**`SQL (Structured Query Language):**

SQL is a standard language for accessing and manipulating databases.

**What is SQL?**

* SQL stands for Structured Query Language
* SQL lets you access and manipulate databases
* SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

**What Can SQL do?**

* SQL can execute queries against a database
* SQL can retrieve data from a database
* SQL can insert records in a database
* SQL can update records in a database
* SQL can delete records from a database
* SQL can create new databases
* SQL can create new tables in a database
* SQL can create stored procedures in a database
* SQL can create views in a database
* SQL can set permissions on tables, procedures, and views

**Using SQL in Your Web Site**

To build a web site that shows data from a database, you will need:

* An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
* To use a server-side scripting language, like PHP or ASP
* To use SQL to get the data you want
* To use HTML / CSS to style the page

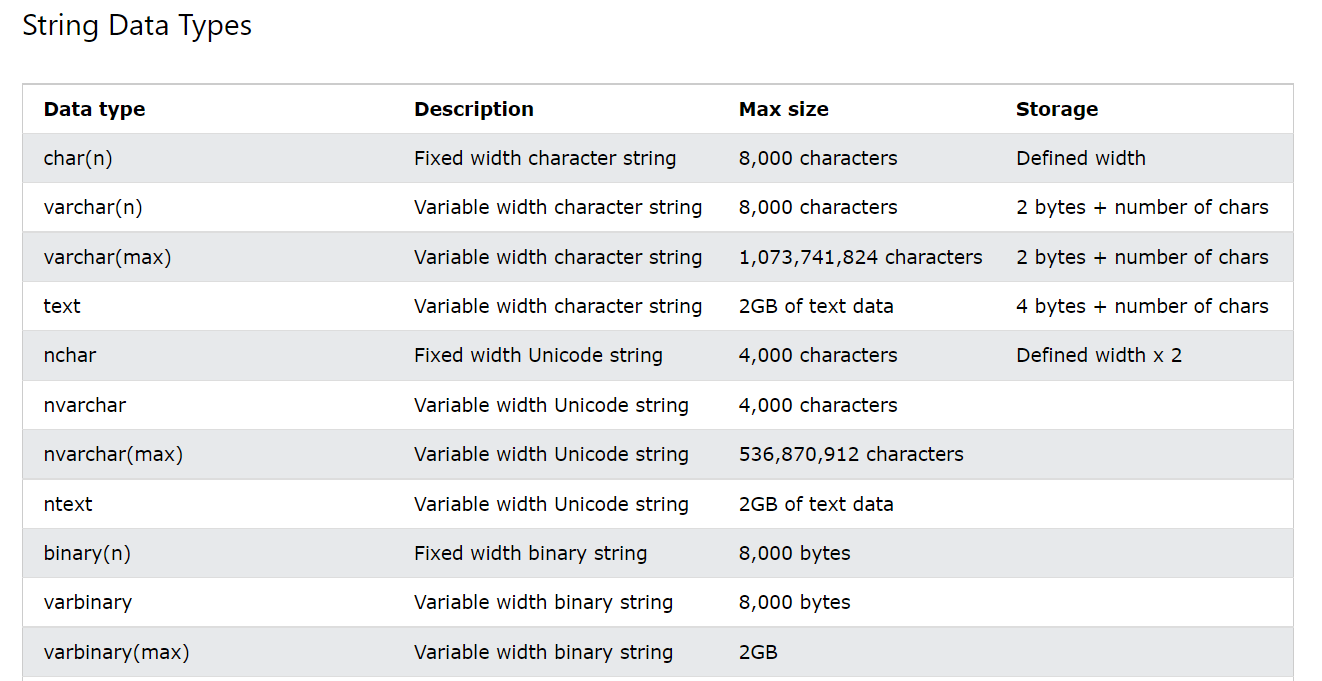
**What is Data :**

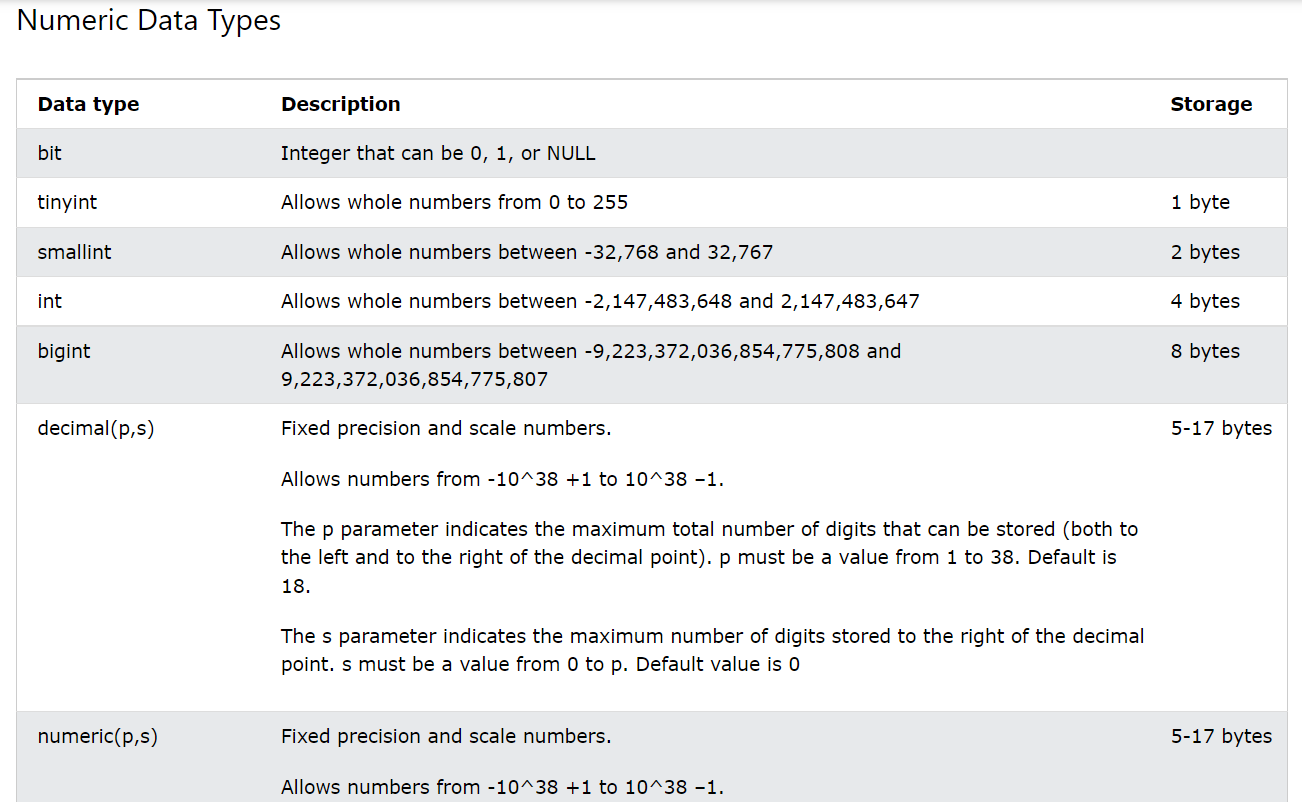
Data is nothing but information about anything.

