Cust-3	-1$
2021-09-01	Cust-3	-1$
2021-10-01	Cust-3	-1$
2021-11-01	Cust-3	-1$
2021-12-01	Cust-3	-1$

SALES_DATA 26

PIVOTING

Here I am using SQL queries to leverage multiple CTEs to:

1. Prepare and clean the data (SALES).
2. Aggregate sales per customer for each month and total (SALES_PER_CUST).
3. Aggregate total sales for each month across all customers (SALES_PER_MONTH).
4. Combine these aggregates (FINAL_DATA).
5. Format the final output with appropriate signs and dollar symbols.

PIVOTING

CTE “SALES” :

- This Common Table Expression (CTE) creates a temporary result set SALES.
- It selects customer ID, formats the sales date to 'Month_Year' format, and removes the dollar sign from the amount.

```
32  WITH SALES AS (
33      SELECT
34          CUSTOMER_ID AS CUSTOMER,
35          DATE_FORMAT(SALES_DATE, '%b_%y') AS SALES_DATE,
36          REPLACE(AMOUNT, "$", "") AS AMOUNT
37      FROM SALES_DATA
38  ),
39
```

PIVOTING

CTE “SALES_PER_CUST” :

- This CTE aggregates sales data per customer for each month in 2021.
- It uses CASE statements to sum amounts for each month and calculates the total sales per customer.

```
40  SALES_PER_CUST AS (
41      SELECT CUSTOMER,
42          SUM(CASE WHEN SALES_DATE = "Jan_21" THEN AMOUNT ELSE 0 END) AS Jan_21,
43          SUM(CASE WHEN SALES_DATE = "Feb_21" THEN AMOUNT ELSE 0 END) AS Feb_21,
44          SUM(CASE WHEN SALES_DATE = "Mar_21" THEN AMOUNT ELSE 0 END) AS Mar_21,
45          SUM(CASE WHEN SALES_DATE = "Apr_21" THEN AMOUNT ELSE 0 END) AS Apr_21,
46          SUM(CASE WHEN SALES_DATE = "May_21" THEN AMOUNT ELSE 0 END) AS May_21,
47          SUM(CASE WHEN SALES_DATE = "Jun_21" THEN AMOUNT ELSE 0 END) AS Jun_21,
48          SUM(CASE WHEN SALES_DATE = "Jul_21" THEN AMOUNT ELSE 0 END) AS Jul_21,
49          SUM(CASE WHEN SALES_DATE = "Aug_21" THEN AMOUNT ELSE 0 END) AS Aug_21,
50          SUM(CASE WHEN SALES_DATE = "Sep_21" THEN AMOUNT ELSE 0 END) AS Sep_21,
51          SUM(CASE WHEN SALES_DATE = "Oct_21" THEN AMOUNT ELSE 0 END) AS Oct_21,
52          SUM(CASE WHEN SALES_DATE = "Nov_21" THEN AMOUNT ELSE 0 END) AS Nov_21,
53          SUM(CASE WHEN SALES_DATE = "Dec_21" THEN AMOUNT ELSE 0 END) AS Dec_21,
54          SUM(AMOUNT) AS TOTAL
55      FROM SALES S
56      GROUP BY CUSTOMER),
57
```

PIVOTING

CTE “SALES_PER_MONTH” :

- This CTE calculates the total sales for each month across all customers.
- It labels this row with 'TOTAL' as the customer.

```
58      SALES_PER_MONTH AS
59          (SELECT 'TOTAL' AS CUSTOMER,
60              SUM(CASE WHEN SALES_DATE = "Jan_21" THEN AMOUNT ELSE 0 END) AS Jan_21,
61              SUM(CASE WHEN SALES_DATE = "Feb_21" THEN AMOUNT ELSE 0 END) AS Feb_21,
62              SUM(CASE WHEN SALES_DATE = "Mar_21" THEN AMOUNT ELSE 0 END) AS Mar_21,
63              SUM(CASE WHEN SALES_DATE = "Apr_21" THEN AMOUNT ELSE 0 END) AS Apr_21,
64              SUM(CASE WHEN SALES_DATE = "May_21" THEN AMOUNT ELSE 0 END) AS May_21,
65              SUM(CASE WHEN SALES_DATE = "Jun_21" THEN AMOUNT ELSE 0 END) AS Jun_21,
66              SUM(CASE WHEN SALES_DATE = "Jul_21" THEN AMOUNT ELSE 0 END) AS Jul_21,
67              SUM(CASE WHEN SALES_DATE = "Aug_21" THEN AMOUNT ELSE 0 END) AS Aug_21,
68              SUM(CASE WHEN SALES_DATE = "Sep_21" THEN AMOUNT ELSE 0 END) AS Sep_21,
69              SUM(CASE WHEN SALES_DATE = "Oct_21" THEN AMOUNT ELSE 0 END) AS Oct_21,
70              SUM(CASE WHEN SALES_DATE = "Nov_21" THEN AMOUNT ELSE 0 END) AS Nov_21,
71              SUM(CASE WHEN SALES_DATE = "Dec_21" THEN AMOUNT ELSE 0 END) AS Dec_21,
72              " " AS TOTAL
73          FROM SALES S),
```

