APNA COLLEGE

SQL

(Notes by Apna College)

What is Database?
Database is a collection of interrelated data.

What is DBMS?
DBMS (*Database Management System*) is software used to create, manage, and organize databases.

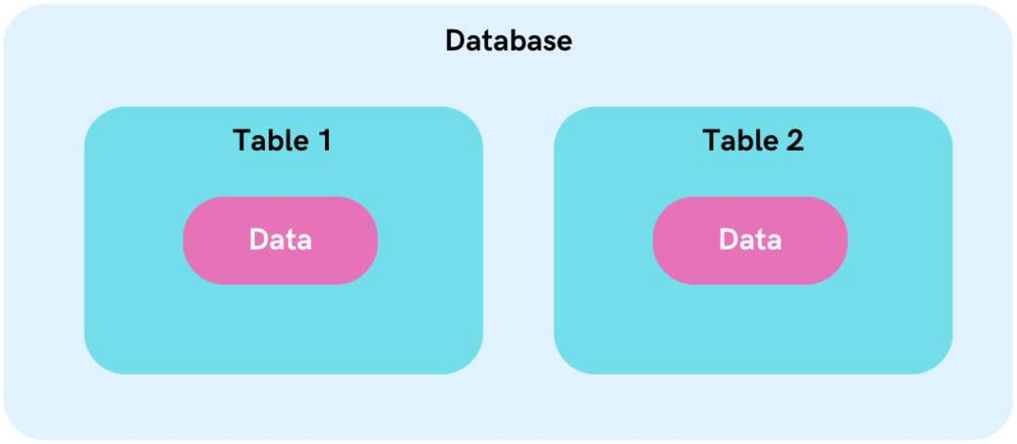
What is RDBMS?
- RDBMS (Relational *Database Management System)* - is a DBMS based on the concept of tables (also called relations).
- Data is organized into tables (also known as relations) with rows (records) and columns (attributes).
- Eg - MySQL, PostgreSQL, Oracle etc.

What is SQL?
SQL is *Structured Query Language* - used to store, manipulate and retrieve data from RDBMS.
(It is not a database, it is a language used to interact with database)

We use SQL for *CRUD* Operations :
- **CREATE** - To create databases, tables, insert tuples in tables etc
- **READ** - To read data present in the database.
- **UPDATE** - Modify already inserted data.
- **DELETE** - Delete database, table or specific data point/tuple/row or multiple rows.

*Note - SQL keywords are NOT case sensitive. Eg: select is the same as SELECT in SQL.

SQL v/s MySQL
SQL is a language used to perform CRUD operations in Relational DB, while MySQL is a RDBMS that uses SQL.



SQL Data Types
In SQL, data types define the kind of data that can be stored in a column or variable.

To See all data types of MYSQL, visit :
https://dev.mysql.com/doc/refman/8.0/en/data-types.html

Here are the frequently used SQL data types:

DATATYPE	DESCRIPTION	USAGE
CHAR	string(0-255), can store characters of fixed length	CHAR(50)
VARCHAR	string(0-255), can store characters up to given length	VARCHAR(50)
BLOB	string(0-65535), can store binary large object	BLOB(1000)
INT	integer(-2,147,483,648 to 2,147,483,647)	INT
TINYINT	integer(-128 to 127)	TINYINT
BIGINT	integer(-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)	BIGINT
BIT	can store x-bit values. x can range from 1 to 64	BIT(2)
FLOAT	Decimal number - with precision to 23 digits	FLOAT
DOUBLE	Decimal number - with 24 to 53 digits	DOUBLE
BOOLEAN	Boolean values 0 or 1	BOOLEAN
DATE	date in format of YYYY-MM-DD ranging from 1000-01-01 to 9999-12-31	DATE
TIME	HH:MM:SS	TIME
YEAR	year in 4 digits format ranging from 1901 to 2155	YEAR

*Note - CHAR is for fixed length & VARCHAR is for variable length strings. Generally, VARCHAR is better as it only occupies necessary memory & works more efficiently.