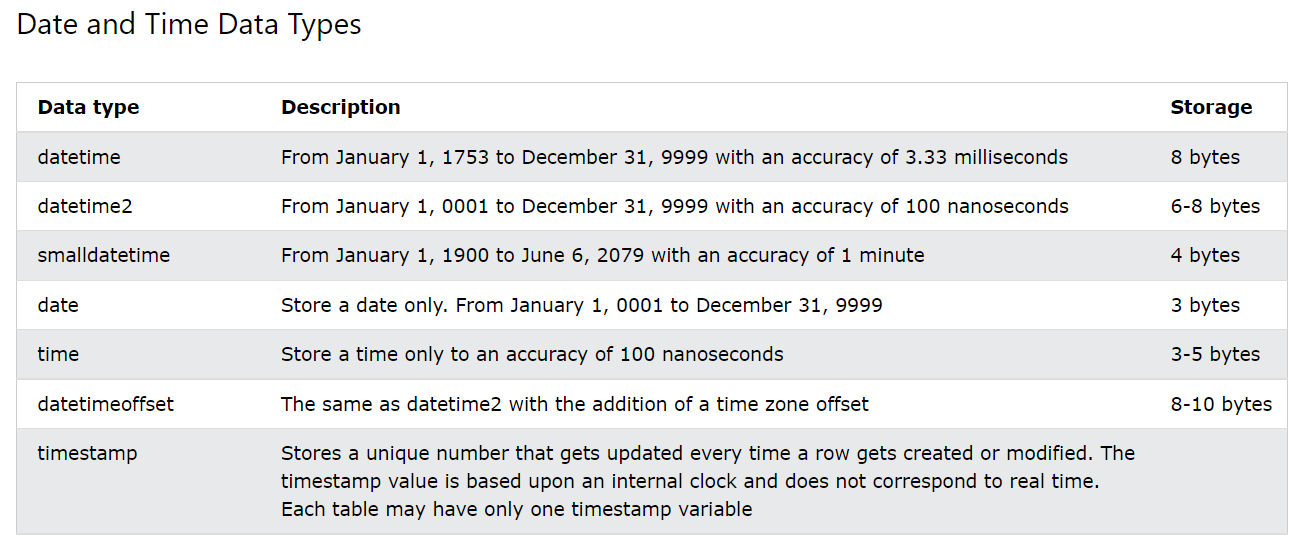
**DataTypes :**

A data type is an attribute that specifies the type of data that the object can hold: integer data, character data, monetary data, date and time data, binary strings, and so on. SQL Server supplies a set of system data types that define all the types of data that can be used with SQL Server.

Link : [://www.w https 3schools.com/sql/sql\_datatypes.asp](https://www.w3schools.com/sql/sql_datatypes.asp)







# **What Is a Database?**

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a [database management system (DBMS)](https://www.oracle.com/in/database/what-is-database/#WhatIsDBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

Data within the most common types of databases in operation today is typically modelled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data.

## RDBMS

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows

**TABLE** :

Every table is broken up into smaller entities called fields. The fields in the Customers table consist of CustomerID, CustomerName, ContactName, Address, City, PostalCode and Country. A field is a column in a table that is designed to maintain specific information about every record in the table.

**Table can have only 1024 columns.**

A record, also called a row, is each individual entry that exists in a table. For example, there are 91 records in the above Customers table. A record is a horizontal entity in a table.

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

**Connecting to SQL Server**

**SQL Server Management Studio (SSMS),** is the client tool that can be used to write and execute SQL queries. To connect to the SQL Server Management Studio

1. Click Start   
2. Select All Programs  
3. Select Microsoft SQL Server 2005, 2008, or 2008 R2 (Depending on the version installed)  
4. Select SQL Server Management Studio

**You will now see, Connect to Server window.**  
1. Select Database Engine as the Server Type. The other options that you will see here are Analysis Services (SSAS), Reporting Services (SSRS) and Integration Services (SSIS).  
**Server type = Database Engine**

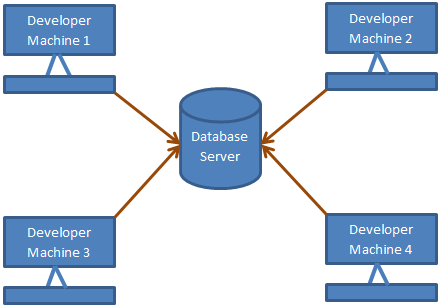
2. Next you need to specify the Server Name. Here we can specify the name or the server or IP Address. If you have SQL Server installed on your local machine, you can specify, (local) or just . (Period) or 127.0.0.1  
**Server name = (local)**

3. Now select Authentication. The options available here, depends on how you have installed SQL Server. During installation, if you have chosen mixed mode authentication, you will have both Windows Authentication and SQL Server Authentication. Otherwise, you will just be able to connect using windows authentication.