PIVOTING

CTE “FINAL_DATA” :

- This part combines the customer-wise monthly sales and the overall monthly totals into a single result set..

```
75      FINAL_DATA AS
76          (SELECT * FROM SALES_PER_CUST
77          UNION
78          SELECT * FROM SALES_PER_MONTH)
79
```

PIVOTING

FINAL SELECTION AND FORMATTING:

- This part of the query selects from the FINAL_DATA and formats the amounts.
- It converts negative amounts to strings with parentheses and appends a dollar sign to all amounts.
- The CASE statements handle the formatting, ensuring negative values are displayed correctly.

```
80      SELECT CUSTOMER,
81          CASE WHEN Jan_21 < 0 THEN CONCAT('(', Jan_21 * -1 , ')$') ELSE CONCAT(Jan_21, '$') END AS Jan_21,
82          CASE WHEN Feb_21 < 0 THEN CONCAT('(', Feb_21 * -1 , ')$') ELSE CONCAT(Feb_21, '$') END AS Feb_21,
83          CASE WHEN Mar_21 < 0 THEN CONCAT('(', Mar_21 * -1 , ')$') ELSE CONCAT(Mar_21, '$') END AS Mar_21,
84          CASE WHEN Apr_21 < 0 THEN CONCAT('(', Apr_21 * -1 , ')$') ELSE CONCAT(Apr_21, '$') END AS Apr_21,
85          CASE WHEN May_21 < 0 THEN CONCAT('(', May_21 * -1 , ')$') ELSE CONCAT(May_21, '$') END AS May_21,
86          CASE WHEN Jun_21 < 0 THEN CONCAT('(', Jun_21 * -1 , ')$') ELSE CONCAT(Jun_21, '$') END AS Jun_21,
87          CASE WHEN Jul_21 < 0 THEN CONCAT('(', Jul_21 * -1 , ')$') ELSE CONCAT(Jul_21, '$') END AS Jul_21,
88          CASE WHEN Aug_21 < 0 THEN CONCAT('(', Aug_21 * -1 , ')$') ELSE CONCAT(Aug_21, '$') END AS Aug_21,
89          CASE WHEN Sep_21 < 0 THEN CONCAT('(', Sep_21 * -1 , ')$') ELSE CONCAT(Sep_21, '$') END AS Sep_21,
90          CASE WHEN Oct_21 < 0 THEN CONCAT('(', Oct_21 * -1 , ')$') ELSE CONCAT(Oct_21, '$') END AS Oct_21,
91          CASE WHEN Nov_21 < 0 THEN CONCAT('(', Nov_21 * -1 , ')$') ELSE CONCAT(Nov_21, '$') END AS Nov_21,
92          CASE WHEN Dec_21 < 0 THEN CONCAT('(', Dec_21 * -1 , ')$') ELSE CONCAT(Dec_21, '$') END AS Dec_21,
93          CASE WHEN Total = '' THEN '' WHEN Total < 0 THEN CONCAT('(', Total * -1 , ')$') ELSE CONCAT(Total, '$')
94          END AS Total
95      FROM FINAL_DATA;
```

PIVOTING

FINAL OUTPUT TABLE :

CUSTOMER	Jan_21	Feb_21	Mar_21	Apr_21	May_21	Jun_21	Jul_21	Aug_21	Sep_21	Oct_21	Nov_21	Dec_21	Total
Cust-1	150$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$	150$
Cust-2	300$	(300)$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$	0$
Cust-3	0$	0$	1$	1$	1$	1$	(1)$	(1)$	(1)$	(1)$	(1)$	(1)$	(2)$
TOTAL	450$	(300)$	1$	1$	1$	1$	(1)$	(1)$	(1)$	(1)$	(1)$	(1)$	