We can also use UNSIGNED with datatypes when we only have positive values to add.
Eg - UNSIGNED INT

Types of SQL Commands:

1. **DQL** *(Data Query Language)* : Used to retrieve data from databases. (SELECT)

2. **DDL** *(Data Definition Language)* : Used to create, alter, and delete database objects like tables, indexes, etc. (CREATE, DROP, ALTER, RENAME, TRUNCATE)

3. **DML** *(Data Manipulation Language)*: Used to modify the database. (INSERT, UPDATE, DELETE)

4. **DCL** *(Data Control Language)*: Used to grant & revoke permissions. (GRANT, REVOKE)

5. **TCL** *(Transaction Control Language)*: Used to manage transactions. (COMMIT, ROLLBACK, START TRANSACTIONS, SAVEPOINT)

---

1. Data Definition Language (DDL)

Data Definition Language (DDL) is a subset of SQL (Structured Query Language) responsible for defining and managing the structure of databases and their objects.

DDL commands enable you to create, modify, and delete database objects like tables, indexes, constraints, and more.

Key DDL Commands are:

- **CREATE TABLE:**
  - Used to create a new table in the database.
  - Specifies the table name, column names, data types, constraints, and more.
  - Example:
    CREATE TABLE employees (id INT PRIMARY KEY, name VARCHAR(50), salary DECIMAL(10, 2));

- **ALTER TABLE:**
  - Used to modify the structure of an existing table.
  - You can add, modify, or drop columns, constraints, and more.
  - Example: ALTER TABLE employees ADD COLUMN email VARCHAR(100);

- **DROP TABLE:**
  - Used to delete an existing table along with its data and structure.
  - Example: DROP TABLE employees;

- **CREATE INDEX:**
    - Used to create an index on one or more columns in a table.
    - Improves query performance by enabling faster data retrieval.
    - Example: CREATE INDEX idx_employee_name ON employees (name);

- **DROP INDEX:**
    - Used to remove an existing index from a table.
    - Example: DROP INDEX idx_employee_name;

- **CREATE CONSTRAINT:**
    - Used to define constraints that ensure data integrity.
    - Constraints include PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, and CHECK.
    - Example: ALTER TABLE orders ADD CONSTRAINT fk_customer FOREIGN KEY (customer_id) REFERENCES customers(id);

- **DROP CONSTRAINT:**
    - Used to remove an existing constraint from a table.
    - Example: ALTER TABLE orders DROP CONSTRAINT fk_customer;

- **TRUNCATE TABLE:**
    - Used to delete the data inside a table, but not the table itself.
    - Syntax – TRUNCATE TABLE table_name

2. DATA QUERY/RETRIEVAL LANGUAGE (DQL or DRL)

DQL (Data Query Language) is a subset of SQL focused on retrieving data from databases.

The SELECT statement is the foundation of DQL and allows us to extract specific columns from a table.

- **SELECT:**

The SELECT statement is used to select data from a database.

Syntax: SELECT column1, column2, ... FROM table_name;

Here, column1, column2, ... are the field names of the table.

If you want to select all the fields available in the table, use the following syntax:
SELECT * FROM table_name;

Ex: SELECT CustomerName, City FROM Customers;

- **WHERE:**

The WHERE clause is used to filter records.

Syntax: SELECT column1, column2, ... FROM table_name WHERE condition;

Ex: SELECT * FROM Customers WHERE Country='Mexico';

Operators used in WHERE are:

=   : Equal
>   : Greater than
<   : Less than
>= : Greater than or equal
<= : Less than or equal
<> : Not equal.

Note: In some versions of SQL this operator may be written as !=

- **AND, OR and NOT:**

- The WHERE clause can be combined with AND, OR, and NOT operators.

- The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND are TRUE.

- The OR operator displays a record if any of the conditions separated by OR is TRUE.

- The NOT operator displays a record if the condition(s) is NOT TRUE.

Syntax:

SELECT column1, column2, ... FROM table_name WHERE condition1 AND condition2 AND condition3 ...;

SELECT column1, column2, ... FROM table_name WHERE condition1 OR condition2 OR condition3 ...;

SELECT column1, column2, ... FROM table_name WHERE NOT condition;

Example:

SELECT * FROM Customers WHERE Country='India' AND City='Japan';

SELECT * FROM Customers WHERE Country='America' AND (City='India' OR City='Korea');

- **DISTINCT:**

Removes duplicate rows from query results.