4. If you have chosen Windows Authentication, you don't have to enter user name and password, otherwise enter the user’s name and password and click connect.

You should now be connected to SQL Server🡪 New Query

SSMS is a client tool and not the Server by itself. Usually database server (SQL Server), will be on a dedicated machine, and developers connect to the server using SSMS from their respective local (development) computers.  
  
Developer Machines 1,2,3 and 4 connects to the database server using SSMS.

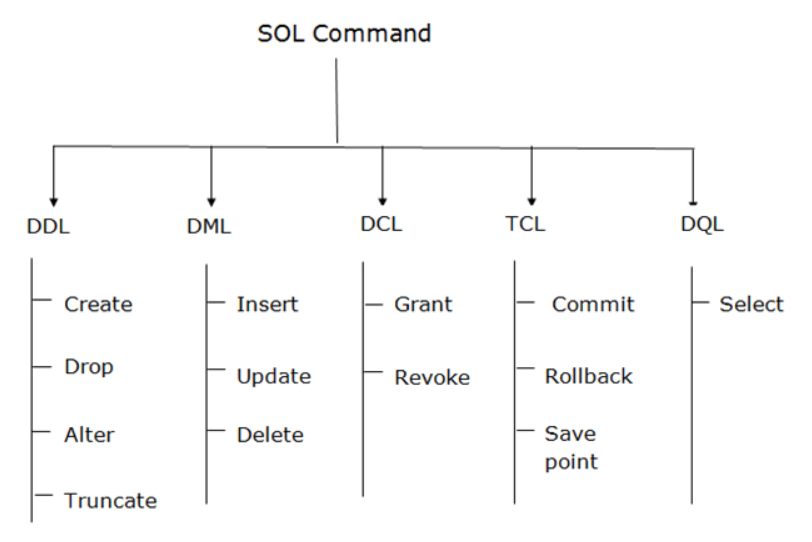


# **SQL Commands**

* 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/database/stored procedure/views in the database.

**Syntax:**

CREATE TABLE **TABLE\_NAME** (COLUMN\_NAME DATATYPES[,....]);

CREATE Procedure **Proc\_Name**

Create Database **Database\_Name**

**Exam:** CREATE TABLE EMPLOYEE(EmpId int, Name VARCHAR(20), Email VARCHAR(100), DOB DATE);

**b. 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:** This command is used to modify structure of existing database object like Table/Proc/Function etc

To add a new column in the table

ALTER TABLE table\_name ADD column\_name COLUMN-definition;

Exam : ALTER TABLE STU\_DETAILS ADD(ADDRESS VARCHAR2(20));

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

Syntax : DROP TABLE table\_name;

Drop Database Database\_Name,

Drop Proc Proc\_Name

Example: DROP TABLE EMPLOYEE;

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

Syntax : TRUNCATE TABLE table\_name;

Example : 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);

**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 CONDITION]

**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:**

1. DELETE FROM table\_name [WHERE condition];

**For example:**

DELETE FROM javatpoint WHERE Author="Sonoo";

### **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**

GRANT SELECT, UPDATE ON MY\_TABLE TO SOME\_USER, ANOTHER\_USER;

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

**Example**

REVOKE SELECT, UPDATE ON MY\_TABLE FROM USER1, USER2;

### **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:** COMMIT;

**Example:**

DELETE FROM CUSTOMERS WHERE AGE = 25;

COMMIT;

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

Syntax: ROLLBACK;

**Example:**

DELETE FROM CUSTOMERS WHERE AGE = 25;

ROLLBACK;

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

**Syntax:**

SAVEPOINT SAVEPOINT\_NAME;

**5. Data Query Language**

DQL is used to fetch the data from the database TABLE.