S

Q

L

PROCEDURES

PROCEDURE WITHOUT PARAMETER

PROCEDURE WITHOUT PARAMETER

TABLES BEFORE PROCEDURE

```
65    SELECT * FROM PRODUCTS;
66
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

product_code	product_name	price	quantity_remaining	quantity_sold
P1	iPhone 13 Pro Max	1000	5	195
NULL	NULL	NULL	NULL	NULL

```
63    SELECT * FROM SALES;
64
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

order_id	order_date	product_code	quantity_ordered	sale_price
1	2022-01-10	P1	100	120000
2	2022-01-20	P1	50	60000
3	2022-02-05	P1	45	540000

PROCEDURE WITHOUT PARAMETER

This procedure is designed to process the sale of an "iPhone 13 Pro Max" by:

- Retrieving its product code and price.
- Inserting a sale record.
- Updating the inventory to reflect the sale.
- Returning a confirmation message.

```
37      DELIMITER $$
38      CREATE PROCEDURE BUY_PRODUCTS()
39      BEGIN
40          DECLARE V_PRODUCT_CODE VARCHAR(20);
41          DECLARE V_PRICE INT;
42
43          SELECT PRODUCT_CODE , PRICE
44          INTO V_PRODUCT_CODE, V_PRICE
45          FROM PRODUCTS
46          WHERE PRODUCT_NAME = 'iPhone 13 Pro Max';
47
48          INSERT INTO SALES(ORDER_DATE, PRODUCT_CODE, QUANTITY_ORDERED, SALE_PRICE)
49              VALUES (CAST(NOW() AS DATE) , V_PRODUCT_CODE, 1, (V_PRODUCT_CODE * 1));
50
51          UPDATE PRODUCTS
52          SET QUANTITY_REMAINING = (QUANTITY_REMAINING - 1) ,
53              QUANTITY_SOLD = (QUANTITY_SOLD + 1)
54          WHERE PRODUCT_CODE = V_PRODUCT_CODE;
55
56          SELECT 'PRODUCT SOLD';
57      END $$
58
```

PROCEDURE WITHOUT PARAMETER

- The procedure BUY_PRODUCTS is defined here. The DELIMITER $$ statement changes the delimiter from ; to $$ to allow the procedure definition to contain semicolons.

- Two variables are declared:
  - V_PRODUCT_CODE : To store the product code of 'iPhone 13 Pro Max'.
  - V_PRICE : To store the price of the 'iPhone 13 Pro Max’.

- The SELECT statement retrieves the PRODUCT_CODE and PRICE of the product named 'iPhone 13 Pro Max' from the PRODUCTS table and stores these values into the variables V_PRODUCT_CODE and V_PRICE, respectively.

- INSERT statement adds a new record into the SALES table

- UPDATE statement updates the PRODUCTS table:
  - It decreases the QUANTITY_REMAINING by 1.
  - It increases the QUANTITY_SOLD by 1.
  - The update is applied to the product with the PRODUCT_CODE equal to V_PRODUCT_CODE

- The last SELECT statement returns the message 'PRODUCT SOLD' to indicate that the procedure has completed successfully.

- END $$ marks the end of the procedure definition.

PROCEDURE WITHOUT PARAMETER

CALLING THE PROCEDURE

```
62
63      CALL BUY_PRODUCTS();
64
```

Result Grid | Filter Rows: | Export:

PRODUCT SOLD
PRODUCT SOLD

UPON CALLING THE PROCEDURE , A SUCCESSFUL MESSAGE “PRODUCT SOLD” WAS RETURNED.

TABLES AFTER PROCEDURE

```
66      SELECT * FROM SALES;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

order_id	order_date	product_code	quantity_ordered	sale_price
1	2022-01-10	P1	100	120000
2	2022-01-20	P1	50	60000
3	2022-02-05	P1	45	540000
4	2024-06-03	P1	1	1000
NULL	NULL	NULL	NULL	NULL

SALES 27

THE SALES TABLE IS UPDATED WITH A NEW RECORD

```
64      SELECT * FROM PRODUCTS;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

product_code	product_name	price	quantity_remaining	quantity_sold
P1	iPhone 13 Pro Max	1000	4	196
NULL	NULL	NULL	NULL	NULL

QUANTITY_REMAINING IS REDUCED FROM 5 TO 4 AND THE QUANTITY_SOLD COLUMN IS INCREMENTED FROM 195 TO 196.