Syntax: SELECT DISTINCT column1, column2 FROM table_name;

- **LIKE**:

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character

Example: SELECT * FROM employees WHERE first_name LIKE 'J%';

WHERE CustomerName LIKE 'a%'
- Finds any values that start with "a"

WHERE CustomerName LIKE '%a'
- Finds any values that end with "a"

WHERE CustomerName LIKE '%or%'
- Finds any values that have "or" in any position

WHERE CustomerName LIKE '_r%'
- Finds any values that have "r" in the second position

WHERE CustomerName LIKE 'a_%'

- Finds any values that start with "a" and are at least 2 characters in length

WHERE CustomerName LIKE 'a__%'

- Finds any values that start with "a" and are at least 3 characters in length

WHERE ContactName LIKE 'a%o'

- Finds any values that start with "a" and ends with "o"

- **IN:**

Filters results based on a list of values in the WHERE clause.

Example: SELECT * FROM products WHERE category_id IN (1, 2, 3);

- **BETWEEN**:

Filters results within a specified range in the WHERE clause.

Example: SELECT * FROM orders WHERE order_date BETWEEN '2023-01-01' AND '2023-06-30';

- **IS NULL**:

Checks for NULL values in the WHERE clause.

Example: SELECT * FROM customers WHERE email IS NULL;

- **AS**:

Renames columns or expressions in query results.

Example: SELECT first_name AS "First Name", last_name AS "Last Name" FROM employees;

- **ORDER BY**

The ORDER BY clause allows you to sort the result set of a query based on one or more columns.

Basic Syntax:

- The ORDER BY clause is used after the SELECT statement to sort query results.

- Syntax: SELECT column1, column2 FROM table_name ORDER BY column1 [ASC|DESC];

Ascending and Descending Order:

- By default, the ORDER BY clause sorts in ascending order (smallest to largest).
- You can explicitly specify descending order using the DESC keyword.
- Example: SELECT product_name, price FROM products ORDER BY price DESC;

Sorting by Multiple Columns:

- You can sort by multiple columns by listing them sequentially in the ORDER BY clause.
- Rows are first sorted based on the first column, and for rows with equal values, subsequent columns are used for further sorting.
- Example: SELECT first_name, last_name FROM employees ORDER BY last_name, first_name;

Sorting by Expressions:

- It's possible to sort by calculated expressions, not just column values.
- Example: SELECT product_name, price, price * 1.1 AS discounted_price FROM products ORDER BY discounted_price;

Sorting NULL Values:

- By default, NULL values are considered the smallest in ascending order and the largest in descending order.
- You can control the sorting behaviour of NULL values using the NULLS FIRST or NULLS LAST options.
- Example: SELECT column_name FROM table_name ORDER BY column_name NULLS LAST;

Sorting by Position:

- Instead of specifying column names, you can sort by column positions in the ORDER BY clause.
- Example: SELECT product_name, price FROM products ORDER BY 2 DESC, 1 ASC;

- **<u>GROUP BY</u>**

The GROUP BY clause in SQL is used to group rows from a table based on one or more columns.

Syntax:

- The GROUP BY clause follows the SELECT statement and is used to group rows based on specified columns.

- Syntax: SELECT column1, aggregate_function(column2) FROM table_name GROUP BY column1;

- Aggregation Functions:
    - Aggregation functions (e.g., COUNT, SUM, AVG, MAX, MIN) are often used with GROUP BY to calculate values for each group.
    - Example: SELECT department, AVG(salary) FROM employees GROUP BY department;
- Grouping by Multiple Columns:
    - You can group by multiple columns by listing them in the GROUP BY clause.
    - This creates a hierarchical grouping based on the specified columns.
    - Example: SELECT department, gender, AVG(salary) FROM employees GROUP BY department, gender;

- HAVING Clause:
    - The HAVING clause is used with GROUP BY to filter groups based on aggregate function results.
    - It's similar to the WHERE clause but operates on grouped data.
    - Example: SELECT department, AVG(salary) FROM employees GROUP BY department HAVING AVG(salary) > 50000;

- Combining GROUP BY and ORDER BY:
    - You can use both GROUP BY and ORDER BY in the same query to control the order of grouped results.
    - Example: SELECT department, COUNT(*) FROM employees GROUP BY department ORDER BY COUNT(*) DESC;

- **<u>AGGREGATE FUNCTIONS</u>**

These are used to perform calculations on groups of rows or entire result sets. They provide insights into data by summarising and processing information.