It uses only one command:

* 1. **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:**

SELECT Columns FROM TABLES

WHERE conditions;

**For example:** SELECT emp\_name FROM employee WHERE age > 20;

# **SQL Constraints**

SQL constraints are used to specify rules for data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

* [NOT NULL](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value
* [UNIQUE](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different (more than one UK, can accept only one null)
* [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table (only 1 PK per table)
* [FOREIGN KEY](https://www.w3schools.com/sql/sql_foreignkey.asp) - Prevents actions that would destroy links between tables
* [CHECK](https://www.w3schools.com/sql/sql_check.asp) - Ensures that the values in a column satisfies a specific condition
* [DEFAULT](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column if no value is specified

NOTE: Constraints can be added to column while table creation or we can add constraint to column after creating table using Alter Table command.

**Syntax:** Alter Table Table\_Name

Add Constraint Constraint\_Name Constraint\_Defination

# **SQL AUTO INCREMENT Field**

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

The following SQL statement defines the "Personid" column to be an auto-increment primary key field in the "Persons" table:

CREATE TABLE Persons (  
    Personid int IDENTITY(1,1) PRIMARY KEY,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int);

The MS SQL Server uses the IDENTITY keyword to perform an auto-increment feature.

In the example above, the starting value for IDENTITY is 1, and it will increment by 1 for each new record.

**Tip:** To specify that the "Personid" column should start at value 10 and increment by 5, change it to IDENTITY(10,5).

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "Personid" column (a unique value will be added automatically):

INSERT INTO Persons (FirstName,LastName)  
VALUES ('Lars','Monsen');

The SQL statement above would insert a new record into the "Persons" table. The "Personid" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

<https://www.w3schools.com/sql/sql_ref_sqlserver.asp>

## The SQL SELECT Statement

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

The data returned is stored in a result table, called the result-set.

### **SELECT Syntax**

SELECT column1, column2, ...  
FROM table\_name;

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

Select \* From Table\_Name

## SELECT Column Example

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:

### **Example**

SELECT CustomerName, City FROM Customers;

## Use of TOP and DISTINCT in SQL SERVER

**TOP Clause:**

*TOP*is mainly used to limit the result of query in terms of number or percentage of the result from database.  It selects result limits to TOP from N numbers of data from ordered rows by using*ORDER BY*statement otherwise it selects undefined order data. *TOP*can be used in statements *SELECT*, *UPDATE*, *DELETE*, *INSERT*, or *MERGE*statement.

**Syntax:**

SELECT TOP <expression> [PERCENT]

FROM <table\_name>

ORDER BY <column\_name>

## You can use clauses like *WHERE*, *JOIN*, *HAVING*and *GROUP BY* with *SELECT*statement.

**Using *TOP*in constant value:**

Constant value to return the top 10.

SELECT TOP 10 First\_Name, Salary FROM EmployeeDb

ORDER BY First\_Name,Salary DESC

***TOP*to return *PERCENT*of *ROWS***

Returns percent of total results of rows:

SELECT TOP 1 PERCENT First\_Name, Salary FROM EmployeeDb

ORDER BY First\_Name DESC

**DISTINCT clause**

It selects only different value which eliminates duplicate record form the result. It only operates on single column and can be used in COUNT, AVG, MAX, etc. while using DISTINCT on column with multiple NULL values, it returns only one NULL as it treats multiple NULL as one distinct value. DINSTINT cannot be operated on multiple columns.

**In single column:**

Syntax:

SELECT DINTINCT <column\_name> FROM <table\_name>

It returns distinct values from the specified column.

