UNIT 4: Introduction to SQL

SQL

- SQL stands for Structured Query Language. It is used for storing and managing data in relational database management system (RDMS).
- It is a standard language for Relational Database System. It enables a user to create, read, update and delete relational databases and tables.
- All the RDBMS like MySQL, Informix, Oracle, MS Access and SQL Server use SQL as their standard database language.
- SQL allows users to query the database in a number of ways, using English-like statements.

Rules:

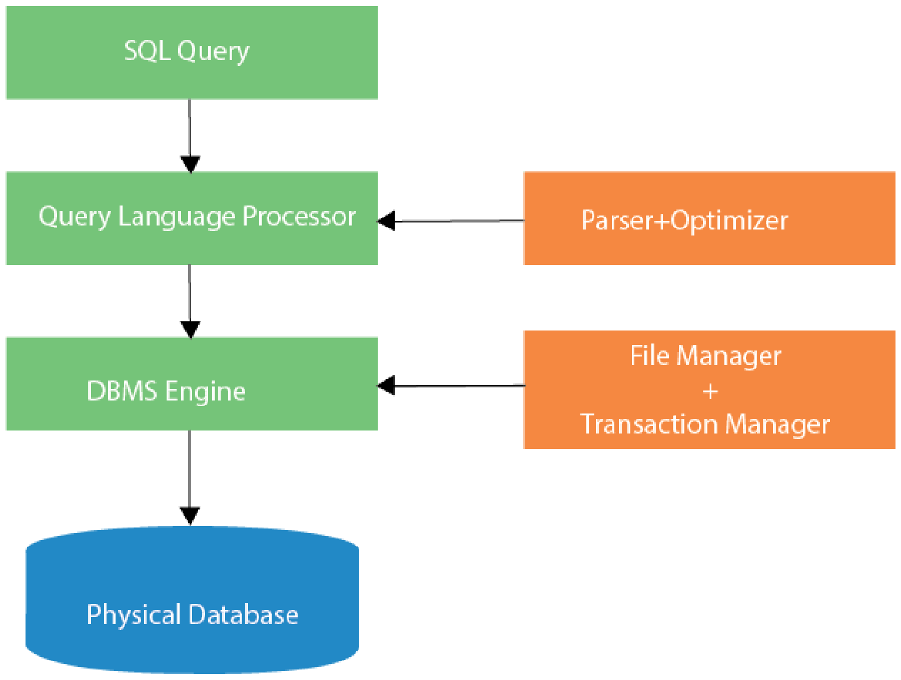
SQL follows the following rules:

- Structure query language is not case sensitive. Generally, keywords of SQL are written in uppercase.
- Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text line.
- Using the SQL statements, you can perform most of the actions in a database.
- SQL depends on tuple relational calculus and relational algebra.

SQL process:

- When an SQL command is executing for any RDBMS, then the system figure out the best way to carry out the request and the SQL engine determines that how to interpret the task.
- In the process, various components are included. These components can be optimization Engine, Query engine, Query dispatcher, classic, etc.

- All the non-SQL queries are handled by the classic query engine, but SQL query engine won't handle logical files.



History of SQL

- **1970** − Dr. Edgar F. "Ted" Codd of IBM is known as the father of relational databases. He described a relational model for databases.
- **1974** − Structured Query Language (SQL) appeared.
- **1978** − IBM worked to develop Codd's ideas and released a product named System/R.
- **1986** − IBM developed the first prototype of relational database and standardized by ANSI. The first relational database was released by Relational Software which later came to be known as Oracle.
- **1987** − SQL became the part of the International Organization for Standardization (ISO).

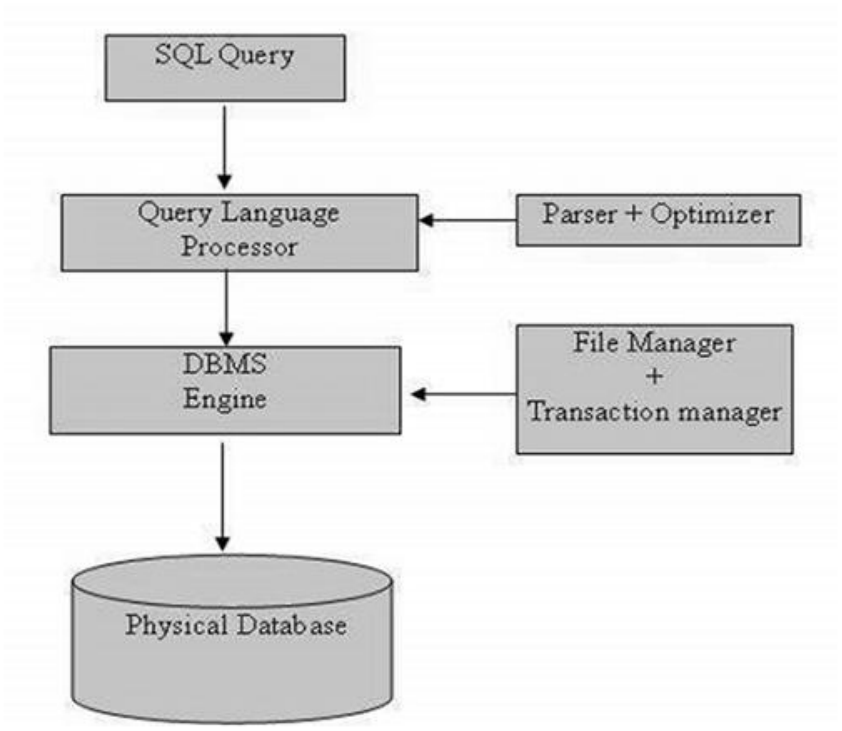
How SQL Works?

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

There are various components included in this process. These components are −

- Query Dispatcher
- Optimization Engines
- Classic Query Engine
- SQL Query Engine, etc.

A classic query engine handles all the non-SQL queries, but a SQL query engine won't handle logical files. Following is a simple diagram showing the SQL Architecture −



Characteristics of SQL

- SQL is easy to learn.
- SQL is used to access data from relational database management systems.
- SQL can execute queries against the database.
- SQL is used to describe the data.
- SQL is used to define the data in the database and manipulate it when needed.
- SQL is used to create and drop the database and table.
- SQL is used to create a view, stored procedure, function in a database.
- SQL allows users to set permissions on tables, procedures, and views.

Advantages of SQL

There are the following advantages of SQL:

High speed

Using the SQL queries, the user can quickly and efficiently retrieve a large amount of records from a database.

No coding needed

In the standard SQL, it is very easy to manage the database system. It doesn't require a substantial amount of code to manage the database system.

Well defined standards

Long established are used by the SQL databases that are being used by ISO and ANSI.

Portability

SQL can be used in laptop, PCs, server and even some mobile phones.

Interactive language

SQL is a domain language used to communicate with the database. It is also used to receive answers to the complex questions in seconds.

Multiple data view

Using the SQL language, the users can make different views of the database structure.

SQL Data Types

Data types are used to represent the nature of the data that can be stored in the database table. For example, in a particular column of a table, if we want to store a string type of data then we will have to declare a string data type of this column.

Data types mainly classified into three categories for every database.

- String Data types
- Numeric Data types
- Date and time Data types

Data Types in MySQL, SQL Server and Oracle Databases

MySQL Data Types

A list of data types used in MySQL database. This is based on MySQL 8.0.

MySQL String Data Types

CHAR(Size)	It is used to specify a fixed length string that can contain numbers, letters, and special characters. Its size can be 0 to 255 characters. Default is 1.
VARCHAR(Size)	It is used to specify a variable length string that can contain numbers, letters, and special characters. Its size can be from 0 to 65535 characters.
BINARY(Size)	It is equal to CHAR() but stores binary byte strings. Its size parameter specifies the column length in the bytes. Default is 1.
VARBINARY(Size)	It is equal to VARCHAR() but stores binary byte strings. Its size parameter specifies the maximum column length in bytes.
TEXT(Size)	It holds a string that can contain a maximum length of 255 characters.
TINYTEXT	It holds a string with a maximum length of 255 characters.
MEDIUMTEXT	It holds a string with a maximum length of 16,777,215.
LONGTEXT	It holds a string with a maximum length of 4,294,967,295 characters.
ENUM(val1, val2, val3,...)	It is used when a string object having only one value, chosen from a list of possible values. It contains 65535 values in an ENUM list. If you insert a value that is not in the list, a blank value will be inserted.
SET(val1,val2,val3,....)	It is used to specify a string that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values at one time in a SET list.

BLOB(size)	It is used for BLOBs (Binary Large Objects). It can hold up to 65,535 bytes.

MySQL Numeric Data Types

BIT(Size)	It is used for a bit-value type. The number of bits per value is specified in size. Its size can be 1 to 64. The default value is 1.
INT(size)	It is used for the integer value. Its signed range varies from -2147483648 to 2147483647 and unsigned range varies from 0 to 4294967295. The size parameter specifies the max display width that is 255.
INTEGER(size)	It is equal to INT(size).
FLOAT(size, d)	It is used to specify a floating point number. Its size parameter specifies the total number of digits. The number of digits after the decimal point is specified by d parameter.
FLOAT(p)	It is used to specify a floating point number. MySQL used p parameter to determine whether to use FLOAT or DOUBLE. If p is between 0 to24, the data type becomes FLOAT (). If p is from 25 to 53, the data type becomes DOUBLE().
DOUBLE(size, d)	It is a normal size floating point number. Its size parameter specifies the total number of digits. The number of digits after the decimal is specified by d parameter.
DECIMAL(size, d)	It is used to specify a fixed point number. Its size parameter specifies the total number of digits. The number of digits after the decimal parameter is specified by d parameter. The maximum value for the size is 65, and the default value is 10. The maximum value for d is 30, and the default value is 0.
DEC(size, d)	It is equal to DECIMAL(size, d).
BOOL	It is used to specify Boolean values true and false. Zero is considered as false, and nonzero values are considered as true.

MySQL Date and Time Data Types

DATE	It is used to specify date format YYYY-MM-DD. Its supported range is from '1000-01-01' to '9999-12-31'.
DATETIME(fsp)	It is used to specify date and time combination. Its format is YYYY-MM-DD hh:mm:ss. Its supported range is from '1000-01-01 00:00:00' to 9999-12-31 23:59:59'.
TIMESTAMP(fsp)	It is used to specify the timestamp. Its value is stored as the number of seconds since the Unix epoch('1970-01-01 00:00:00' UTC). Its format is YYYY-MM-DD hh:mm:ss. Its supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC.
TIME(fsp)	It is used to specify the time format. Its format is hh:mm:ss. Its supported range is from '-838:59:59' to '838:59:59'
YEAR	It is used to specify a year in four-digit format. Values allowed in four digit format from 1901 to 2155, and 0000.

SQL Server Data Types

SQL Server String Data Type

char(n)	It is a fixed width character string data type. Its size can be up to 8000 characters.
varchar(n)	It is a variable width character string data type. Its size can be up to 8000 characters.
varchar(max)	It is a variable width character string data types. Its size can be up to 1,073,741,824 characters.
text	It is a variable width character string data type. Its size can be up to 2GB of text data.
nchar	It is a fixed width Unicode string data type. Its size can be up to 4000 characters.

nvarchar	It is a variable width Unicode string data type. Its size can be up to 4000 characters.
ntext	It is a variable width Unicode string data type. Its size can be up to 2GB of text data.
binary(n)	It is a fixed width Binary string data type. Its size can be up to 8000 bytes.
varbinary	It is a variable width Binary string data type. Its size can be up to 8000 bytes.
image	It is also a variable width Binary string data type. Its size can be up to 2GB.