Common Aggregate Functions:

- COUNT():
  Counts the number of rows in a group or result set.

- SUM():
  Calculates the sum of numeric values in a group or result set.

- AVG():

Computes the average of numeric values in a group or result set.

- MAX():
  Finds the maximum value in a group or result set.

- MIN():
  Retrieves the minimum value in a group or result set.

3. DATA MANIPULATION LANGUAGE

Data Manipulation Language (DML) in SQL encompasses commands that manipulate data within a database. DML allows you to insert, update, and delete records, ensuring the accuracy and currency of your data.

- **INSERT:**

- The INSERT statement adds new records to a table.
- Syntax: INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);

- Example: INSERT INTO employees (first_name, last_name, salary) VALUES ('John', 'Doe', 50000);

- **UPDATE**:

- The UPDATE statement modifies existing records in a table.
- Syntax: UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;
- Example: UPDATE employees SET salary = 55000 WHERE first_name = 'John';

- **DELETE**:

- The DELETE statement removes records from a table.
- Syntax: DELETE FROM table_name WHERE condition;
- Example: DELETE FROM employees WHERE last_name = 'Doe';

4. Data Control Language (DCL)

Data Control Language focuses on the management of access rights, permissions, and security-related aspects of a database system.

DCL commands are used to control who can access the data, modify the data, or perform administrative tasks within a database.

DCL is an important aspect of database security, ensuring that data remains protected and only authorised users have the necessary privileges.

There are two main DCL commands in SQL: GRANT and REVOKE.

1. GRANT:

The GRANT command is used to provide specific privileges or permissions to users or roles. Privileges can include the ability to perform various actions on tables, views, procedures, and other database objects.

Syntax:

```
GRANT privilege_type
ON object_name
TO user_or_role;
```

In this syntax:

- privilege_type refers to the specific privilege or permission being granted (e.g., SELECT, INSERT, UPDATE, DELETE).
- object_name is the name of the database object (e.g., table, view) to which the privilege is being granted.
- user_or_role is the name of the user or role that is being granted the privilege.

Example: Granting SELECT privilege on a table named "Employees" to a user named "Analyst":

```
GRANT SELECT ON Employees TO Analyst;
```

2. REVOKE:

The REVOKE command is used to remove or revoke specific privileges or permissions that have been previously granted to users or roles.

Syntax:

```
REVOKE privilege_type
ON object_name
```

FROM user_or_role;

In this syntax:

- privilege_type is the privilege or permission being revoked.
- object_name is the name of the database object from which the privilege is being revoked.
- user_or_role is the name of the user or role from which the privilege is being revoked.

Example: Revoking the SELECT privilege on the "Employees" table from the "Analyst" user:

REVOKE SELECT ON Employees FROM Analyst;

DCL and Database Security:

DCL plays a crucial role in ensuring the security and integrity of a database system.

By controlling access and permissions, DCL helps prevent unauthorised users from tampering with or accessing sensitive data. Proper use of GRANT and REVOKE commands ensures that only users who require specific privileges can perform certain actions on database objects.

5. Transaction Control Language (TCL)

Transaction Control Language (TCL) deals with the management of transactions within a database.
TCL commands are used to control the initiation, execution, and termination of transactions, which are sequences of one or more SQL statements that are executed as a single unit of work.
Transactions ensure data consistency, integrity, and reliability in a database by grouping related operations together and either committing or rolling back changes based on the success or failure of those operations.

There are three main TCL commands in SQL: COMMIT, ROLLBACK, and SAVEPOINT.

1. COMMIT:

The COMMIT command is used to permanently save the changes made during a transaction.

It makes all the changes applied to the database since the last COMMIT or ROLLBACK command permanent.
Once a COMMIT is executed, the transaction is considered successful, and the changes are made permanent.

Example: Committing changes made during a transaction:

```
UPDATE Employees
SET Salary = Salary * 1.10
WHERE Department = 'Sales';

COMMIT;
```

2. ROLLBACK:

The ROLLBACK command is used to undo changes made during a transaction.
It reverts all the changes applied to the database since the transaction began.

ROLLBACK is typically used when an error occurs during the execution of a transaction, ensuring that the database remains in a consistent state.

