SQL, which stands for Structured Query Language, is a programming language that is used to communicate with and manage databases. SQL is a standard language for manipulating data held in relational database management systems (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It was first developed in the 1970s by IBM.

Operators in Sql server:

**Arithmetic Operators**: These are used to perform mathematical operations. Here is a list of these operators:

* + : Addition
* - : Subtraction
* \* : Multiplication
* / : Division
* % : Modulus

**Comparison Operators**: These are used in the where clause to compare one expression with another. Some of these operators are:

* = : Equal
* != or <> : Not equal
* > : Greater than
* < : Less than
* >=: Greater than or equal
* <=: Less than or equal

**Logical Operators**: They are used to combine the result set of two different component conditions. These include:

* AND: Returns true if both components are true.
* OR : Returns true if any one of the component is true.
* NOT: Returns the opposite boolean value of the condition.

**Bitwise Operators**: These perform bit-level operations on the inputs. Here is a list of these operators:

* & : Bitwise AND
* | : Bitwise OR
* ^ : Bitwise XOR

**Data Definition Language (DDL)**

Data Definition Language (DDL) is a subset of SQL. Its primary function is to create, modify, and delete database structures but not data. The commands in DDL are:

1.create

2.Drop

3.Alter

4.Rename

5.truncate

1.

**Create Table**

syntax

Create Table Tablename(

column1name DataType,

column2 and dataType,

..

..

)

Ex:

Create TAble Table1 (

Id VARCHAR(500),

Column1 VARCHAR(200),

Column2 VArchar(300)

)

**Create table with not Null constrain and unique with name**

CREATE TABLE Employees (

ID int NOT NULL,

Name varchar(255) NOT NULL,

Salary int,

Department varchar(255),

Position varchar(255));

Create Table tablename

(column1 varchar Constraint constrintName constraintType)

**2. Drop table**

**--Syntax**

**Drop Table Tablename**

**Ex:**

**Drop Table Table1**

**3. Alter:**

The ALTER TABLE command in SQL is used to add, delete/drop, or modify columns in an existing table. It’s also useful for adding and dropping constraints such as primary key, foreign key, etc.

**Add column in existing table:**

Syntax:

Alter table tablename

Add CloumnName DataType:

Ex

alter table Table1

Add Column3 VARCHAR(200);

**Alter Table:**

**Alter, Delete column from Existing table.**

**Syntax:**

**For alter data type:**

Alter table tableName

Alter Column ColumnName DataType.

**Drop Column:**

Alter table tableName

Drop Column ColumnName

**Add and Drop constraint for particular existing column.**

**Add constraint;**

ALTER TABLE TABLENAME  
ADD Constraint constraintName ConstrintType (columnName)

Ex:

ALTER TABLE tableName

ADD CONSTRAINT constraintName

PRIMARY KEY **(column1, column2, ... column\_n);**

**Drop constraint:**

Alter table TableName

Drop constraint constraintName

**ReName column:**

EXEC sp\_rename 'tablename.ColunName', 'new\_column\_name', 'COLUMN';

**Truncate table:**

Truncate table tablename

# Data Manipulation Language (DML)

DML is a subcategory of **SQL** which stands for Data Manipulation Language. The purpose of DML is to **insert, retrieve, update and delete data from the database**. With this, we can perform operations on existing records.

1.INSERT

2.DELETE FROM

3.SELECT

4.UPDATE

**1. INSERT:**

**Syntax:**

INSERT INTO table\_name ( column1, column2, column3, ... )

VALUES ( value1, value2, value3, ... )

**2.DeleteFrom**

DELETE FROM table\_name WHERE condition;

EX:

DELETE FROM Table2 WHERE Column1 = 'fdgfdgdfg'

# SELECT

The **SELECT** statement in SQL is majorly used for fetching data from the database. It is one of the most essential elements of SQL.

## Syntax

Here’s how your **SELECT** command will look like:

SELECT column1, column2, ...