SQL Server Numeric Data Types

bit	It is an integer that can be 0, 1 or null.
tinyint	It allows whole numbers from 0 to 255.
Smallint	It allows whole numbers between -32,768 and 32,767.
Int	It allows whole numbers between -2,147,483,648 and 2,147,483,647.
bigint	It allows whole numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.
float(n)	It is used to specify floating precision number data from -1.79E+308 to 1.79E+308. The n parameter indicates whether the field should hold the 4 or 8 bytes. Default value of n is 53.
real	It is a floating precision number data from -3.40E+38 to 3.40E+38.
money	It is used to specify monetary data from -922,337,233,685,477.5808 to 922,337,203,685,477.5807.

SQL Server Date and Time Data Type

datetime	It is used to specify date and time combination. It supports range from January 1, 1753, to December 31, 9999 with an accuracy of 3.33 milliseconds.
datetime2	It is used to specify date and time combination. It supports range from January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds
date	It is used to store date only. It supports range from January 1, 0001 to December 31, 9999
time	It stores time only to an accuracy of 100 nanoseconds
timestamp	It stores a unique number when a new row gets created or modified. The time stamp value is based upon an internal clock and does not correspond to real time. Each table may contain only one-time stamp variable.

SQL Server Other Data Types

Sql_variant	It is used for various data types except for text, timestamp, and ntext. It stores up to 8000 bytes of data.
XML	It stores XML formatted data. Maximum 2GB.
cursor	It stores a reference to a cursor used for database operations.
table	It stores result set for later processing.
uniqueidentifier	It stores GUID (Globally unique identifier).

Oracle Data Types

Oracle String data types

CHAR(size)	It is used to store character data within the predefined length. It can be stored up to 2000 bytes.
NCHAR(size)	It is used to store national character data within the predefined length. It can be stored up to 2000 bytes.
VARCHAR2(size)	It is used to store variable string data within the predefined length. It can be stored up to 4000 byte.
VARCHAR(SIZE)	It is the same as VARCHAR2(size). You can also use VARCHAR(size), but it is suggested to use VARCHAR2(size)
NVARCHAR2(size)	It is used to store Unicode string data within the predefined length. We have to must specify the size of NVARCHAR2 data type. It can be stored up to 4000 bytes.

Oracle Numeric Data Types

NUMBER(p, s)	It contains precision p and scale s. The precision p can range from 1 to 38, and the scale s can range from -84 to 127.
FLOAT(p)	It is a subtype of the NUMBER data type. The precision p can range from 1 to 126.
BINARY_FLOAT	It is used for binary precision(32-bit). It requires 5 bytes, including length byte.
BINARY_DOUBLE	It is used for double binary precision (64-bit). It requires 9 bytes, including length byte.

Oracle Date and Time Data Types

DATE	It is used to store a valid date-time format with a fixed length. Its range varies from January 1, 4712 BC to December 31, 9999 AD.

TIMESTAMP	It is used to store the valid date in YYYY-MM-DD with time hh:mm:ss format.

Oracle Large Object Data Types (LOB Types)

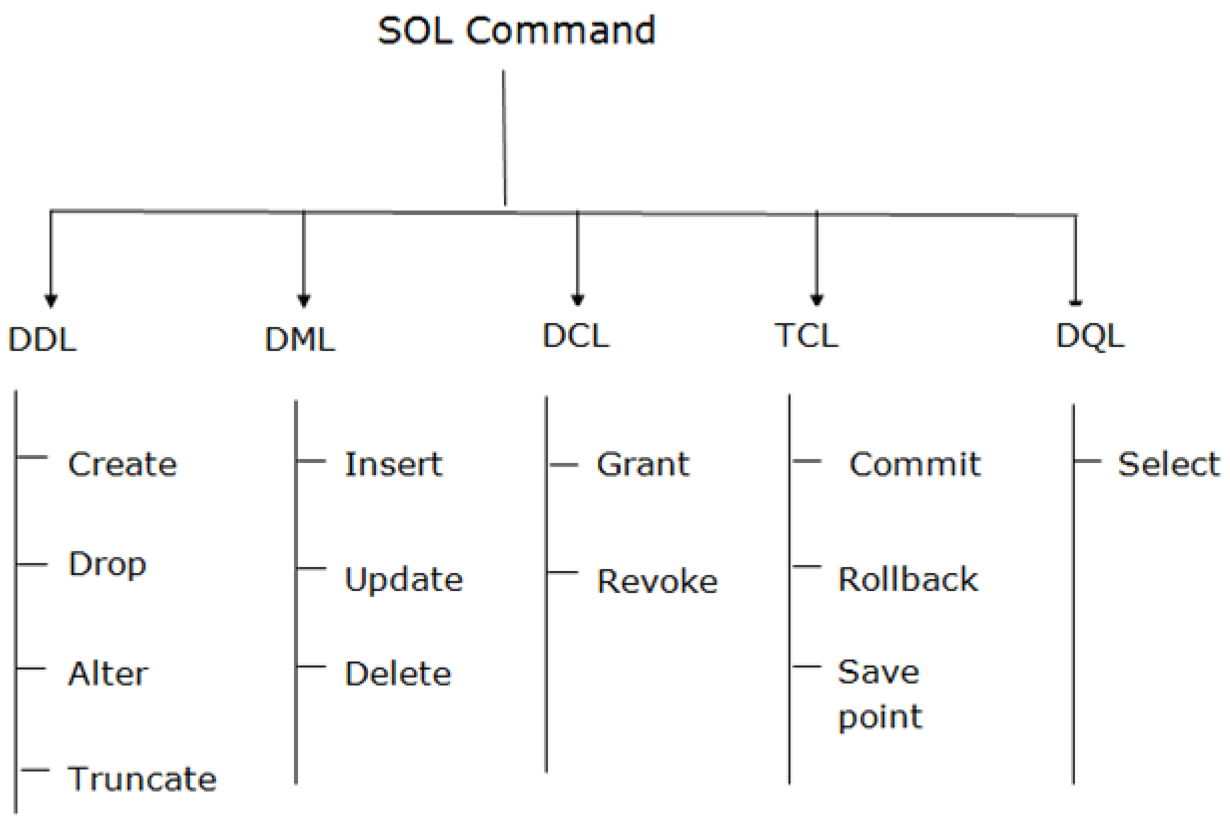
BLOB	It is used to specify unstructured binary data. Its range goes up to 2^{32}-1 bytes or 4 GB.
BFILE	It is used to store binary data in an external file. Its range goes up to 2^{32}-1 bytes or 4 GB.
CLOB	It is used for single-byte character data. Its range goes up to 2^{32}-1 bytes or 4 GB.
NCLOB	It is used to specify single byte or fixed length multibyte national character set (NCHAR) data. Its range is up to 2^{32}-1 bytes or 4 GB.
RAW(size)	It is used to specify variable length raw binary data. Its range is up to 2000 bytes per row. Its maximum size must be specified.
LONG RAW	It is used to specify variable length raw binary data. Its range up to 2^{31}-1 bytes or 2 GB, per row.

SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

Types of SQL Commands

There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.



1. Data Definition Language (DDL)

- DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.
- All the command of DDL are auto-committed that means it permanently save all the changes in the database.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

a. CREATE It is used to create a new table in the database.

Syntax:

1. CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);

   Example:

1. CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

   b. DROP: It is used to delete both the structure and record stored in the table.

   Syntax

1. DROP TABLE table_name;

   Example

1. DROP TABLE EMPLOYEE;

   c. ALTER: It is used to alter the structure of the database. This change could be either to modify the characteristics of an existing attribute or probably to add a new attribute.

   Syntax:

   To add a new column in the table

1. ALTER TABLE table_name ADD column_name COLUMN-definition;

   To modify existing column in the table:

1. ALTER TABLE table_name MODIFY(column_definitions....);

   EXAMPLE

1. ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
2. ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));

   d. TRUNCATE: It is used to delete all the rows from the table and free the space containing the table.

   Syntax:

1. TRUNCATE TABLE table_name;

   Example:

1. TRUNCATE TABLE EMPLOYEE;

2. Data Manipulation Language

- DML commands are used to modify the database. It is responsible for all form of changes in the database.

- The command of DML is not auto-committed that means it can't permanently save all the changes in the database. They can be rollback.

Here are some commands that come under DML:

- INSERT
- UPDATE
- DELETE

a. INSERT: The INSERT statement is a SQL query. It is used to insert data into the row of a table.

Syntax:

```
INSERT INTO TABLE_NAME
(col1, col2, col3,.... col N)
VALUES (value1, value2, value3, .... valueN);
```

Or

```
INSERT INTO TABLE_NAME
VALUES (value1, value2, value3, .... valueN);
```

For example:

```
INSERT INTO javatpoint (Author, Subject) VALUES ("Sonoo", "DBMS");
```

b. UPDATE: This command is used to update or modify the value of a column in the table.

Syntax:

```
UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITI
ON]
```

For example:

```
UPDATE students
SET User_Name = 'Sonoo'
WHERE Student_Id = '3'
```

c. DELETE: It is used to remove one or more row from a table.

Syntax:

```
DELETE FROM table_name [WHERE condition];
```

For example:

```
1. DELETE FROM javatpoint
2. WHERE Author="Sonoo";
```

3. Data Control Language

DCL commands are used to grant and take back authority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

a. Grant: It is used to give user access privileges to a database.

Example

```
1. GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
```

b. Revoke: It is used to take back permissions from the user.

Example

```
1. REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;
```

4. Transaction Control Language

TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

a. Commit: Commit command is used to save all the transactions to the database.

Syntax:

```
1. COMMIT;
```

Example:

```
1. DELETE FROM CUSTOMERS
```

2. WHERE AGE = 25;
3. COMMIT;

b. Rollback: Rollback command is used to undo transactions that have not already been saved to the database.

Syntax:

1. ROLLBACK;

Example:

1. DELETE FROM CUSTOMERS
2. WHERE AGE = 25;
3. ROLLBACK;

c. SAVEPOINT: It is used to roll the transaction back to a certain point without rolling back the entire transaction.

Syntax:

1. SAVEPOINT SAVEPOINT_NAME;

5. Data Query Language

DQL is used to fetch the data from the database.

It uses only one command:

- SELECT

a. SELECT: This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.

Syntax:

1. SELECT expressions
2. FROM TABLES
3. WHERE conditions;

For example:

1. SELECT emp_name
2. FROM employee
3. WHERE age > 20;

SQL Operators

Every database administrator and user uses SQL queries for manipulating and accessing the data of database tables and views.

The manipulation and retrieving of the data are performed with the help of reserved words and characters, which are used to perform arithmetic operations, logical operations, comparison operations, compound operations, etc.

What is SQL Operator?

The SQL reserved words and characters are called operators, which are used with a WHERE clause in a SQL query. In SQL, an operator can either be a unary or binary operator. The unary operator uses only one operand for performing the unary operation, whereas the binary operator uses two operands for performing the binary operation.

Syntax of Unary SQL Operator

1. Operator SQL_Operand

Syntax of Unary SQL Operator

1. Operand1 SQL_Operator Operand2

Note: SQL operators are used for filtering the table's data by a specific condition in the SQL statement.

What is the Precedence of SQL Operator?