SELECT DISTINCT First\_Name FROM EmployeeDb ORDER BY First\_Name

**Use of *DISTINCT*and *TOP***

Implementation of DISTINCT and TOP in a column. It also cannot be operated on multiple columns.

Example:

SELECT DISTINCT TOP 10 First\_Name FROM EmployeeDb ORDER BY First\_Name

## The SQL WHERE Clause

The WHERE clause is used to filter records.

It is used to extract only those records that fulfil a specified condition.

### **WHERE Syntax**

SELECT *column1*,*column2, ...*  
FROM *table\_name*  
WHERE *condition*;

**Note:** The WHERE clause is not only used in SELECT statements, it is also used in UPDATE, DELETE, etc.!

## WHERE Clause Example

The following SQL statement selects all the customers from the country "Mexico", in the "Customers" table:

### **Example**

SELECT \* FROM Customers  
WHERE Country='Mexico';

## Text Fields vs. Numeric Fields

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

### **Example**

SELECT \* FROM Customers  
WHERE CustomerID=1;

## Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

= Equal

< Less than > Greater than

<= Less than equal to >= Greater than equal to

<> Not Equal to

Between – Between a certain range

Like - Search for a pattern

In - To specify multiple possible values for a column

## SQL Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the [LIKE](https://www.w3schools.com/sql/sql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

### **Wildcard Characters in SQL Server**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Description** | **Example** |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents any single character within the specified range | c[a-b]t finds cat and cbt |

All the wildcards can also be used in combinations!

Here are some examples showing different LIKE operators with '%' and '\_' wildcards:

|  |  |
| --- | --- |
| **LIKE Operator** | **Description** |
| WHERE CustomerName LIKE 'a%' | Finds any values that starts with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that ends with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |
| WHERE CustomerName LIKE '\_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a\_\_%' | Finds any values that starts with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that starts with "a" and ends with "o" |

## The SQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

### **ORDER BY Syntax**

SELECT column1, column2, ...  
FROM table\_name  
ORDER BY column1, column2, ... ASC|DESC;

## ORDER BY Example

## The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

### **Example**

SELECT \* FROM Customers  
ORDER BY Country; Order by clause can be used with several columns also.

## SQL Aliases

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of that query.

An alias is created with the AS keyword.

SELECT column\_name AS alias\_name  
FROM table\_name;

# **SQL Server Functions**

SQL Server has many built-in functions.

This reference contains string, numeric, date, conversion, and some advanced functions in SQL Server.