FROM table\_name;

If you want to select all the columns of a table, you can use **\*** like this:

SELECT \* FROM table\_name;

## Example

For instance, consider we have a table **EMPLOYEES** with columns **name**, **designation**, and **salary**. We can use **SELECT** in the following way:

SELECT name, designation FROM EMPLOYEES;

This will retrieve all the names and designations of all employees from the table **EMPLOYEES**.

## SELECT DISTINCT

The **SELECT DISTINCT** statement is used to return only distinct (different) values. The DISTINCT keyword eliminates duplicate records from the results.

Here’s how you can use it:

SELECT DISTINCT column1, column2, ...

FROM table\_name;

For example, if we want to select all unique designations from the **EMPLOYEES** table, the query will look like this:

SELECT DISTINCT designation FROM EMPLOYEES;

## SELECT WHERE

**SELECT** statement combined with **WHERE** gives us the ability to filter records based on a condition.

Syntax:

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

For example, to select employees with salary more than 50000, you can use this query:

SELECT \* FROM EMPLOYEES WHERE salary > 50000;

## SELECT ORDER BY

Using **SELECT** statement in conjunction with **ORDER BY**, we can sort the result-set in ascending or descending order.

Syntax:

SELECT column1, column2, ...

FROM table\_name

ORDER BY column ASC|DESC;

For example, to select all employees and order them by their name in ascending fashion:

SELECT \* FROM EMPLOYEES ORDER BY name ASC;

Remember that the default sort order is ascending if the ASC|DESC parameter is not defined.

SELECT \* FROM Table1,Table2 WHERE Table1.Column2 = Table2.Column2 ORDER by Table1.Column2,Table2.Column1

ORDER BY clause, the sorting will first be done based on column1, and if there are any ties, then it will be further sorted based on column2, and so on.

# FROM

The **FROM** clause in SQL specifies the tables from which the retrieval should be made. It is an integral part of **SELECT** statements and variants of **SELECT** like **SELECT INTO** and **SELECT WHERE**. **FROM** can be used to join tables as well.

Typically, **FROM** is followed by space delimited list of tables in which the SELECT operation is to be executed. If you need to pull data from multiple tables, you would separate each table with a comma.

Here are some examples:

**Example 1 - Simple Usage**

If you’ve a table called **employees**, you can select all employees’ data like this:

SELECT \*

FROM employees;

In this example, **\*** means “all columns”. So, **SELECT \* FROM employees;** will retrieve all data from the **employees** table.

**Example 2 - FROM with Multiple Tables**

If you’ve multiple tables, say **employees** and **departments**, and you want to select data from both, you can do the following:

SELECT employees.name, departments.department

FROM employees, departments

WHERE employees.dept\_id = departments.dept\_id;

In this example, the **FROM** clause is following by two tables: **employees** and **departments**. **employees.name** and **departments.department** indicate that we’re selecting the **name** column from the **employees** table and the **department** column from the **departments** table.

Remember, always respect the order of operations in SQL. The **FROM** clause works only after tables are identified.

In complex SQL queries where you might need to pull data from multiple tables, aliases are used to temporarily rename the tables within the individual SQL statement.

**Example 3 - FROM with Aliases**

Below is an example of a **FROM** clause with aliases:

SELECT e.name, d.department

FROM employees AS e, departments AS d

WHERE e.dept\_id = d.dept\_id;

In this example, **employees** and **departments** tables are termed as **e** and **d** respectively.

That’s it! Remember that **FROM** is not limited only to **SELECT**. It is applicable to **UPDATE** and **DELETE** operations as well.

**GROUP BY:**

Group By” is a clause in SQL that is used to arrange identical data into groups. This clause comes under the category of Group Functions, alongside the likes of Count, Sum, Average, etc.

The syntax for ‘Group by’ is:

SELECT column1, column2

FROM table\_name

GROUP BY column1, column2;

Here, column1, column2, are the names of the columns based on which we want to group the results.

## Example:

Assume we have a “Sales” table. This table has three columns: ID, Item, and Amount.