PROCEDURE WITH PARAMETER

PROCEDURE WITH PARAMETER

TABLES BEFORE PROCEDURE

```
119     SELECT * FROM PRODUCTS_NW;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

product_code	product_name	price	quantity_remaining	quantity_sold
P1	iPhone 13 Pro Max	1200	5	195
P2	AirPods Pro	279	10	90
P3	MacBook Pro 16	5000	2	48
P4	iPad Air	650	1	9
NULL	NULL	NULL	NULL	NULL

```
121     SELECT * FROM SALES_NW;
122
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap

order_id	order_date	product_code	quantity_ordered	sale_price
1	2022-01-10	P1	100	120000
2	2022-01-20	P1	50	60000
3	2022-02-05	P1	45	540000
4	2022-01-15	P2	50	13950
5	2022-03-25	P2	40	11160
6	2022-02-25	P3	10	50000
7	2022-03-15	P3	10	50000
8	2022-03-25	P3	20	100000
9	2022-04-21	P3	8	40000

SALES_NW 30

PROCEDURE WITH PARAMETER

The BUY_PRODUCTS_NW procedure is designed to:

- Check if the specified quantity of a product is available.
- If available, retrieve the product details and insert a sale record.
- Update the product inventory to reflect the sale.
- Return a confirmation message 'PRODUCT SOLD'.
- If not available, return the message 'INSUFFICIENT QUANTITY'

```
DELIMITER $$
CREATE PROCEDURE BUY_PRODUCTS_NW(P_PRODUCT_NAME VARCHAR(20), P_QUANTITY INT)
BEGIN
    DECLARE V_COUNT INT;
    DECLARE V_PRODUCT_CODE VARCHAR(20);
    DECLARE V_PRICE INT;

    SELECT COUNT(*)
    INTO V_COUNT
    FROM PRODUCTS_NW
    WHERE PRODUCT_NAME = P_PRODUCT_NAME
    AND QUANTITY_REMAINING > P_QUANTITY;

    IF V_COUNT > 0
    THEN
        SELECT PRODUCT_CODE , PRICE
        INTO V_PRODUCT_CODE, V_PRICE
        FROM PRODUCTS_NW
        WHERE PRODUCT_NAME = P_PRODUCT_NAME
        AND QUANTITY_REMAINING > P_QUANTITY;

        INSERT INTO SALES_NW(ORDER_DATE, PRODUCT_CODE, QUANTITY_ORDERED, SALE_PRICE)
            VALUES (CAST(NOW() AS DATE) , V_PRODUCT_CODE, P_QUANTITY, (V_PRICE * P_QUANTITY));

        UPDATE PRODUCTS_NW
        SET QUANTITY_REMAINING = (QUANTITY_REMAINING - P_QUANTITY) ,
            QUANTITY_SOLD = (QUANTITY_SOLD + P_QUANTITY)
        WHERE PRODUCT_CODE = V_PRODUCT_CODE;

        SELECT 'PRODUCT SOLD';
    ELSE
        SELECT 'INSUFFICIENT QUANTITY';
    END IF;
END $$
```

- A new stored procedure called BUY_PRODUCTS_NW is created which takes two input parameters:
  - P_PRODUCT_NAME : The name of the product to be purchased.
  - P_QUANTITY : The quantity of the product to be purchased.

- Three variables are declared:
  - V_COUNT : To store the count of products matching the criteria.
  - V_PRODUCT_CODE : To store the product code of the specified product.
  - V_PRICE : To store the price of the specified product.

- The SELECT statement checks if there are enough products available:
  - It counts the number of products in the PRODUCTS_NW table that match the P_PRODUCT_NAME and have a QUANTITY_REMAINING greater than P_QUANTITY.
  - The count is stored in V_COUNT.

- The IF statement checks if the count (V_COUNT) is greater than 0, indicating that there is sufficient quantity available.