Example: Rolling back changes due to an error during a transaction:

```
BEGIN;

UPDATE Inventory
SET Quantity = Quantity - 10
WHERE ProductID = 101;

-- An error occurs here

ROLLBACK;
```

3. SAVEPOINT:

The SAVEPOINT command creates a named point within a transaction, allowing you to set a point to which you can later ROLLBACK if needed.

SAVEPOINTs are useful when you want to undo part of a transaction while preserving other changes.

Syntax: SAVEPOINT savepoint_name;

Example: Using SAVEPOINT to create a point within a transaction:

```
BEGIN;
```

```
UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 123;

SAVEPOINT before_withdrawal;

UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 456;

-- An error occurs here

ROLLBACK TO before_withdrawal;

-- The first update is still applied

COMMIT;
```

TCL and Transaction Management:

Transaction Control Language (TCL) commands are vital for managing the integrity and consistency of a database's data.
They allow you to group related changes into transactions, and in the event of errors, either commit those changes or roll them back to maintain data integrity.
TCL commands are used in combination with Data Manipulation Language (DML) and other SQL commands to ensure that the database remains in a reliable state despite unforeseen errors or issues.

JOINS

In a DBMS, a join is an operation that combines rows from two or more tables based on a related column between them.
Joins are used to retrieve data from multiple tables by linking them together using a common key or column.

Types of Joins:

1. Inner Join
2. Outer Join
3. Cross Join
4. Self Join

1) Inner Join

An inner join combines data from two or more tables based on a specified condition, known as the join condition.
The result of an inner join includes only the rows where the join condition is met in all participating tables.
It essentially filters out non-matching rows and returns only the rows that have matching values in both tables.

Syntax:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Here:

- columns refers to the specific columns you want to retrieve from the tables.
- table1 and table2 are the names of the tables you are joining.
- column is the common column used to match rows between the tables.
- The ON clause specifies the join condition, where you define how the tables are related.

Example: Consider two tables: Customers and Orders.

Customers Table:

CustomerID	CustomerName
1	Alice
2	Bob
3	Carol

Orders Table:

OrderID	CustomerID	Product

101	1	Laptop
102	3	Smartphone
103	2	Headphones

Inner Join Query:

```
SELECT Customers.CustomerName, Orders.Product
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Result:

CustomerName	Product
Alice	Laptop
Bob	Headphones
Carol	Smartphone

2) Outer Join

Outer joins combine data from two or more tables based on a specified condition, just like inner joins. However, unlike inner joins, outer joins also include rows that do not have matching values in both tables.
Outer joins are particularly useful when you want to include data from one table even if there is no corresponding match in the other table.

Types:

There are three types of outer joins: left outer join, right outer join, and full outer join.

1. Left Outer Join (Left Join):

A left outer join returns all the rows from the left table and the matching rows from the right table.

If there is no match in the right table, the result will still include the left table's row with NULL values in the right table's columns.

Example:

```
SELECT Customers.CustomerName, Orders.Product
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Result:

CustomerName	Product
Alice	Laptop
Bob	Headphones
Carol	Smartphone
NULL	Monitor

In this example, the left outer join includes all rows from the Customers table.

Since there is no matching customer for the order with OrderID 103 (Monitor), the result includes a row with NULL values in the CustomerName column.

2. Right Outer Join (Right Join):

A right outer join is similar to a left outer join, but it returns all rows from the right table and the matching rows from the left table.

If there is no match in the left table, the result will still include the right table's row with NULL values in the left table's columns.

Example: Using the same Customers and Orders tables.

```
SELECT Customers.CustomerName, Orders.Product
FROM Customers
RIGHT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Result:

CustomerName	Product
Alice	Laptop
Carol	Smartphone
Bob	Headphones
NULL	Keyboard

Here, the right outer join includes all rows from the Orders table. Since there is no matching order for the customer with CustomerID 4, the result includes a row with NULL values in the CustomerName column.

3. Full Outer Join (Full Join):

A full outer join returns all rows from both the left and right tables, including matches and non-matches.

If there's no match, NULL values appear in columns from the table where there's no corresponding value.

Example: Using the same Customers and Orders tables.

```
SELECT Customers.CustomerName, Orders.Product
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Result:

CustomerName	Product
Alice	Laptop
Bob	Headphones
Carol	Smartphone

NULL	Monitor
NULL	Keyboard

In this full outer join example, all rows from both tables are included in the result. Both non-matching rows from the Customers and Orders tables are represented with NULL values.