## SQL Server String Functions

|  |  |
| --- | --- |
| **Function** | **Description** |
|  |  |
|  |  |
| [CHARINDEX](https://www.w3schools.com/sql/func_sqlserver_charindex.asp) | Returns the position of a substring in a string |
| [CONCAT](https://www.w3schools.com/sql/func_sqlserver_concat.asp) | Adds two or more strings together |
| [Concat with](https://www.w3schools.com/sql/func_sqlserver_concat_with_plus.asp) + | Adds two or more strings together |
| [DATALENGTH](https://www.w3schools.com/sql/func_sqlserver_datalength.asp) | Returns the number of bytes used to represent an expression |
| [DIFFERENCE](https://www.w3schools.com/sql/func_sqlserver_difference.asp) | Compares two SOUNDEX values, and returns an integer value |
| [FORMAT](https://www.w3schools.com/sql/func_sqlserver_format.asp) | Formats a value with the specified format |
| [LEFT](https://www.w3schools.com/sql/func_sqlserver_left.asp) | Extracts a number of characters from a string (starting from left) |
| [LEN](https://www.w3schools.com/sql/func_sqlserver_len.asp) | Returns the length of a string |
| [LOWER](https://www.w3schools.com/sql/func_sqlserver_lower.asp) | Converts a string to lower-case |
| [LTRIM](https://www.w3schools.com/sql/func_sqlserver_ltrim.asp) | Removes leading spaces from a string |
| [NCHAR](https://www.w3schools.com/sql/func_sqlserver_nchar.asp) | Returns the Unicode character based on the number code |
| [PATINDEX](https://www.w3schools.com/sql/func_sqlserver_patindex.asp) | Returns the position of a pattern in a string |
| [QUOTENAME](https://www.w3schools.com/sql/func_sqlserver_quotename.asp) | Returns a Unicode string with delimiters added to make the string a valid SQL Server delimited identifier |
| [REPLACE](https://www.w3schools.com/sql/func_sqlserver_replace.asp) | Replaces all occurrences of a substring within a string, with a new substring |
| [REPLICATE](https://www.w3schools.com/sql/func_sqlserver_replicate.asp) | Repeats a string a specified number of times |
| [REVERSE](https://www.w3schools.com/sql/func_sqlserver_reverse.asp) | Reverses a string and returns the result |
| [RIGHT](https://www.w3schools.com/sql/func_sqlserver_right.asp) | Extracts a number of characters from a string (starting from right) |
| [RTRIM](https://www.w3schools.com/sql/func_sqlserver_rtrim.asp) | Removes trailing spaces from a string |
| [SOUNDEX](https://www.w3schools.com/sql/func_sqlserver_soundex.asp) | Returns a four-character code to evaluate the similarity of two strings |
| [SPACE](https://www.w3schools.com/sql/func_sqlserver_space.asp) | Returns a string of the specified number of space characters |
| [STR](https://www.w3schools.com/sql/func_sqlserver_str.asp) | Returns a number as string |
| [STUFF](https://www.w3schools.com/sql/func_sqlserver_stuff.asp) | Deletes a part of a string and then inserts another part into the string, starting at a specified position |
| [SUBSTRING](https://www.w3schools.com/sql/func_sqlserver_substring.asp) | Extracts some characters from a string |
| [TRANSLATE](https://www.w3schools.com/sql/func_sqlserver_translate.asp) | Returns the string from the first argument after the characters specified in the second argument are translated into the characters specified in the third argument. |
| [TRIM](https://www.w3schools.com/sql/func_sqlserver_trim.asp) | Removes leading and trailing spaces (or other specified characters) from a string |
| [UNICODE](https://www.w3schools.com/sql/func_sqlserver_unicode.asp) | Returns the Unicode value for the first character of the input expression |
| [UPPER](https://www.w3schools.com/sql/func_sqlserver_upper.asp) | Converts a string to upper-case |

## 1.ASCII - Return the ASCII value of the first character in "Shivraj":

## SELECT ASCII(‘Shivraj’) AS NumCodeOfFirstChar FROM Customers

## 2.CHAR - The CHAR() function returns the character based on the ASCII code.

## Syntax : CHAR(code)

## Example : SELECT CHAR(65) AS CodeToCharacter;

3.The **CHARINDEX()** function searches for a substring in a string, and returns the position.

If the substring is not found, this function returns 0.

Note: This function performs a case-insensitive search.

**Syntax** : CHARINDEX(substring, string, start)

Example : SELECT CHARINDEX('a', 'Amit') as 'xyz'

## 4.The CONCAT() function adds two or more strings together.

## CONCAT(string1, string2, ...., string\_n) WE CAN USE + ALSO

## Syntax : SELECT CONCAT('Shivraj',' ','Wankar') AS FullName

## Example : SELECT ('Shivraj'+' '+'Wankar') AS FullName -- Concat using Plus(+) sign

5.The **DATALENGTH()** function returns the number of bytes used to represent an expression.

Note: The **DATALENGTH()** function counts both leading and trailing spaces when calculating the length of the expression.

## Syntax : DATALENGTH(expression)

## Example : SELECT DATALENGTH(‘Shivraj’) AS CharacterLength

## 7. The SOUNDEX() function accepts a string and converts it to a four-character code based on how the string sounds when it is spoken.