- If sufficient quantity is available:
  - The product code and price are retrieved for the specified product and stored in V_PRODUCT_CODE and V_PRICE.
  - A new sale record is inserted into the SALES table with:
    - ORDER_DATE : The current date.
    - PRODUCT_CODE : The code of the product.
    - QUANTITY_ORDERED : The quantity ordered (P_QUANTITY).
    - SALE_PRICE : The total sale price (price per unit multiplied by the quantity).

  The UPDATE statement adjusts the inventory:

    - Decreases the QUANTITY_REMAINING by the ordered quantity (P_QUANTITY).
    - Increases the QUANTITY_SOLD by the ordered quantity.
    - This update is applied to the product with the retrieved PRODUCT_CODE.

- If the product was sold, the final SELECT statement returns the message 'PRODUCT SOLD’.
- If there was not enough quantity available (V_COUNT is 0), the procedure returns the message 'INSUFFICIENT QUANTITY’.

PROCEDURE WITH PARAMETER

CALLING THE PROCEDURE

Example 1

```
158 •  CALL BUY_PRODUCTS_NW("AirPods Pro", 3);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

PRODUCT SOLD
PRODUCT SOLD

Example 2

```
158 •  CALL BUY_PRODUCTS_NW("iPad Air", 2);
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

INSUFFICIENT QUANTITY
INSUFFICIENT QUANTITY

HERE THE PROCEDURE RETURNED THE MESSAGE "INSUFFICIENT QUANTITY" AS FOR THE PRODUCT "IPAD AIR" ONLY 1 QUANTITY WAS AVAILABLE

TABLES AFTER PROCEDURE

```
163    SELECT * FROM SALES_NW;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cel

order_id	order_date	product_code	quantity_ordered	sale_price
8	2022-03-25	P3	20	100000
9	2022-04-21	P3	8	40000
10	2022-04-27	P4	9	5850
11	2024-06-03	P2	3	837
NULL	NULL	NULL	NULL	NULL

SALES_NW 44

THE SALES_NW TABLE IS UPDATED WITH A NEW RECORD

```
161    SELECT * FROM PRODUCTS_NW;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

product_code	product_name	price	quantity_remaining	quantity_sold
P1	iPhone 13 Pro Max	1200	5	195
P2	AirPods Pro	279	7	93
P3	MacBook Pro 16	5000	2	48
P4	iPad Air	650	1	9
NULL	NULL	NULL	NULL	NULL

PRODUCTS_NW 43

QUANTITY OF "AIRPODS PRO " IS REDUCED FROM 10 TO 7 AND THE QUANTITY_SOLD COLUMN IS INCREMENTED FROM 90 TO 93.

S

Q

L

FUNCTIONS IN SQL

CREATING A SALES TABLE FOR PRACTICE

CREATE

```
4   -- Create the sample table
5   CREATE TABLE SalesTransactions (
6       TransactionID INT PRIMARY KEY AUTO_INCREMENT,
7       ProductName VARCHAR(100),
8       ProductCategory VARCHAR(50),
9       TransactionDate DATETIME,
10      UnitPrice DECIMAL(10, 2),
11      QuantitySold INT,
12      TotalAmount DECIMAL(10, 2)
13  );
```

INSERT

```
15  -- Insert sample data into the table
16  INSERT INTO SalesTransactions (ProductName, ProductCategory, TransactionDate, UnitPrice, QuantitySold, TotalAmount)
17   VALUES ('Laptop', 'Electronics', '2023-05-01 10:00:00', 1000.00, 2, 2000.00),
18          ('Smartphone', 'Electronics', '2023-05-02 11:30:00', 800.00, 5, 4000.00),
19          ('Tablet', 'Electronics', '2023-05-03 12:45:00', 600.00, 3, 1800.00),
20          ('Headphones', 'Accessories', '2023-05-04 13:00:00', 150.00, 10, 1500.00),
21          ('Charger', 'Accessories', '2023-05-05 14:15:00', 20.00, 20, 400.00),
22          ('Office Chair', 'Furniture', '2023-05-06 15:00:00', 200.00, 4, 800.00),
23          ('Desk', 'Furniture', '2023-05-07 16:30:00', 300.00, 2, 600.00);
```