3) **Cross Join**

A cross join, also known as a Cartesian product, is a type of join operation in a Database Management System (DBMS) that combines every row from one table with every row from another table.

Unlike other join types, a cross join does not require a specific condition to match rows between the tables. Instead, it generates a result set that contains all possible combinations of rows from both tables.

Cross joins can lead to a large result set, especially when the participating tables have many rows.

Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

In this syntax:

- columns refers to the specific columns you want to retrieve from the cross-joined tables.
- table1 and table2 are the names of the tables you want to combine using a cross join.

Example: Consider two tables: Students and Courses.

Students Table:

StudentID	StudentName
1	Alice

2	Bob

Courses Table:

CourseID	CourseName
101	Maths
102	Science

Cross Join Query:

```
SELECT Students.StudentName, Courses.CourseName
FROM Students
CROSS JOIN Courses;
```

Result:

StudentName	CourseName
Alice	Maths
Alice	Science
Bob	Maths
Bob	Science

In this example, the cross join between the Students and Courses tables generates all possible combinations of rows from both tables. As a result, each student is paired with each course, leading to a total of four rows in the result set.

4) Self Join

A self join involves joining a table with itself.

This technique is useful when a table contains hierarchical or related data and you need to compare or analyse rows within the same table.

Self joins are commonly used to find relationships, hierarchies, or patterns within a single table.

In a self join, you treat the table as if it were two separate tables, referring to them with different aliases.

Syntax:

The syntax for performing a self join in SQL is as follows:

```
SELECT columns
FROM table1 AS alias1
JOIN table1 AS alias2 ON alias1.column = alias2.column;
```

In this syntax:

- columns refers to the specific columns you want to retrieve from the self-joined table.
- table1 is the name of the table you're joining with itself.
- alias1 and alias2 are aliases you assign to the table instances for differentiation.
- column is the column you use as the join condition to link rows from the same table.

Example: Consider an Employees table that contains information about employees and their managers.

Employees Table:

EmployeeID	EmployeeName	ManagerID
1	Alice	3
2	Bob	3
3	Carol	NULL
4	David	1

Self Join Query:

```
SELECT e1.EmployeeName AS Employee, e2.EmployeeName AS Manager
FROM Employees AS e1
JOIN Employees AS e2 ON e1.ManagerID = e2.EmployeeID;
```

Result:

Employee	Manager

Alice	Carol
Bob	Carol
David	Alice

In this example, the self join is performed on the Employees table to find the relationship between employees and their managers. The join condition connects the ManagerID column in the e1 alias (representing employees) with the EmployeeID column in the e2 alias (representing managers).

SET OPERATIONS

Set operations in SQL are used to combine or manipulate the result sets of multiple SELECT queries.
They allow you to perform operations similar to those in set theory, such as union, intersection, and difference, on the data retrieved from different tables or queries.

Set operations provide powerful tools for managing and manipulating data, enabling you to analyse and combine information in various ways.

There are four primary set operations in SQL:

- UNION
- INTERSECT
- EXCEPT (or MINUS)
- UNION ALL

1. UNION:

The UNION operator combines the result sets of two or more SELECT queries into a single result set.
It removes duplicates by default, meaning that if there are identical rows in the result sets, only one instance of each row will appear in the final result.

Example:

Assume we have two tables: Customers and Suppliers.

Customers Table:

CustomerID	CustomerName
1	Alice
2	Bob

Suppliers Table:

SupplierID	SupplierName
101	SupplierA
102	SupplierB

UNION Query:

```
SELECT CustomerName FROM Customers
UNION
SELECT SupplierName FROM Suppliers;
```

Result:

CustomerName
Alice
Bob
SupplierA
SupplierB

2. INTERSECT:

The INTERSECT operator returns the common rows that exist in the result sets of two or more SELECT queries.

It only returns distinct rows that appear in all result sets.

Example: Using the same tables as before.

```
SELECT CustomerName FROM Customers
INTERSECT
SELECT SupplierName FROM Suppliers;
```

Result:

CustomerName

In this example, there are no common names between customers and suppliers, so the result is an empty set.