The precedence of SQL operators is the sequence in which the SQL evaluates the different operators in the same expression. Structured Query Language evaluates those operators first, which have high precedence.

In the following table, the operators at the top have high precedence, and the operators that appear at the bottom have low precedence.

SQL Operator Symbols	Operators
**	Exponentiation operator

+, -	Identity operator, Negation operator
*, /	Multiplication operator, Division operator
+, -, ||	Addition (plus) operator, subtraction (minus) operator, String Concatenation operator
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparison Operators
NOT	Logical negation operator
&& or AND	Conjunction operator
OR	Inclusion operator

For Example,

```
UPDATE employee
SET salary = 20 - 3 * 5 WHERE Emp_Id = 5;
```

In the above SQL example, salary is assigned 5, not 85, because the * (Multiplication)

Operator has higher precedence than the - (subtraction) operator, so it first gets multiplied with 3*5 and then subtracts from 20.

Types of Operator

SQL operators are categorized in the following categories:

1. SQL Arithmetic Operators
2. SQL Comparison Operators
3. SQL Logical Operators
4. SQL Set Operators
5. SQL Bit-wise Operators
6. SQL Unary Operators

Let's discuss each operator with their types.

SQL Arithmetic Operators

The Arithmetic Operators perform the mathematical operation on the numerical data of the SQL tables. These operators perform addition, subtraction, multiplication, and division operations on the numerical operands.

Following are the various arithmetic operators performed on the SQL data:

1. SQL Addition Operator (+)
2. SQL Subtraction Operator (-)
3. SQL Multiplication Operator (+)
4. SQL Division Operator (-)
5. SQL Modulus Operator (+)

SQL Addition Operator (+)

The Addition Operator in SQL performs the addition on the numerical data of the database table. In SQL, we can easily add the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also add the numbers to the existing numbers of the specific column.

Syntax of SQL Addition Operator:

```
1. SELECT operand1 + operand2;
```

Let's understand the below example which explains how to execute Addition Operator in SQL query:

This example consists of an Employee_details table, which has four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_Monthlybonus.

Emp Id	**Emp Name**	**Emp Salary**	**Emp Monthlybonus**
101	Tushar	25000	4000

102	Anuj	30000	200

- Suppose, we want to add 20,000 to the salary of each employee specified in the table. Then, we have to write the following query in the SQL:

```
1. SELECT Emp_Salary + 20000 as Emp_New_Salary FROM Employee_details;
```

In this query, we have performed the SQL addition operation on the single column of the given table.

- Suppose, we want to add the Salary and monthly bonus columns of the above table, then we have to write the following query in SQL:

```
1. SELECT Emp_Salary + Emp_Monthlybonus as Emp_Total_Salary FROM Employee_details;
```

In this query, we have added two columns with each other of the above table.

SQL Subtraction Operator (-)

The Subtraction Operator in SQL performs the subtraction on the numerical data of the database table. In SQL, we can easily subtract the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also subtract the number from the existing number of the specific table column.

Syntax of SQL Subtraction Operator:

```
1. SELECT operand1 - operand2;
```

Let's understand the below example which explains how to execute Subtraction Operator in SQL query:

This example consists of an Employee_details table, which has four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_Monthlybonus.

Emp Id	**Emp Name**	**Emp Salary**	**Penalty**
201	Abhay	25000	200
202	Sumit	30000	500

- Suppose we want to subtract 5,000 from the salary of each employee given in the Employee_details table. Then, we have to write the following query in the SQL:

```
1. SELECT Emp_Salary - 5000 as Emp_New_Salary FROM Employee_details;
```

In this query, we have performed the SQL subtraction operation on the single column of the given table.

- If we want to subtract the penalty from the salary of each employee, then we have to write the following query in SQL:

```
1. SELECT Emp_Salary - Penalty as Emp_Total_Salary FROM Employee_details;
```

SQL Multiplication Operator (*)

The Multiplication Operator in SQL performs the Multiplication on the numerical data of the database table. In SQL, we can easily multiply the numerical values of two columns of the same table by specifying both the column names as the first and second operand.

Syntax of SQL Multiplication Operator:

```
1. SELECT operand1 * operand2;
```

Let's understand the below example which explains how to execute Multiplication Operator in SQL query:

This example consists of an Employee_details table, which has four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_Monthlybonus.

Emp Id	Emp Name	Emp Salary	Penalty
201	Abhay	25000	200
202	Sumit	30000	500

- Suppose, we want to double the salary of each employee given in the Employee_details table. Then, we have to write the following query in the SQL:

```
1. SELECT Emp_Salary * 2 as Emp_New_Salary FROM Employee_details;
```

In this query, we have performed the SQL multiplication operation on the single column of the given table.

- If we want to multiply the Emp_Id column to Emp_Salary column of that employee whose Emp_Id is 202, then we have to write the following query in SQL:

1. SELECT Emp_Id * Emp_Salary as Emp_Id * Emp_Salary FROM Employee_details WHERE Emp_Id = 202;

In this query, we have multiplied the values of two columns by using the WHERE clause.

SQL Division Operator (/)

The Division Operator in SQL divides the operand on the left side by the operand on the right side.

Syntax of SQL Division Operator:

1. SELECT operand1 / operand2;

In SQL, we can also divide the numerical values of one column by another column of the same table by specifying both column names as the first and second operand.

We can also perform the division operation on the stored numbers in the column of the SQL table.

Let's understand the below example which explains how to execute Division Operator in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, and Emp_Salary.

Emp Id	**Emp Name**	**Emp Salary**
201	Abhay	25000
202	Sumit	30000

- Suppose, we want to half the salary of each employee given in the Employee_details table. For this operation, we have to write the following query in the SQL:

1. SELECT Emp_Salary / 2 as Emp_New_Salary FROM Employee_details;

In this query, we have performed the SQL division operation on the single column of the given table.

SQL Modulus Operator (%)

The Modulus Operator in SQL provides the remainder when the operand on the left side is divided by the operand on the right side.

Syntax of SQL Modulus Operator:

```
1. SELECT operand1 % operand2;
```

Let's understand the below example which explains how to execute Modulus Operator in SQL query:

This example consists of a Division table, which has three columns Number, First_operand, and Second_operand.

Number	First operand	Second operand
1	56	4
2	32	8
3	89	9
4	18	10
5	10	5

- If we want to get the remainder by dividing the numbers of First_operand column by the numbers of Second_operand column, then we have to write the following query in SQL:

```
1. SELECT First_operand % Second_operand as Remainder FROM Employee_details;
```

SQL Comparison Operators

The Comparison Operators in SQL compare two different data of SQL table and check whether they are the same, greater, and lesser. The SQL comparison operators are used with the WHERE clause in the SQL queries

Following are the various comparison operators which are performed on the data stored in the SQL database tables:

1. SQL Equal Operator (=)
2. SQL Not Equal Operator (!=)
3. SQL Greater Than Operator (>)
4. SQL Greater Than Equals to Operator (>=)
5. SQL Less Than Operator (<)\
6. SQL Less Than Equals to Operator (<=)

SQL Equal Operator (=)

This operator is highly used in SQL queries. The Equal Operator in SQL shows only data that matches the specified value in the query.

This operator returns TRUE records from the database table if the value of both operands specified in the query is matched.

Let's understand the below example which explains how to execute Equal Operator in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, and Emp_Salary.

Emp Id	Emp Name	Emp Salary
201	Abhay	30000
202	Ankit	40000
203	Bheem	30000
204	Ram	29000
205	Sumit	30000

- Suppose, we want to access all the records of those employees from the Employee_details table whose salary is 30000. Then, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Salary = 30000;
```

In this example, we used the SQL equal operator with WHERE clause for getting the records of those employees whose salary is 30000.

SQL Equal Not Operator (!=)

The Equal Not Operator in SQL shows only those data that do not match the query's specified value.

This operator returns those records or rows from the database views and tables if the value of both operands specified in the query is not matched with each other.

Let's understand the below example which explains how to execute Equal Not Operator in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, and Emp_Salary.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the Employee_details table whose salary is not 45000. Then, we have to write the following query in the SQL database:

1. SELECT * FROM Employee_details WHERE Emp_Salary != 45000;

In this example, we used the SQL equal not operator with WHERE clause for getting the records of those employees whose salary is not 45000.

SQL Greater Than Operator (>)

The Greater Than Operator in SQL shows only those data which are greater than the value of the right-hand operand.

Let's understand the below example which explains how to execute Greater ThanOperator (>) in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, and Emp_Salary.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the Employee_details table whose employee id is greater than 202. Then, we have to write the following query in the SQL database:

1. SELECT * FROM Employee_details WHERE Emp_Id > 202;

Here, SQL greater than operator displays the records of those employees from the above table whose Employee Id is greater than 202.

SQL Greater Than Equals to Operator (>=)

The Greater Than Equals to Operator in SQL shows those data from the table which are greater than and equal to the value of the right-hand operand.

Let's understand the below example which explains how to execute greater than equals to the operator (>=) in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, and Emp_Salary.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the Employee_details table whose employee id is greater than and equals to 202. For this, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Id >= 202;
```

Here,'SQL greater than equals to operator' with WHERE clause displays the rows of those employees from the table whose Employee Id is greater than and equals to 202.

SQL Less Than Operator (<)

The Less Than Operator in SQL shows only those data from the database tables which are less than the value of the right-side operand.

This comparison operator checks that the left side operand is lesser than the right side operand. If the condition becomes true, then this operator in SQL displays the data which is less than the value of the right-side operand.

Let's understand the below example which explains how to execute less than operator (<) in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, and Emp_Salary.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the Employee_details table whose employee id is less than 204. For this, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Id < 204;
```

Here,SQL less than operator with WHERE clause displays the records of those employees from the above table whose Employee Id is less than 204.

SQL Less Than Equals to Operator (<=)

The Less Than Equals to Operator in SQL shows those data from the table which are lesser and equal to the value of the right-side operand.

This comparison operator checks that the left side operand is lesser and equal to the right side operand.

Let's understand the below example which explains how to execute less than equals to the operator (<=) in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, and Emp_Salary.

Emp Id	Emp Name	Emp Salary
201	Abhay	45000
202	Ankit	45000
203	Bheem	30000
204	Ram	29000
205	Sumit	29000

- Suppose, we want to access all the records of those employees from the Employee_details table whose employee id is less and equals 203. For this, we have to write the following query in the SQL database:

```
1. SELECT * FROM Employee_details WHERE Emp_Id <= 203;
```

Here, SQL less than equals to the operator with WHERE clause displays the rows of those employees from the table whose Employee Id is less than and equals 202.

SQL Logical Operators

The Logical Operators in SQL perform the Boolean operations, which give two results True and False. These operators provide True value if both operands match the logical condition.

Following are the various logical operators which are performed on the data stored in the SQL database tables:

1. SQL ALL operator
2. SQL AND operator
3. SQL OR operator
4. SQL BETWEEN operator
5. SQL IN operator
6. SQL NOT operator

7. SQL ANY operator
8. SQL LIKE operator

SQL ALL Operator

The ALL operator in SQL compares the specified value to all the values of a column from the sub-query in the SQL database.

This operator is always used with the following statement:

1. SELECT,
2. HAVING, and
3. WHERE.

Syntax of ALL operator:

```
1. SELECT column_Name1, ...., column_NameN FROM table_Name WHERE column Comparison_op
erator ALL (SELECT column FROM tablename2)
```

Let's understand the below example which explains how to execute ALL logical operators in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	**Emp Name**	**Emp Salary**	**Emp City**
201	Abhay	25000	Gurgaon
202	Ankit	45000	Delhi
203	Bheem	30000	Jaipur
204	Ram	29000	Mumbai
205	Sumit	40000	Kolkata

- If we want to access the employee id and employee names of those employees from the table whose salaries are greater than the salary of employees who lives in Jaipur city, then we have to type the following query in SQL.