SALES TABLE

```
25  SELECT * FROM SALESTRANSACTIONS;
26
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

TransactionID	ProductName	ProductCategory	TransactionDate	UnitPrice	QuantitySold	TotalAmount
1	Laptop	Electronics	2023-05-01 10:00:00	1000.00	2	2000.00
2	Smartphone	Electronics	2023-05-02 11:30:00	800.00	5	4000.00
3	Tablet	Electronics	2023-05-03 12:45:00	600.00	3	1800.00
4	Headphones	Accessories	2023-05-04 13:00:00	150.00	10	1500.00
5	Charger	Accessories	2023-05-05 14:15:00	20.00	20	400.00
6	Office Chair	Furniture	2023-05-06 15:00:00	200.00	4	800.00
7	Desk	Furniture	2023-05-07 16:30:00	300.00	2	600.00
NULL	NULL	NULL	NULL	NULL	NULL	NULL

SCALAR FUNCTIONS

LENGTH

```
29      -- Get the length of the 'ProductName' field
30 •    SELECT ProductName, LENGTH(ProductName) AS NameLength FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductName	NameLength
Laptop	6
Smartphone	10
Tablet	6
Headphones	10
Charger	7

Result 2

UPPER

```
32      -- Convert 'ProductCategory' to uppercase
33 •    SELECT ProductCategory, UPPER(ProductCategory) AS Upper_Category FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductCategory	Upper_Category
Electronics	ELECTRONICS
Electronics	ELECTRONICS
Electronics	ELECTRONICS
Accessories	ACCESSORIES
Accessories	ACCESSORIES

Result 4

LOWER

```
35      -- Convert 'ProductCategory' to lowercase
36 •    SELECT ProductCategory, LOWER(ProductCategory) AS Lower_Category FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductCategory	Lower_Category
Electronics	electronics
Electronics	electronics
Electronics	electronics
Accessories	accessories
Accessories	accessories

Result 5

CONCAT

```
38      -- Concatenate 'ProductName' and 'ProductCategory'
39 •    SELECT CONCAT(ProductName, ' - ', ProductCategory) AS Full_Description
40      FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Full_Description
Laptop - Electronics
Smartphone - Electronics
Tablet - Electronics
Headphones - Accessories
Charger - Accessories
Office Chair - Furniture

Result 7

STRING FUNCTIONS

SUBSTRING

```
45      -- Extract substring from 'ProductName'
46 •    SELECT ProductName, SUBSTRING(ProductName, 1, 5) AS SubName FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductName	SubName
Laptop	Lapto
Smartphone	Smart
Tablet	Table
Headphones	Headp
Charger	Charg
Office Chair	Offic

Result 10

LOCATE

```
48      -- Find position of 'Phone' in 'ProductName'
49 •    SELECT ProductName, LOCATE('Phone', ProductName) AS PositionOfPhone FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductName	PositionOfPhone
Laptop	0
Smartphone	6
Tablet	0
Headphones	5
Charger	0

Result 11

REPLACE

```
51      -- Replace 'Phone' with 'Device' in 'ProductName'
52 •    SELECT ProductName, REPLACE(ProductName, 'Laptop', 'Device') AS NewProductName FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductName	NewProductName
Laptop	Device
Smartphone	Smartphone
Tablet	Tablet
Headphones	Headphones
Charger	Charger

Result 15

DATE AND TIME FUNCTIONS

NOW

```
57      -- Get the current date and time
58 •    SELECT NOW() AS CurrentDateTime;
59
```

Result Grid | Filter Rows: | Export:

CurrentDateTime
2024-06-04 00:06:38

DATE_ADD

```
60      -- Add 7 days to 'TransactionDate'
61 •    SELECT TransactionDate, DATE_ADD(TransactionDate, INTERVAL 7 DAY) AS NewTransactionDate
62      FROM SalesTransactions;
63
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

TransactionDate	NewTransactionDate
2023-05-01 10:00:00	2023-05-08 10:00:00
2023-05-02 11:30:00	2023-05-09 11:30:00
2023-05-03 12:45:00	2023-05-10 12:45:00
2023-05-04 13:00:00	2023-05-11 13:00:00
2023-05-05 14:15:00	2023-05-12 14:15:00

Result 17

DATEDIFF

```
64      -- Calculate the difference in days between 'TransactionDate' and today
65 •    SELECT TransactionDate, DATEDIFF(NOW(), TransactionDate) AS DaysSinceTransaction
66      FROM SalesTransactions;
67
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