3. EXCEPT (or MINUS):

The EXCEPT operator (also known as MINUS in some databases) returns the distinct rows that are present in the result set of the first SELECT query but not in the result set of the second SELECT query.

Example: Using the same tables as before.

```
SELECT CustomerName FROM Customers
EXCEPT
SELECT SupplierName FROM Suppliers;
```

Result:

CustomerName
Alice
Bob

In this example, the names "Alice" and "Bob" are customers but not suppliers, so they appear in the result set.

4. UNION ALL:

The UNION ALL operator performs the same function as the UNION operator but does not remove duplicates from the result set. It simply concatenates all rows from the different result sets.

Example: Using the same tables as before.

```
SELECT CustomerName FROM Customers
UNION ALL
SELECT SupplierName FROM Suppliers;
```

Result:

CustomerName
Alice
Bob
SupplierA
SupplierB

Difference between Set Operations and Joins

Aspect	**Set Operations**	**Joins**
Purpose	Manipulate result sets based on set theory principles.	Combine data from related tables based on specified conditions.

Data Source	Result sets of SELECT queries.	Tables that are related by common columns.
Combining Rows	Combine rows from different result sets. May remove duplicates.	Combine rows from different tables based on specified conditions.
Output Columns	Require the SELECT queries to have the same number of output columns and compatible data types.	Can combine columns from different tables, regardless of data types or column numbers.
Common Operations	UNION, INTERSECT, EXCEPT (MINUS).	INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN.
Conditional Requirements	No specific join conditions are required.	Require specified join conditions for combining data.
Handling Duplicates	UNION removes duplicates by default.	Joins do not inherently handle duplicates; it depends on the join type and data.

Usage Scenarios	Useful for combining and analysing related data from different queries or tables.	Used to retrieve and relate data from different tables based on their relationships.
Result Set Structure	Result sets may have different column names, but data types and counts must match.	Result sets can have different column names, data types, and counts.
Performance Considerations	Generally faster and less complex than joins.	Joins can be more complex and resource-intensive, especially for larger datasets.

SUB QUERIES

Subqueries, also known as nested queries or inner queries, allow you to use the result of one query (the inner query) as the input for another query (the outer query).

Subqueries are often used to retrieve data that will be used for filtering, comparison, or calculation within the context of a larger query.

They are a way to break down complex tasks into smaller, manageable steps.

Syntax:

```
SELECT columns
```

```
FROM table
WHERE column OPERATOR (SELECT column FROM table WHERE condition);
```

In this syntax:

- columns refers to the specific columns you want to retrieve from the outer query.
- table is the name of the table you're querying.
- column is the column you're applying the operator to in the outer query.
- OPERATOR is a comparison operator such as =, >, <, IN, NOT IN, etc.
- (SELECT column FROM table WHERE condition) is the subquery that provides the input for the comparison.

Example: Consider two tables: Products and Orders.

Products Table:

ProductID	ProductName	Price
1	Laptop	1000
2	Smartphone	500
3	Headphones	50

Orders Table:

OrderID	ProductID	Quantity
101	1	2
102	3	1

For Example: Retrieve the product names and quantities for orders with a total cost greater than the average price of all products.

```
SELECT ProductName, Quantity
FROM Products
WHERE Price * Quantity > (SELECT AVG(Price) FROM Products);
```

Result:

ProductName	Quantity

Laptop	2

Differences Between Subqueries and Joins:

Aspect	Subqueries	Joins
Purpose	Retrieve data for filtering, comparison, or calculation within the context of a larger query.	Combine data from related tables based on specified conditions.
Data Source	Result of one query used as input for another query.	Data from multiple related tables.
Combining Rows	Not used for combining rows; used to filter or evaluate data.	Combines rows from different tables based on specified join conditions.
Result Set Structure	Subqueries return scalar values, single-column results, or small result sets.	Joins return multi-column result sets.
Performance Considerations	Subqueries can be slower and less efficient, especially when dealing with large datasets.	Joins can be more efficient for combining data from multiple tables.
Complexity	Subqueries can be easier to understand for simple tasks or smaller datasets.	Joins can become complex, but are more suited for handling large-scale data retrieval and combination tasks.
Versatility	Subqueries can be used in various clauses: WHERE, FROM, HAVING, etc.	Joins are primarily used in the FROM clause for combining tables.