```
1. SELECT Emp_Id, Emp_Name FROM Employee_details WHERE Emp_Salary > ALL ( SELECT Emp_Sa
   lary FROM  Employee_details WHERE Emp_City = Jaipur)
```

Here, we used the SQL ALL operator with greater than the operator.

SQL AND Operator

The AND operator in SQL would show the record from the database table if all the conditions separated by the AND operator evaluated to True. It is also known as the conjunctive operator and is used with the WHERE clause.

Syntax of AND operator:

```
1. SELECT column1, ...., columnN FROM table_Name WHERE condition1 AND condition2 AND cond
   ition3 AND ....... AND conditionN;
```

Let's understand the below example which explains how to execute AND logical operator in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	**Emp Name**	**Emp Salary**	**Emp City**
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to access all the records of those employees from the Employee_details table whose salary is 25000 and the city is Delhi. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Salary = 25000 OR Emp_City = 'Delhi';

   Here,SQL AND operator with WHERE clause shows the record of employees whose salary is 25000 and the city is Delhi.

SQL OR Operator

The OR operator in SQL shows the record from the table if any of the conditions separated by the OR operator evaluates to True. It is also known as the conjunctive operator and is used with the WHERE clause.

Syntax of OR operator:

1. SELECT column1, ..., columnN FROM table_Name WHERE condition1 OR condition2 OR conditio n3 OR .... OR conditionN;

Let's understand the below example which explains how to execute OR logical operator in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- If we want to access all the records of those employees from the Employee_details table whose salary is 25000 or the city is Delhi. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Salary = 25000 OR Emp_City = 'Delhi';
```

Here, SQL OR operator with WHERE clause shows the record of employees whose salary is 25000 or the city is Delhi.

SQL BETWEEN Operator

The BETWEEN operator in SQL shows the record within the range mentioned in the SQL query. This operator operates on the numbers, characters, and date/time operands.

If there is no value in the given range, then this operator shows NULL value.

Syntax of BETWEEN operator:

```
1. SELECT column_Name1, column_Name2 ...., column_NameN FROM table_Name WHERE column_
   nameBETWEEN value1 and value2;
```

Let's understand the below example which explains how to execute BETWEEN logical operator in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to access all the information of those employees from the Employee_details table who is having salaries between 20000 and 40000. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Salary BETWEEN 30000 AND 45000;

Here, we used the SQL BETWEEN operator with the Emp_Salary field.

SQL IN Operator

The IN operator in SQL allows database users to specify two or more values in a WHERE clause. This logical operator minimizes the requirement of multiple OR conditions.

This operator makes the query easier to learn and understand. This operator returns those rows whose values match with any value of the given list.

Syntax of IN operator:

1. SELECT column_Name1, column_Name2 ...., column_NameN FROM table_Name WHERE column_name IN (list_of_values);

Let's understand the below example which explains how to execute IN logical operator in SQL query:

This example consists of an Employee_details table, which has three columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	**Emp Name**	**Emp Salary**	**Emp City**
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi
204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to show all the information of those employees from the Employee_details table whose Employee Id is 202, 204, and 205. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Id IN (202, 204, 205);
```

Here, we used the SQL IN operator with the Emp_Id column.

- Suppose, we want to show all the information of those employees from the Employee_details table whose Employee Id is not equal to 202 and 205. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE Emp_Id NOT IN (202,205);
2.
```

Here, we used the SQL NOT IN operator with the Emp_Id column.

SQL NOT Operator

The NOT operator in SQL shows the record from the table if the condition evaluates to false. It is always used with the WHERE clause.

Syntax of NOT operator:

```
1. SELECT column1, column2 ...., columnN FROM table_Name WHERE NOT condition;
```

Let's understand the below example which explains how to execute NOT logical operator in SQL query:

This example consists of an Employee_details table, which has four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	Emp Name	Emp Salary	Emp City
201	Abhay	25000	Delhi
202	Ankit	45000	Chandigarh
203	Bheem	30000	Delhi

204	Ram	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to show all the information of those employees from the Employee_details table whose Cityis not Delhi. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE NOT Emp_City = 'Delhi' ;
```

In this example, we used the SQL NOT operator with the Emp_City column.

- Suppose, we want to show all the information of those employees from the Employee_details table whose Cityis not Delhi and Chandigarh. For this, we have to write the following query in SQL:

```
1. SELECT * FROM Employee_details WHERE NOT Emp_City = 'Delhi' AND NOT Emp_City = 'Chandi
   garh';
```

In this example, we used the SQL NOT operator with the Emp_City column.

SQL ANY Operator

The ANY operator in SQL shows the records when any of the values returned by the sub-query meet the condition.

The ANY logical operator must match at least one record in the inner query and must be preceded by any SQL comparison operator.

Syntax of ANY operator:

```
1. SELECT column1, column2 ...., columnN FROM table_Name WHERE column_name comparison_o
   perator ANY ( SELECT column_name FROM table_name WHERE condition(s)) ;
```

SQL LIKE Operator

The LIKE operator in SQL shows those records from the table which match with the given pattern specified in the sub-query.

The percentage (%) sign is a wildcard which is used in conjunction with this logical operator.

This operator is used in the WHERE clause with the following three statements:

1. SELECT statement

2. UPDATE statement
3. DELETE statement

Syntax of LIKE operator:

1. SELECT column_Name1, column_Name2 ...., column_NameN FROM table_Name WHERE column_name LIKE pattern;

Let's understand the below example which explains how to execute LIKE logical operator in SQL query:

This example consists of an Employee_details table, which has four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	**Emp Name**	**Emp Salary**	**Emp City**
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- If we want to show all the information of those employees from the Employee_details whose name starts with ''s''. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Name LIKE 's%' ;

In this example, we used the SQL LIKE operator with Emp_Name column because we want to access the record of those employees whose name starts with s.

- If we want to show all the information of those employees from the Employee_detailswhose name ends with ''y''. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Name LIKE '%y' ;

- If we want to show all the information of those employees from the Employee_detailswhose name starts with ''S'' and ends with ''y''. For this, we have to write the following query in SQL:

1. SELECT * FROM Employee_details WHERE Emp_Name LIKE 'S%y' ;

SQL Set Operators

The Set Operators in SQL combine a similar type of data from two or more SQL database tables. It mixes the result, which is extracted from two or more SQL queries, into a single result.

Set operators combine more than one select statement in a single query and return a specific result set.

Following are the various set operators which are performed on the similar data stored in the two SQL database tables:

1. SQL Union Operator
2. SQL Union ALL Operator
3. SQL Intersect Operator
4. SQL Minus Operator

SQL Union Operator

The SQL Union Operator combines the result of two or more SELECT statements and provides the single output.

The data type and the number of columns must be the same for each SELECT statement used with the UNION operator. This operator does not show the duplicate records in the output table.

Syntax of UNION Set operator:

1. SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]
2. UNION
3. SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];

Let's understand the below example which explains how to execute Union operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

- Suppose, we want to see the employee name and employee id of each employee from both tables in a single output. For this, we have to write the following query in SQL:

1. SELECT Emp_ID, Emp_Name FROM Employee_details1
2. UNION

3. SELECT Emp_ID, Emp_Name FROM Employee_details2 ;

SQL Union ALL Operator

The SQL Union Operator is the same as the UNION operator, but the only difference is that it also shows the same record.

Syntax of UNION ALL Set operator:

1. SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]
2. UNION ALL
3. SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];

Let's understand the below example which explains how to execute Union ALL operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala

201	Sanjay	25000	Delhi

Table: Employee_details2

- If we want to see the employee name of each employee of both tables in a single output. For this, we have to write the following query in SQL:

```
1. SELECT Emp_Name FROM Employee_details1
2. UNION ALL
3. SELECT Emp_Name FROM Employee_details2 ;
```

SQL Intersect Operator

The SQL Intersect Operator shows the common record from two or more SELECT statements. The data type and the number of columns must be the same for each SELECT statement used with the INTERSECT operator.

Syntax of INTERSECT Set operator:

```
1. SELECT column1, column2 ...., columnN FROM table_Name1 [WHERE conditions]
2. INTERSECT
3. SELECT column1, column2 ...., columnN FROM table_Name2 [WHERE conditions];
```

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	**Emp Name**	**Emp Salary**	**Emp City**
201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

Suppose, we want to see a common record of the employee from both the tables in a single output. For this, we have to write the following query in SQL:

1. SELECT Emp_Name FROM Employee_details1
2. INTERSECT
3. SELECT Emp_Name FROM Employee_details2 ;

SQL Minus Operator

The SQL Minus Operator combines the result of two or more SELECT statements and shows only the results from the first data set.

Syntax of MINUS operator:

1. SELECT column1, column2 ...., columnN FROM First_tablename [WHERE conditions]
2. MINUS
3. SELECT column1, column2 ...., columnN FROM Second_tablename [WHERE conditions];

Let's understand the below example which explains how to execute INTERSECT operator in Structured Query Language:

In this example, we used two tables. Both tables have four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	Emp Name	Emp Salary	Emp City

201	Sanjay	25000	Delhi
202	Ajay	45000	Delhi
203	Saket	30000	Aligarh

Table: Employee_details1

Emp Id	Emp Name	Emp Salary	Emp City
203	Saket	30000	Aligarh
204	Saurabh	40000	Delhi
205	Ram	30000	Kerala
201	Sanjay	25000	Delhi

Table: Employee_details2

Suppose, we want to see the name of employees from the first result set after the combination of both tables. For this, we have to write the following query in SQL:

1. SELECT Emp_Name FROM Employee_details1
2. MINUS
3. SELECT Emp_Name FROM Employee_details2 ;

SQL Unary Operators

The Unary Operators in SQL perform the unary operations on the single data of the SQL table, i.e., these operators operate only on one operand.

These types of operators can be easily operated on the numeric data value of the SQL table.

Following are the various unary operators which are performed on the numeric data stored in the SQL table:

1. SQL Unary Positive Operator
2. SQL Unary Negative Operator
3. SQL Unary Bitwise NOT Operator

SQL Unary Positive Operator

The SQL Positive (+) operator makes the numeric value of the SQL table positive.

Syntax of Unary Positive Operator

1. SELECT +(column1), +(column2) ...., +(columnN) FROM table_Name [WHERE conditions] ;

Let's understand the below example which explains how to execute a Positive unary operator on the data of SQL table:

This example consists of anEmployee_details table, which has four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	Emp Name	Emp Salary	Emp City
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to see the salary of each employee as positive from the Employee_details table. For this, we have to write the following query in SQL:

1. SELECT +Emp_Salary Employee_details ;

SQL Unary Negative Operator

The SQL Negative (-) operator makes the numeric value of the SQL table negative.

Syntax of Unary Negative Operator

1. SELECT -(column_Name1), -(column_Name2) ...., -(column_NameN) FROM table_Name [WHERE conditions] ;

Let's understand the below example which explains how to execute Negative unary operator on the data of SQL table:

This example consists of an Employee_details table, which has four columns Emp_Id, Emp_Name, Emp_Salary, and Emp_City.

Emp Id	**Emp Name**	**Emp Salary**	**Emp City**
201	Sanjay	25000	Delhi
202	Ajay	45000	Chandigarh
203	Saket	30000	Delhi
204	Abhay	25000	Delhi
205	Sumit	40000	Kolkata

- Suppose, we want to see the salary of each employee as negative from the Employee_details table. For this, we have to write the following query in SQL:

1. SELECT -Emp_Salary Employee_details ;

- Suppose, we want to see the salary of those employees as negative whose city is Kolkatain the Employee_details table. For this, we have to write the following query in SQL:

1. SELECT -Emp_Salary Employee_details WHERE Emp_City = 'Kolkata';

SQL Bitwise NOT Operator

The SQL Bitwise NOT operator provides the one's complement of the single numeric operand. This operator turns each bit of numeric value. If the bit of any numerical value is 001100, then this operator turns these bits into 110011.

Syntax of Bitwise NOT Operator