TransactionDate	DaysSinceTransaction
2023-05-01 10:00:00	400
2023-05-02 11:30:00	399
2023-05-03 12:45:00	398
2023-05-04 13:00:00	397
2023-05-05 14:15:00	396

Result 18

MATHEMATICAL FUNCTIONS

ABS

```
72 •  UPDATE SALESTRANSACTIONS
73    SET TOTALAMOUNT = -4000
74    WHERE TRANSACTIONID= 2;
75
76    -- Get the absolute value of 'TotalAmount' (useful if dealing with adjustments)
77 •  SELECT TRANSACTIONID, TotalAmount, ABS(TotalAmount) AS AbsoluteTotalAmount FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

TRANSACTIONID	TotalAmount	AbsoluteTotalAmount
1	2000.00	2000.00
2	-4000.00	4000.00
3	1800.00	1800.00
4	1500.00	1500.00
5	400.00	400.00

Result 28

ROUND

```
79    -- Round the 'UnitPrice' to 1 decimal place
80 •  SELECT UnitPrice, ROUND(UnitPrice, 0) AS RoundedUnitPrice FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

UnitPrice	RoundedUnitPrice
1000.00	1000
800.00	800
600.00	600
150.00	150
20.00	20

Result 31

SQRT

```
82    -- Get the square root of 'QuantitySold' (not very common but for practice)
83 •  SELECT QuantitySold, SQRT(QuantitySold) AS SquareRootQuantitySold FROM SalesTransactions;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

QuantitySold	SquareRootQuantitySold
2	1.4142135623730951
5	2.23606797749979
3	1.7320508075688772
10	3.1622776601683795
20	4.47213595499958

Result 32

SQL

COALESCE

EMPLOYEE TABLE

```
21 •    SELECT * FROM EMPLOYEEDETAILS;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

EmployeeID	FirstName	MiddleName	LastName	Department	Salary
1	John	NULL	Doe	Sales	50000.00
2	Jane	A.	Smith	Marketing	NULL
3	NULL	NULL	Taylor	HR	45000.00
4	Emily	B.	NULL	Finance	60000.00
5	James	NULL	Brown	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL

YEEDETAILS 1 ×

RETRIEVE FULL NAME OF EMPLOYEES

```
22
23      -- RETRIEVE FULL NAME OF EMPLOYEES
24
25 •        SELECT EmployeeID, COALESCE(FirstName, MiddleName, LastName) AS FullName FROM EmployeeDetails;
26
27          -- This query will return the first non-null value among FirstName, MiddleName, and LastName for each employee.
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

EmployeeID	FullName
1	John
2	Jane
3	Taylor
4	Emily
5	James

Result 5

RETRIEVE SALARY WITH DEFAULT VALUE ‘0’

```
29      -- Retrieve Salary with Default Value
30
31 •        SELECT EmployeeID, COALESCE(Salary, 0) AS Salary
32          FROM EmployeeDetails;
33
34          -- This query will replace null salaries with 0.
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

EmployeeID	Salary
1	50000.00
2	0.00
3	45000.00
4	60000.00
5	0.00

Result 2

CONCATENATE FULL NAME OF EMPLOYEES

```sql
-- Concatenate Full Name

SELECT EmployeeID,
    COALESCE(FirstName, '') AS FirstName,
    COALESCE(MiddleName, '') AS MiddleName,
    COALESCE(LastName, '') AS LastName,
    CONCAT(COALESCE(FirstName, ''), ' ', COALESCE(MiddleName, ''), ' ', COALESCE(LastName, '')) AS FullName
FROM EmployeeDetails;

-- This query will concatenate the first name, middle name, and last name into a full name string, handling null values properly.
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

EmployeeID	FirstName	MiddleName	LastName	FullName
1	John		Doe	John Doe
2	Jane	A.	Smith	Jane A. Smith
3			Taylor	Taylor
4	Emily	B.		Emily B.
5	James		Brown	James Brown

Result 3

PROVIDE A DEFAULT VALUE FOR DEPARTMENT

```sql
-- Provide a Default value for Department

SELECT
    EmployeeID,
    COALESCE(Department, 'Not Assigned') AS Department
FROM
    EmployeeDetails;

-- This query will provide 'Not Assigned' for employees without a department.
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

EmployeeID	Department
1	Sales
2	Marketing
3	HR
4	Finance
5	Not Assigned

Result 1