ID Item Amount

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

1 A 150

2 B 200

3 A 100

4 B 50

5 A 200

6 A 100

7 B 150

Execute the following SQL statement…

SELECT Item, SUM(Amount)

FROM Sales

GROUP BY Item;

This will concatenate, or “group”, all items that are the same into one row, applying the SUM() function on their respective Amounts. The output will then be:

Item SUM(Amount)

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

A 550

B 400

## Group By with Having Clause

The Group By clause can also be used with the Having keyword. The Having keyword allows you to filter the results of the group function.

For example:

SELECT Item, SUM(Amount)

FROM Sales

GROUP BY Item

HAVING SUM(Amount) > 150;

This will return all grouped items where the total amount is more than 150. Hence, the result will be:

Item SUM(Amount)

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

A 550

B 400

**Windows Function:**

**Row\_Number()**

**After using ROW\_NUMBER() in SQL Server, you can use several keywords and clauses to further manipulate or filter the result set. Here are some common keywords and clauses that you might use after ROW\_NUMBER():**

**PARTITION BY: This clause is used with ROW\_NUMBER() to divide the result set into partitions based on the values of one or more columns. Each partition is numbered separately. For example:**

**sql**

**Copy code**

**SELECT employee\_id, employee\_name,**

**ROW\_NUMBER() OVER (PARTITION BY department ORDER BY employee\_id) AS row\_num**

**FROM employees;**

**ORDER BY: This clause is used with ROW\_NUMBER() to specify the order in which the row numbers are assigned within each partition. For example:**

**sql**

**Copy code**

**SELECT product\_id, product\_name,**

**ROW\_NUMBER() OVER (ORDER BY product\_name) AS row\_num**

**FROM products;**

**WHERE: You can use the WHERE clause to filter the rows based on certain conditions, including the row number generated by ROW\_NUMBER(). For example, to retrieve only the first three rows from each partition:**

**sql**

**Copy code**

**SELECT order\_id, customer\_id, order\_date**

**FROM (**

**SELECT order\_id, customer\_id, order\_date,**

**ROW\_NUMBER() OVER (PARTITION BY customer\_id ORDER BY order\_date) AS row\_num**

**FROM orders**

**) AS ranked**

**WHERE row\_num <= 3;**

**SELECT: After using ROW\_NUMBER(), you can use the SELECT keyword to specify the columns you want to include in the final result set. You can also perform calculations or use other aggregate functions on the row numbers if needed.**

**JOIN: You can join the result set with other tables based on common columns.**

**GROUP BY: You can use the GROUP BY clause to aggregate data within partitions.**

**HAVING: You can use the HAVING clause to filter the result set based on aggregate functions.**

**DISTINCT: You can use the DISTINCT keyword to remove duplicate rows from the final result set.**

**Remember that the order of the clauses in your SQL query is essential. For example, ORDER BY must come after PARTITION BY in the OVER() clause when using ROW\_NUMBER(). Always ensure that the syntax of your SQL query is correct and test it on a sample dataset before applying it to your production data.**

**Composite primary key**

**If you're referring to having two primary keys in the same table, which is known as a composite primary key, here's an explanation using the example of the Borrowings table:**

**In the context of the Borrowings table, a composite primary key means that the combination of BorrowerID and BookID together uniquely identifies each row in the Borrowings table.**

**sql**

**Copy code**

**CREATE TABLE Borrowings (**

**BorrowerID INT,**

**BookID INT,**

**BorrowDate DATE,**

**DueDate DATE,**

**PRIMARY KEY (BorrowerID, BookID)**

**);**

**In this case:**

**BorrowerID and BookID together form the composite primary key.**

**This means that each combination of BorrowerID and BookID must be unique within the Borrowings table.**

**This uniqueness ensures that a borrower cannot borrow the same book multiple times, and each book can only have one borrowing record per borrower.**

**Using a composite primary key in the Borrowings table provides several benefits:**

**It enforces data integrity by preventing duplicate borrowings.**

**It allows efficient querying for specific borrower-book pairs or all borrowings by a particular borrower.**

**It directly aligns with the business logic of uniquely identifying a borrowing event based on the combination of borrower and book.**

**However, it's important to note that while composite primary keys have their advantages, they can also introduce complexity, especially when it comes to handling updates or maintaining the table. They should be used judiciously based on the specific requirements of your application and data model**