```
1. SELECT ~(column1), ~(column2) ...., ~(columnN) FROM table_Name [WHERE conditions] ;
```

Let's understand the below example which explains how to execute the Bitwise NOT operator on the data of SQL table:

This example consists of aStudent_details table, which has four columns Roll_No, Stu_Name, Stu_Marks, and Stu_City.

Emp Id	Stu Name	Stu Marks	Stu City
101	Sanjay	85	Delhi
102	Ajay	97	Chandigarh
103	Saket	45	Delhi
104	Abhay	68	Delhi
105	Sumit	60	Kolkata

If we want to perform the Bitwise Not operator on the marks column of Student_details, we have to write the following query in SQL:

```
1. SELECT ~Stu_Marks Employee_details ;
```

SQL Bitwise Operators

The Bitwise Operators in SQL perform the bit operations on the Integer values. To understand the performance of Bitwise operators, you just knew the basics of Boolean algebra.

Following are the two important logical operators which are performed on the data stored in the SQL database tables:

1. Bitwise AND (&)
2. Bitwise OR(|)

Bitwise AND (&)

The Bitwise AND operator performs the logical AND operation on the given Integer values. This operator checks each bit of a value with the corresponding bit of another value.

Syntax of Bitwise AND Operator

1. SELECT column1 & column2 & .... & columnN FROM table_Name [WHERE conditions] ;

Let's understand the below example which explains how to execute Bitwise AND operator on the data of SQL table:

This example consists of the following table, which has two columns. Each column holds numerical values.

When we use the Bitwise AND operator in SQL, then SQL converts the values of both columns in binary format, and the AND operation is performed on the converted bits.

After that, SQL converts the resultant bits into user understandable format, i.e., decimal format.

Column1	**Column2**
1	1
2	5
3	4
4	2
5	3

- Suppose, we want to perform the Bitwise AND operator between both the columns of the above table. For this, we have to write the following query in SQL:

1. SELECT Column1 & Column2 From TABLE_AND ;

Bitwise OR (|)

The Bitwise OR operator performs the logical OR operation on the given Integer values. This operator checks each bit of a value with the corresponding bit of another value.

Syntax of Bitwise OR Operator

```
1. SELECT column1 | column2 | .... | columnN FROM table_Name [WHERE conditions] ;
```

Let's understand the below example which explains how to execute Bitwise OR operator on the data of SQL table:

This example consists of a table that has two columns. Each column holds numerical values.

When we used the Bitwise OR operator in SQL, then SQL converts the values of both columns in binary format, and the OR operation is performed on the binary bits. After that, SQL converts the resultant binary bits into user understandable format, i.e., decimal format.

Column1	**Column2**
1	1
2	5
3	4
4	2
5	3

- Suppose, we want to perform the Bitwise OR operator between both the columns of the above table. For this, we have to write the following query in SQL:

```
1. SELECT Column1 | Column2 From TABLE_OR ;
```

Operators Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 * 2**; here, **x** is assigned **13**, not 20 because operator * has higher precedence than +, so it first gets multiplied with **3*2** and then adds into **7**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	conjunction
OR	inclusion

Example

Try the following example to understand the operator precedence available in PL/SQL −

```
DECLARE
   a number(2) := 20;
   b number(2) := 10;
```

```
  c number(2) := 15;
  d number(2) := 5;
  e number(2) ;
BEGIN
  e := (a + b) * c / d;      -- ( 30 * 15 ) / 5
  dbms_output.put_line('Value of (a + b) * c / d is : '|| e );
  e := ((a + b) * c) / d;   -- (30 * 15 ) / 5
  dbms_output.put_line('Value of ((a + b) * c) / d is  : ' ||  e );
  e := (a + b) * (c / d);   -- (30) * (15/5)
  dbms_output.put_line('Value of (a + b) * (c / d) is  : '||  e );
  e := a + (b * c) / d;     --  20 + (150/5)
  dbms_output.put_line('Value of a + (b * c) / d is  : ' ||  e );
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is  : 90
Value of (a + b) * (c / d) is  : 90
Value of a + (b * c) / d is  : 50

PL/SQL procedure successfully completed.
```

SQL Table

Table is a collection of data, organized in terms of rows and columns. In DBMS term, table is known as relation and row as tuple.

Note: A table has a specified number of columns, but can have any number of rows.

Table is the simple form of data storage. A table is also considered as a convenient representation of relations.

Let's see an example of an employee table:

Employee		
EMP_NAME	**ADDRESS**	**SALARY**
Ankit	Lucknow	15000

Raman	Allahabad	18000
Mike	New York	20000

In the above table, "Employee" is the table name, "EMP_NAME", "ADDRESS" and "SALARY" are the column names. The combination of data of multiple columns forms a row e.g. "Ankit", "Lucknow" and 15000 are the data of one row.

SQL TABLE Variable

The **SQL Table variable** is used to create, modify, rename, copy and delete tables. Table variable was introduced by Microsoft.

It was introduced with SQL server 2000 to be an alternative of temporary tables.

It is a variable where we temporary store records and results. This is same like temp table but in the case of temp table we need to explicitly drop it.

Table variables are used to store a set of records. So declaration syntax generally looks like CREATE TABLE syntax.

```
create table "tablename"
("column1" "data type",
"column2" "data type",
...
"columnN" "data type");
```

When a transaction rolled back the data associated with table variable is not rolled back.

A table variable generally uses lesser resources than a temporary variable.

Table variable cannot be used as an input or an output parameter.

Topics of SQL TABLE Statement

SQL TABLE Variable

What TABLE variable can do?

SQL CREATE TABLE

How to create a table using SQL query>

SQL DROP TABLE

How to drop a table?

SQL DELETE TABLE

How to delete all the records of a table?

SQL RENAME TABLE

How to rename a table?

SQL TRUNCATE TABLE

How to truncate a table?

SQL COPY TABLE

How to copy a table?

SQL TEMP TABLE

What is temporary table? What are the advantage of temporary table?

SQL ALTER TABLE

How to add, modify, rename and drop column.

views and indexes

A **view** is simply any SELECT query that has been given a name and saved in the database. For this reason, a view is sometimes called a **named query** or a **stored query**. To create a view, you use the SQL syntax:

```
CREATE OR REPLACE VIEW <view_name> AS
SELECT <any valid select query>;
```

 The view query itself is saved in the database, but it is not actually run until it is called with another SELECT statement. For this reason, the view does not take up any disk space for data storage, and it does not create any redundant copies of data that is already stored in the tables that it references (which are sometimes called the **base tables** of the view).

 Although it is not required, many database developers identify views with names such as v_Customers or Customers_view. This not only avoids name conflicts with base tables, it helps in reading any query that uses a view.

 The keywords OR REPLACE in the syntax shown above are optional. Although you don't need to use them the first time that you create a view, including them will overwrite an older version of the view with your latest one, without giving you an error message.

 The syntax to remove a view from your schema is exactly what you would expect:

```
DROP VIEW <view_name>;
```

Using views

A view name may be used in exactly the same way as a table name in any SELECT query. Once stored, the view can be used again and again, rather than re-writing the same query many times.

 The most basic use of a view would be to simply SELECT * from it, but it also might represent a pre-written subquery or a simplified way to write part of a FROM clause.

 In many systems, views are stored in a pre-compiled form. This might save some execution time for the query, but usually not enough for a human user to notice.

 One of the most important uses of views is in large multi-user systems, where they make it easy to control access to data for different types of users. As a very simple example, suppose that you have a table of employee information on the scheme Employees = {employeeID, empFName, empLName, empPhone, jobTitle, payRate, managerID}. Obviously, you can't let everyone in the company look at all of this information, let alone make changes to it.

 Your database administrator (DBA) can define **roles** to represent different groups of users, and then grant membership in one or more roles to any specific user account (schema). In turn, you can grant table-level or view-level permissions to a role as well as to a specific user. Suppose that the DBA has created the roles *managers* and *payroll* for people who occupy those positions. In Oracle®, there is also a pre-defined role named *public*, which means every user of the database.

 You could create separate views even on just the Employees table, and control access to them like this:

```
CREATE VIEW phone_view AS
SELECT empFName, empLName, empPhone FROM Employees;
GRANT SELECT ON phone_view TO public;

CREATE VIEW job_view AS
SELECT employeeID, empFName, empLName, jobTitle, managerID FROM Employees;
GRANT SELECT, UPDATE ON job_view TO managers;

CREATE VIEW pay_view AS
SELECT employeeID, empFName, empLName, payRate FROM Employees;
GRANT SELECT, UPDATE ON pay_view TO payroll;
```

 Only a very few trusted people would have SELECT, UPDATE, INSERT, and DELETE privileges on the entire Employees base table; everyone else would now have exactly the access that they need, but no more.

 When a view is the target of an UPDATE statement, the base table value is changed. You can't change a computed value in a view, or any value in a view that is based on a UNION query. You may also use a view as the target of an INSERT or DELETE statement, subject to any integrity constraints that have been placed on the base tables.

Materialized views

Sometimes, the execution speed of a query is so important that a developer is willing to trade increased disk space use for faster response, by creating a **materialized view**. Unlike the view discussed above, a materialized view *does* create and store the result table in advance, filled with data. The scheme of this table is given by the SELECT clause of the view definition.

 This technique is most useful when the query involves many joins of large tables, or any other SQL feature that could contribute to long execution times. You might encounter this in a Web project, where the site visitor simply can't be kept waiting while the query runs.

 Since the view would be useless if it is out of date, it must be re-run, at the minimum, when there is a change to any of the tables that it is based on. The SQL syntax to create a materialized view includes many options for when it is first to be run, how often it is to be re-run, and so on.

This requires an advanced reference manual for your specific system, and is beyond the scope of this tutorial.

Indexes

An **index**, as you would expect, is a data structure that the database uses to find records within a table more quickly. Indexes are built on one or more columns of a table; each index maintains a list of values within that field that are sorted in ascending or descending order. Rather than sorting records on the field or fields during query execution, the system can simply access the rows in order of the index.

Unique and non-unique indexes: When you create an index, you may allow the indexed columns to contain duplicate values; the index will still list all of the rows with duplicates. You may also specify that values in the indexed columns must be unique, just as they must be with a primary key. In fact, when you create a primary key constraint on a table, Oracle and most other systems will automatically create a unique index on the primary key columns, as well as not allowing null values in those columns. One good reason for you to create a unique index on non-primary key fields is to enforce the integrity of a candidate key, which otherwise might end up having (nonsense) duplicate values in different rows.

Queries versus insertion/update: It might seem as if you should create an index on every column or group of columns that will ever by used in an ORDER BY clause (for example: lastName, firstName). However, each index will have to be updated every time that a row is inserted or a value in that column is updated. Although index structures such as B or B+ trees allow this to happen very quickly, there still might be circumstances where too many indexes would detract from overall system performance. This and similar issues are often covered in more advanced courses.

Syntax: As you would expect by now, the SQL to create an index is:

```
    CREATE INDEX <indexname> ON <tablename> (<column>, <column>...);
```

To enforce unique values, add the UNIQUE keyword:

```
    CREATE UNIQUE INDEX <indexname> ON <tablename> (<column>, <column>...);
```

To specify sort order, add the keyword ASC or DESC after each column name, just as you would do in an ORDER BY clause.

To remove an index, simply enter:

```
DROP INDEX <indexname>;
```

SQL Sub Query

A Subquery is a query within another SQL query and embedded within the WHERE clause.

Important Rule:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

1. Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

Syntax

```
SELECT column_name
FROM table_name
WHERE column_name expression operator
( SELECT column_name  from table_name WHERE ... );
```

Example

Consider the EMPLOYEE table have the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00
7	Jackson	25	Mizoram	10000.00

The subquery with a SELECT statement will be:

```
SELECT *
   FROM EMPLOYEE
   WHERE ID IN (SELECT ID
   FROM EMPLOYEE
   WHERE SALARY > 4500);
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00

7	Jackson	25	Mizoram	10000.00

2. Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax:

```
INSERT INTO table_name (column1, column2, column3....)
SELECT *
FROM table_name
WHERE VALUE OPERATOR
```

Example

Consider a table EMPLOYEE_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE_BKP table.

```
INSERT INTO EMPLOYEE_BKP
   SELECT * FROM EMPLOYEE
   WHERE ID IN (SELECT ID
   FROM EMPLOYEE);
```

3. Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax

```
UPDATE table
SET column_name = new_value
```

```
3. WHERE VALUE OPERATOR
4.    (SELECT COLUMN_NAME
5.    FROM TABLE_NAME
6.    WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

```
1. UPDATE EMPLOYEE
2.    SET SALARY = SALARY * 0.25
3.    WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
4.       WHERE AGE >= 29);
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	1625.00
5	Kathrin	34	Bangalore	2125.00
6	Harry	42	China	1125.00
7	Jackson	25	Mizoram	10000.00

4. Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax

```
DELETE FROM TABLE_NAME
WHERE VALUE OPERATOR
  (SELECT COLUMN_NAME
  FROM TABLE_NAME
  WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

```
DELETE FROM EMPLOYEE
  WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP
    WHERE AGE >= 29 );
```

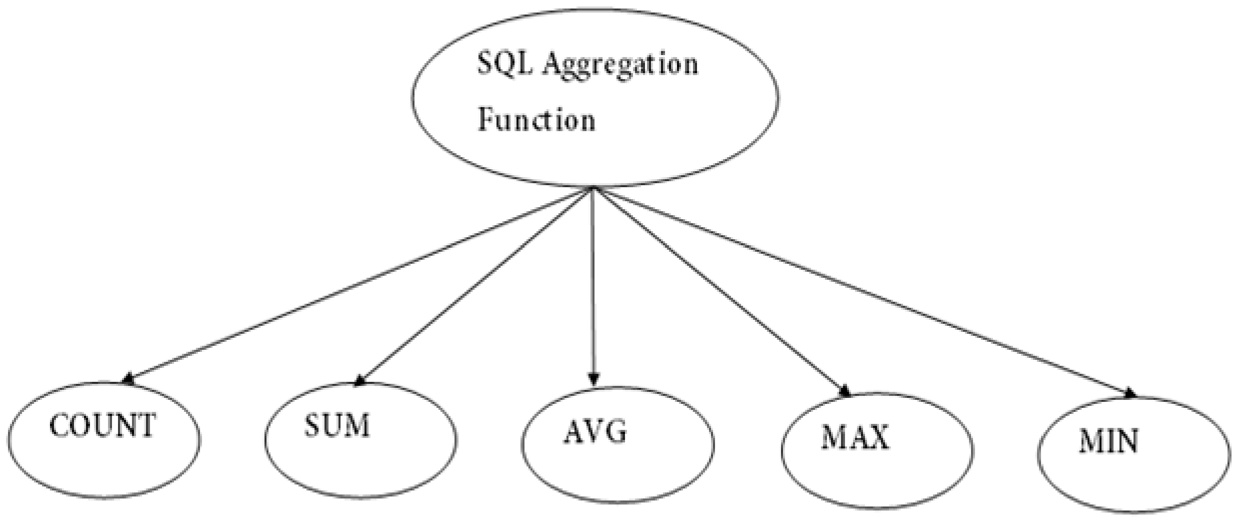
This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
7	Jackson	25	Mizoram	10000.00

SQL Aggregate Functions

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

Types of SQL Aggregation Function



1. COUNT FUNCTION

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

1. COUNT(*)
2. or
3. COUNT([ALL|DISTINCT] expression)

Sample table:

PRODUCT_MAST

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

1. SELECT COUNT(*)
2. FROM PRODUCT_MAST;

Output:

```
10
```

Example: COUNT with WHERE

```
SELECT COUNT(*)
FROM PRODUCT_MAST;
WHERE RATE>=20;
```

Output:

```
7
```

Example: COUNT() with DISTINCT

```
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_MAST;
```

Output:

```
3
```

Example: COUNT() with GROUP BY

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY;
```

Output:

```
Com1    5
Com2    3
Com3    2
```

Example: COUNT() with HAVING

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY
HAVING COUNT(*)>2;
```

Output:

```
Com1    5
Com2    3
```

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

1. SUM()
2. or
3. SUM([ALL|DISTINCT] expression)

Example: SUM()

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST;

Output:

```
670
```

Example: SUM() with WHERE

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>3;

Output:

```
320
```

Example: SUM() with GROUP BY

1. SELECT SUM(COST)
2. FROM PRODUCT_MAST
3. WHERE QTY>3
4. GROUP BY COMPANY;

Output:

```
Com1    150
Com2    170
```

Example: SUM() with HAVING

1. SELECT COMPANY, SUM(COST)
2. FROM PRODUCT_MAST
3. GROUP BY COMPANY
4. HAVING SUM(COST)>=170;

Output:

```
Com1    335
Com3    170
```

3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

1. AVG()
2. or
3. AVG([ALL|DISTINCT] expression)

Example:

1. SELECT AVG(COST)
2. FROM PRODUCT_MAST;

Output:

```
67.00
```

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

1. MAX()
2. or
3. MAX([ALL|DISTINCT] expression)

Example:

1. SELECT MAX(RATE)
2. FROM PRODUCT_MAST;

```
30
```

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

1. MIN()
2. or
3. MIN([ALL|DISTINCT] expression)

Example:

1. SELECT MIN(RATE)
2. FROM PRODUCT_MAST;

Output:

```
10
```

SQL | Join (Inner, Left, Right and Full Joins)

SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are as follows:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- NATURAL JOIN

Consider the two tables below as follows:

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

The simplest Join is INNER JOIN.

A. INNER JOIN

The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

INNER JOIN table2

ON table1.matching_column = table2.matching_column;
```

table1: First table.
table2: Second table
matching_column: Column common to both the tables.
Note*: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.*

Example Queries(INNER JOIN)
This query will show the names and age of students enrolled in different courses.

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student

INNER JOIN StudentCourse

ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

B. LEFT JOIN

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.
Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....

FROM table1
```

LEFT JOIN table2

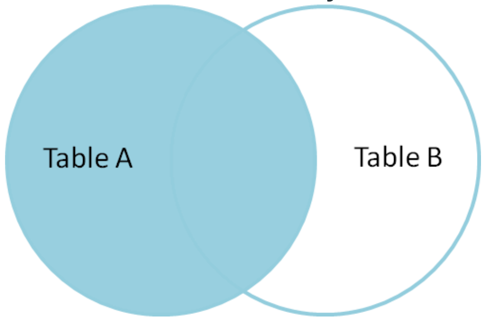
ON table1.matching_column = table2.matching_column;

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

***Note**: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*



Example Queries(LEFT JOIN):

SELECT Student.NAME,StudentCourse.COURSE_ID

FROM Student

LEFT JOIN StudentCourse

ON StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

C. RIGHT JOIN

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

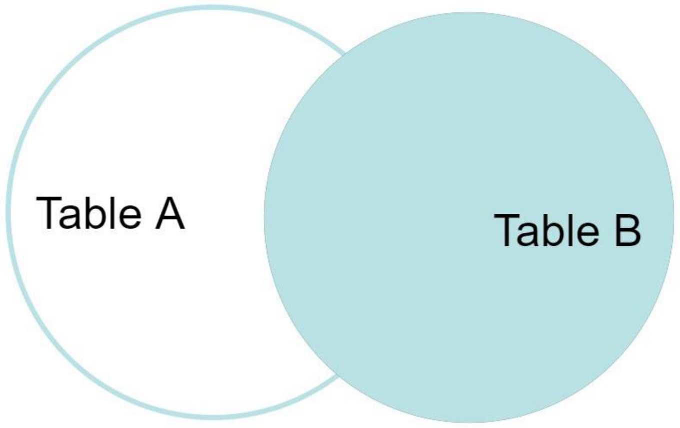
```
SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

RIGHT JOIN table2

ON table1.matching_column = table2.matching_column;
```

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

Note*: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*



Example Queries(RIGHT JOIN):

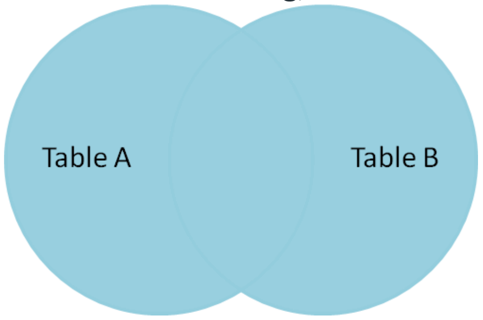
```
SELECT Student.NAME,StudentCourse.COURSE_ID

FROM Student

RIGHT JOIN StudentCourse

ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

D. FULL JOIN

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.



Syntax:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

FULL JOIN table2

ON table1.matching_column = table2.matching_column;

table1: First table.

table2: Second table

matching_column: Column common to both the tables.

Example Queries(FULL JOIN):

SELECT Student.NAME,StudentCourse.COURSE_ID

FROM Student

FULL JOIN StudentCourse

ON StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	**COURSE_ID**
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	4
NULL	5

NAME	COURSE_ID
NULL	4

[Left JOIN (Video)]
[Right JOIN (Video)]
[Full JOIN (Video)]
[SQL | JOIN (Cartesian Join, Self Join)]

E. Natural join (⋈)

Natural join can join tables based on the common columns in the tables being joined. A natural join returns all rows by matching values in common columns having same name and data type of columns and that column should be present in both tables.

Both table must have at list one common column with same column name and same data type.

The two table are joined using Cross join.

DBMS will look for a common column with same name and data type Tuples having exactly same values in common columns are kept in result.

Example:

Employee		
Emp_id	**Emp_name**	**Dept_id**
1	**Ram**	**10**
2	**Jon**	**30**
3	**Bob**	**50**

Department	
Dept_id	**Dept_name**
10	**IT**
30	**HR**
40	**TIS**

Query: Find all Employees and their respective departments.

Solution: (Employee) ⋈ (Department)

Emp_id	Emp_name	Dept_id	Dept_id	Dept_name
1	**Ram**	**10**	**10**	**IT**
2	**Jon**	**30**	**30**	**HR**
Employee data			**Department data**	

SQL UNION

UNION is an SQL operator which combines the result of two or more SELECT queries and provides the single set in the output.

Syntax of UNION in SQL:

```sql
SELECT Column_Name_1, Column_Name_2 ...., Column_NameN FROM Table_Name_1
UNION
SELECT Column_Name1, Column_Name_2 ...., Column_Name_N FROM Table_Name_2
UNION ....... UNION
SELECT Column_Name1, Column_Name_2 ...., Column_Name_N FROM Table_Name_N ;
```

The data type and the number of fields must be same for every SELECT statement connected with the UNION operator. The database system uses the UNION operator for removing the duplicate values from the combined result set.

Example of UNION operator in SQL

Let's create two different tables and insert records in both tables.

The following query creates the **Old_Employee** table with four fields:

```
CREATE TABLE Old_Employee
(
Employee_Id INT NOT NULL,
Employee_Name Varchar (40),
Emp_Age INT,
Emp_Salary INT
);
```

The following query creates the **New_Employee** table with four fields:

```
CREATE TABLE New_Employee
(
Employee_Id INT NOT NULL,
Employee_Name Varchar (40),
Emp_Age INT,
Emp_Salary INT
);
```

The following INSERT query inserts the record of old employees into the Old_Employee table:

```
INSERT INTO Old_Employee (Employee_Id, Employee_Name, Emp_Age, Emp_Salary) VALUES (1
01, Akhil, 28, 25000),
(102, Abhay, 27, 26000),
(103, Sorya, 26, 29000),
(104, Abhishek, 27, 26000),
(105, Ritik, 26, 29000),
6. (106, Yash, 29, 28000);
```

The following query shows the details of the **Old_Employee** table:

```
1. SELECT * FROM Old_Employee;
```

Employee_Id	Employee_Name	Emp_Age	Emp_Salary
101	Akhil	28	25000
102	Abhay	27	26000
103	Sorya	26	29000
104	Abhishek	27	26000
105	Ritik	26	29000
106	Yash	29	28000

Table: Old_Employee

The following INSERT query inserts the record of new employees into the **New_Employee** table:

```
1. INSERT INTO New_Employee (Employee_Id, Employee_Name, Emp_Age, Emp_Salary) VALUES (
   201, Jack, 28, 45000),
2. (202, Berry, 29, 35000),
3. (105, Ritik, 26, 29000),
4. (203, Shyam, 27, 26000),
5. (204, Ritika, 28, 38000),
6. (106, Yash, 29, 28000);
```

The following query shows the details of the **New_Employee** table:

```
1. SELECT * FROM New_Employee;
```

Emp_Id	Emp_Name	Emp_Salary	Emp_City
201	Jack	28	45000
202	Berry	29	35000
105	Ritik	26	29000
203	Shyam	27	26000
204	Ritika	28	38000
106	Yash	29	28000

Table: New_Employee

The following query shows all records of both tables in one table using the UNION operator:

```
1. SELECT * FROM Old_EmployeeUNION SELECT * FROM New_Employee;
```

Output:

Employee_Id	Employee_Name	Emp_Age	Emp_Salary
101	Akhil	28	25000
102	Abhay	27	26000
103	Sorya	26	29000

104	Abhishek	27	26000
105	Ritik	26	29000
106	Yash	29	28000
201	Jack	28	45000
202	Berry	29	35000
203	Shyam	27	26000
204	Ritika	28	38000

Where Clause with the UNION operator

The WHERE clause can also be used with UNION operator to filter the records from one or both tables.

Syntax of UNION with WHERE clause

```
1. SELECT Column_Name_1, Column_Name_2 ...., Column_NameN FROM Table_Name_1 [WHERE c
   ondition]
2. UNION
3. SELECT Column_Name1, Column_Name_2 ...., Column_Name_N FROM Table_Name_2 [WHERE c
   ondition];
```

Example of UNION with WHERE Clause

The following query shows those records of employees from the above tables whose salary is greater than and equal to 29000:

```
1. SELECT * FROM Old_Employee WHERE Emp_Salary >= 29000UNION SELECT * FROM New_Em
   ployee WHERE Emp_Salary >= 29000;
```

Output:

Employee_Id	Employee_Name	Emp_Age	Emp_Salary
103	Sorya	26	29000
105	Ritik	26	29000
201	Jack	28	45000
202	Berry	29	35000
204	Ritika	28	38000

Union ALL Operator in SQL

The SQL Union ALL Operator is same as the UNION operator, but the only difference is that UNION ALL operator also shows the common rows in the result.

Syntax of UNION ALL Set operator:

```
SELECT Column_Name_1, Column_Name_2 ...., Column_Name_N FROM Table_Name_1 [WHERE condition]
UNION ALL
SELECT Column_Name_1, Column_Name_2 ...., Column_Name_N FROM Table_Name_2 [WHERE condition];
```

Example of UNION ALL

Let's create two different tables and insert records in both tables.

The following query creates the **Passed_Students** table with four fields:

```
CREATE TABLE Passed_Students
(
Student_Id INT NOT NULL,
Student_Name Varchar (40),
Student_Age INT,
Student_Marks INT
);
```

The following query creates the **New_Students** table with four fields:

```
CREATE TABLE New_Students
(
Student_Id INT NOT NULL,
Student_Name Varchar (40),
Student_Age INT,
Student_Marks INT
);
```

The following INSERT query inserts the record of passed students into the Passed_Students table:

```
INSERT INTO Passed_Students (Student_Id, Student_Name, Student_Age, Student_Marks) VALUES (101, Akhil, 28, 95),
(102, Abhay, 27, 86),
(103, Sorya, 26, 79),
(104, Abhishek, 27, 66),
(105, Ritik, 26, 79),
(106, Yash, 29, 88);
```

The following query shows the details of the Passed_Students table:

```
SELECT * FROM Passed_Students;
```

Student_Id	Student_Name	Student_Age	Student_Marks
101	Akhil	28	95
102	Abhay	27	86

103	Sorya	26	79
104	Abhishek	27	66
105	Ritik	26	79
106	Yash	29	88

Table: Passed_Students

The following INSERT query inserts the record of new students into the New_Students table:

```
INSERT INTO New_Students (Student_Id, Student_Name, Student_Age, Student_Marks)  VALUES (201, Jack, 28, 77),
(202, Berry, 29, 68),
(105, Ritik, 26, 82),
(203, Shyam, 27, 70),
(204, Ritika, 28, 99),
(106, Yash, 29, 86);
```

The following query shows the details of the **New_Students** table:

```
SELECT * FROM New_Students;
```

Student_Id	Student_Name	Student_Age	Student_Marks
201	Jack	28	77
202	Berry	29	66
105	Ritik	26	82

203	Shyam	27	70
204	Ritika	28	99
106	Yash	29	86

Table: New_Students

The following query shows all duplicate and unique records from both tables:

1. **SELECT** * **FROM** Passed_StudentsUNION ALL **SELECT** * **FROM** New_Students;

Output:

Student_Id	Student_Name	Student_Age	Student_Marks
101	Akhil	28	95
102	Abhay	27	86
103	Sorya	26	79
104	Abhishek	27	66
105	Ritik	26	79
106	Yash	29	88
201	Jack	28	77
202	Berry	29	68

105	Ritik	26	82
203	Shyam	27	70
204	Ritika	28	99
106	Yash	29	86

Cursor in SQL Server

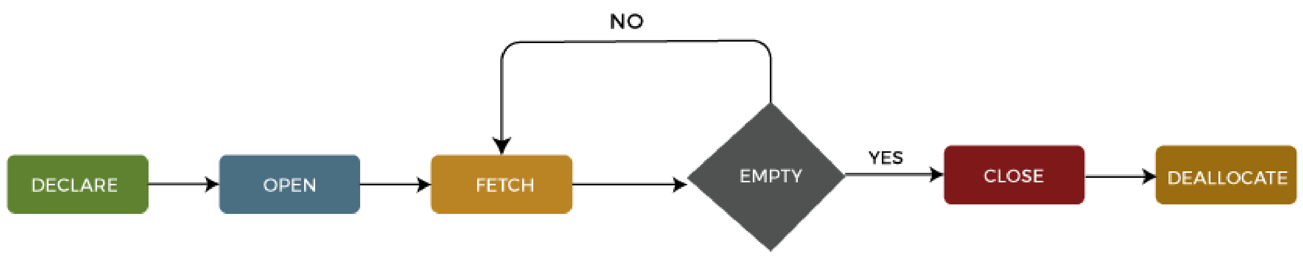
A cursor in SQL Server is a d**atabase object that allows us to retrieve each row at a time and manipulate its data**. A cursor is nothing more than a pointer to a row. It's always used in conjunction with a SELECT statement. It is usually a collection of SQL logic that loops through a predetermined number of rows one by one. A simple illustration of the cursor is when we have an extensive database of worker's records and want to calculate each worker's salary after deducting taxes and leaves.

The SQL Server **cursor's purpose is to update the data row by row, change it, or perform calculations that are not possible when we retrieve all records at once**. It's also useful for performing administrative tasks like SQL Server database backups in sequential order. Cursors are mainly used in the development, DBA, and ETL processes.

This article explains everything about SQL Server cursor, such as cursor life cycle, why and when the cursor is used, how to implement cursors, its limitations, and how we can replace a cursor.

Life Cycle of the cursor

We can describe the life cycle of a cursor into the **five different sections** as follows:



1: Declare Cursor

The first step is to declare the cursor using the below SQL statement:

```
DECLARE cursor_name CURSOR
   FOR select_statement;
```

We can declare a cursor by specifying its name with the data type CURSOR after the DECLARE keyword. Then, we will write the SELECT statement that defines the output for the cursor.

2: Open Cursor

It's a second step in which we open the cursor to store data retrieved from the result set. We can do this by using the below SQL statement:

```
OPEN cursor_name;
```

3: Fetch Cursor

It's a third step in which rows can be fetched one by one or in a block to do data manipulation like insert, update, and delete operations on the currently active row in the cursor. We can do this by using the below SQL statement:

```
FETCH NEXT FROM cursor INTO variable_list;
```

We can also use the **@@FETCHSTATUS function** in SQL Server to get the status of the most recent FETCH statement cursor that was executed against the cursor. The **FETCH** statement was successful when the @@FETCHSTATUS gives zero output. The **WHILE** statement can be used to retrieve all records from the cursor. The following code explains it more clearly:

```
WHILE @@FETCH_STATUS = 0
   BEGIN
      FETCH NEXT FROM cursor_name;
   END;
```

4: Close Cursor

It's a fourth step in which the cursor should be closed after we finished work with a cursor. We can do this by using the below SQL statement:

```sql
CLOSE cursor_name;
```

5: Deallocate Cursor

It is the fifth and final step in which we will erase the cursor definition and release all the system resources associated with the cursor. We can do this by using the below SQL statement:

```sql
DEALLOCATE cursor_name;
```

Uses of SQL Server Cursor

We know that relational database management systems, including SQL Server, are excellent in handling data on a set of rows called result sets. **For example**, we have a table **product_table** that contains the product descriptions. If we want to update the **price** of the product, then the below **'UPDATE'** query will update all records that match the condition in the **'WHERE'** clause:

```sql
UPDATE product_table SET unit_price = 100 WHERE product_id = 105;
```

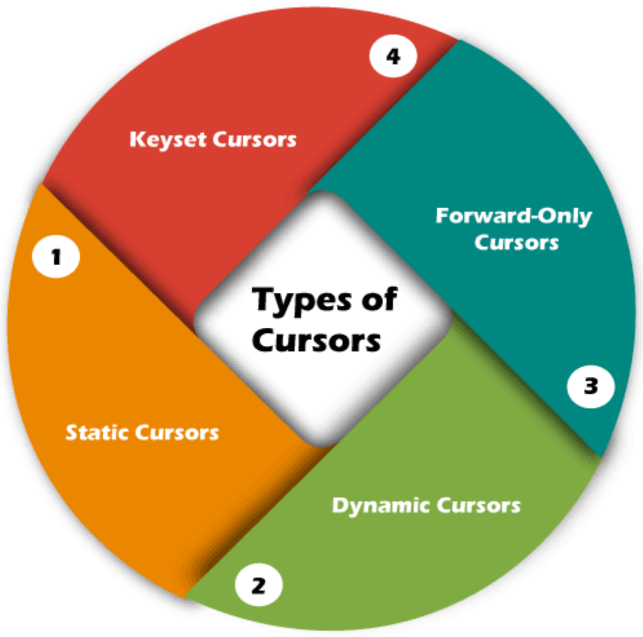
Sometimes the application needs to process the rows in a singleton fashion, i.e., on row by row basis rather than the entire result set at once. We can do this process by using cursors in SQL Server. Before using the cursor, we must know that cursors are very bad in performance, so it should always use only when there is no option except the cursor.

The cursor uses the same technique as we use loops like FOREACH, FOR, WHILE, DO WHILE to iterate one object at a time in all programming languages. Hence, it could be chosen because it applies the same logic as the programming language's looping process.

Types of Cursors in SQL Server

The following are the different types of cursors in SQL Server listed below:

- Static Cursors
- Dynamic Cursors
- Forward-Only Cursors
- Keyset Cursors



Static Cursors

The result set shown by the static cursor is always the same as when the cursor was first opened. Since the static cursor will store the result in **tempdb**, they are always **read-only**. We can use the static cursor to move both forward and backward. In contrast to other cursors, it is slower and consumes more memory. As a result, we can use it only when scrolling is necessary, and other cursors aren't suitable.

This cursor shows rows that were removed from the database after it was opened. A static cursor does not represent any INSERT, UPDATE, or DELETE operations (unless the cursor is closed and reopened).

Dynamic Cursors

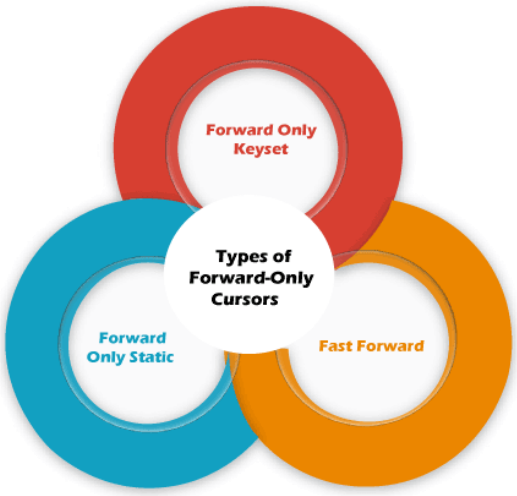
The dynamic cursors are opposite to the static cursors that allow us to perform the data updation, deletion, and insertion operations while the cursor is open. It is **scrollable by default**. It can detect all changes made to the rows, order, and values in the result set, whether the changes occur inside the cursor or outside the cursor. Outside the cursor, we cannot see the updates until they are committed.

Forward-Only Cursors

It is the default and fastest cursor type among all cursors. It is called a forward-only cursor because it **moves only forward through the result set**. This cursor doesn't support scrolling. It can only retrieve rows from the beginning to the end of the result set. It allows us to perform insert, update, and delete operations. Here, the effect of insert, update and delete operations made by the user that affect rows in the result set are visible as the rows are fetched from the cursor. When the row was fetched, we cannot see the changes made to rows through the cursor.

The Forward-Only cursors are three categorize into three types:

1. Forward_Only Keyset
2. Forward_Only Static
3. Fast_Forward



Keyset Driven Cursors

This cursor functionality **lies between a static and a dynamic cursor** regarding its ability to detect changes. It can't always detect changes in the result set's membership and order like a static cursor. It can detect changes in the result set's rows values as like a dynamic cursor. It can only **move from the first to last and last to the first row**. The order and the membership are fixed whenever this cursor is opened.

It is operated by a set of unique identifiers the same as the keys in the keyset. The keyset is determined by all rows that qualified the SELECT statement when the cursor was first opened. It can also detect any changes to the data source, which supports update and delete operations. It is scrollable by default.

Implementation of Example

Let us implement the cursor example in the SQL server. We can do this by first creating a table named "**customer**" using the below statement:

```
CREATE TABLE customer (
  id int PRIMARY KEY,
  c_name nvarchar(45) NOT NULL,
  email nvarchar(45) NOT NULL,
  city nvarchar(25) NOT NULL
);
```

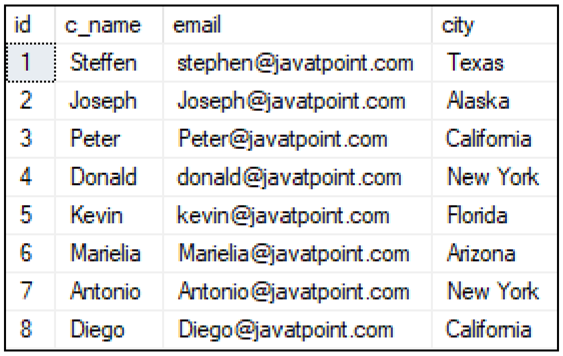
Next, we will insert values into the table. We can execute the below statement to add data into a table:

```
INSERT INTO customer (id, c_name, email, city)
VALUES (1,'Steffen', 'stephen@javatpoint.com', 'Texas'),
(2, 'Joseph', 'Joseph@javatpoint.com', 'Alaska'),
(3, 'Peter', 'Peter@javatpoint.com', 'California'),
(4,'Donald', 'donald@javatpoint.com', 'New York'),
(5, 'Kevin', 'kevin@javatpoint.com', 'Florida'),
(6, 'Marielia', 'Marielia@javatpoint.com', 'Arizona'),
(7,'Antonio', 'Antonio@javatpoint.com', 'New York'),
(8, 'Diego', 'Diego@javatpoint.com', 'California');
```

We can verify the data by executing the **SELECT** statement:

```
SELECT * FROM customer;
```

After executing the query, we can see the below output where we have **eight rows** into the table:



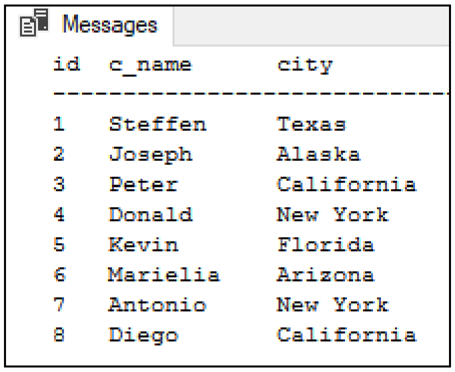
id	c_name	email	city
1	Steffen	stephen@javatpoint.com	Texas
2	Joseph	Joseph@javatpoint.com	Alaska
3	Peter	Peter@javatpoint.com	California
4	Donald	donald@javatpoint.com	New York
5	Kevin	kevin@javatpoint.com	Florida
6	Marielia	Marielia@javatpoint.com	Arizona
7	Antonio	Antonio@javatpoint.com	New York
8	Diego	Diego@javatpoint.com	California

Now, we will create a cursor to display the customer records. The below code snippets explain the all steps of the cursor declaration or creation by putting everything together:

```sql
--Declare the variables for holding data.
DECLARE @id INT, @c_name NVARCHAR(50), @city NVARCHAR(50)

--Declare and set counter.
DECLARE @Counter INT
SET @Counter = 1

--Declare a cursor
DECLARE PrintCustomers CURSOR
FOR
SELECT id, c_name, city FROM customer

--Open cursor
OPEN PrintCustomers

--Fetch the record into the variables.
FETCH NEXT FROM PrintCustomers INTO
@id, @c_name, @city

--LOOP UNTIL RECORDS ARE AVAILABLE.
WHILE @@FETCH_STATUS = 0
   BEGIN
      IF @Counter = 1
      BEGIN
         PRINT 'id' + CHAR(9) + 'c_name' + CHAR(9) + CHAR(9) + 'city'
         PRINT '--------------------------'
      END

      --Print the current record
      PRINT CAST(@id AS NVARCHAR(10)) + CHAR(9) + @c_name + CHAR(9) + CHAR(9) + @city

      --Increment the counter variable
      SET @Counter = @Counter + 1
```

```sql

        --Fetch the next record into the variables.
        FETCH NEXT FROM PrintCustomers INTO
        @id, @c_name, @city
    END

--Close the cursor
CLOSE PrintCustomers

--Deallocate the cursor
DEALLOCATE PrintCustomers
```

After executing a cursor, we will get the below output:



Messages

```
id  c_name      city
-----------------------------
1   Steffen     Texas
2   Joseph      Alaska
3   Peter       California
4   Donald      New York
5   Kevin       Florida
6   Marielia    Arizona
7   Antonio     New York
8   Diego       California
```

Limitations of SQL Server Cursor

A cursor has some limitations so that it should always use only when there is no option except the cursor. These limitations are:

- Cursor consumes network resources by requiring a network roundtrip each time it fetches a record.
- A cursor is a memory resident set of pointers, which means it takes some memory that other processes could use on our machine.
- It imposes locks on a portion of the table or the entire table when processing data.
- The cursor's performance and speed are slower because they update table records one row at a time.
- Cursors are quicker than while loops, but they do have more overhead.

- The number of rows and columns brought into the cursor is another aspect that affects cursor speed. It refers to how much time it takes to open your cursor and execute a fetch statement.

How can we avoid cursors?

The main job of cursors is to traverse the table row by row. The easiest way to avoid cursors are given below:

Using the SQL while loop

The easiest way to avoid the use of a cursor is by using a while loop that allows the inserting of a result set into the temporary table.

User-defined functions

Sometimes cursors are used to calculate the resultant row set. We can accomplish this by using a user-defined function that meets the requirements.

Using Joins

Join processes only those columns that meet the specified condition and thus reduces the lines of code that give faster performance than cursors in case huge records need to be processed.

What is Embedded SQL?

As we have seen in our previous tutorials, SQL is known as the Structured Query Language. It is the language that we use to perform operations and transactions on the databases.

When we talk about industry-level applications we need properly connected systems which could draw data from the database and present to the user. In such cases, the embedded SQL comes to our rescue.

We embed SQL queries into high-level languages such that they can easily perform the logic part of our analysis.

Some of the prominent examples of languages with which we embed SQL are as follows:

- C++
- Java
- Python etc.

Why do we need Embedded SQL?

Embedded SQL gives us the freedom to use databases as and when required. Once the application we develop goes into the production mode several things need to be taken care of.

We need to take care of a thousand things out of which one major aspect is the problem of authorization and fetching and feeding of data into/from the database.

With the help of the embedding of queries, we can easily use the database without creating any bulky code. With the embedded SQL, we can create API's which can easily fetch and feed data as and when required.