8. The **DIFFERENCE()** function compares two SOUNDEX values, and returns an integer. The integer value indicates the match for the two SOUNDEX values, from 0 to 4.

0 indicates weak or no similarity between the SOUNDEX values. 4 indicates strong similarity or identically SOUNDEX values.

## 9. The LEN() function returns the length of a string.

Syntax : LEN(string)

Example : SELECT LEN(' W3Schools.com ');

Note: Trailing spaces at the end of the string is not included when calculating the length. However, leading spaces at the start of the string is included when calculating the length.

## 10. The LOWER() function converts a string to lower-case.

## Syntax : LOWER(text)

## Example : SELECT lower('Shivraj') AS CharacterLength

## 11. The UPPER() function converts a string to lower-case.

## Syntax : Upper(Text)

## SELECT upper('Shivraj') AS CharacterLength

## 12. The RIGHT() function extracts a number of characters from a string (starting from right).

## And The LEFT() function extracts a number of characters from a string (starting from left).

## Syntax : RIGHT(string, number\_of\_chars) and Left(string, number\_of\_chars)

## Example : SELECT LEFT(Name,3),Name, right(Name,2) FROM Departments

## 13. The LTRIM() function removes leading spaces from a string.

## The RTRIM() function removes trailing spaces from a string.

The **TRIM()** function removes the space character OR other specified characters from the start or end of a string.

By default, the TRIM() function removes leading and trailing spaces from a string.

## Syntax : TRIM([characters FROM ]string) or LTRIM([characters FROM ]string)

**Example :** SELECT trim(' SQL Tutorial! ') AS TrimmedString;

## SELECT REPLACE('SQL Tutorial SQL SERVER', 'SQL', 'HTML');

**14.** The **REPLACE()** function replaces all occurrences of a substring within a string, with a new substring.

Note: The search is case-insensitive.

## Syntax : REPLACE(string, old\_string, new\_string)

## Example : SELECT REPLACE(LASTNAME,'WANKAR','Jadhav') AS ReplacedLastName,LastName,FirstName,Age FROM Persons

## 15. The REPLICATE() function repeats a string a specified number of times.

## Syntax : REPLICATE(string, integer)

## Example : SELECT REPLICATE(CustomerName, 2)FROM Customers;

## 16. The SUBSTRING() function extracts some characters from a string.

## Syntax : SUBSTRING(string, start, length)

## Example : SELECT SUBSTRING(NAME,1,8),NAME FROM Departments

## 17. The STUFF() function deletes a part of a string and then inserts another part into the string, starting at a specified position.

## Syntax : STUFF(string, start, length, new\_string)

## Example : SELECT STUFF('SQL Tutorial!', 13, 1, ' is fun!');

18. The **PATINDEX()** function returns the position of a pattern in a string.

If the pattern is not found, this function returns 0.

Note: The search is case-insensitive and the first position in string is 1.

## Syntax : PATINDEX(%pattern%, string)

Example : SELECT PATINDEX('%schools%', 'W3Schools.com');

## SELECT PATINDEX('%s%com%', 'W3Schools.com');

## Note : This are the some of the commonly used string functions. For all the other string function visit below url.

## https://www.w3schools.com/sql/

## SQL Server Math/Numeric Functions

|  |  |
| --- | --- |
| **Function** | **Description** |
| [ABS](https://www.w3schools.com/sql/func_sqlserver_abs.asp) | Returns the absolute value of a number |
| [AVG](https://www.w3schools.com/sql/func_sqlserver_avg.asp) | Returns the average value of an expression |
| [CEILING](https://www.w3schools.com/sql/func_sqlserver_ceiling.asp) | Returns the smallest integer value that is >= a number |
| [COUNT](https://www.w3schools.com/sql/func_sqlserver_count.asp) | Returns the number of records returned by a select query |
| [FLOOR](https://www.w3schools.com/sql/func_sqlserver_floor.asp) | Returns the largest integer value that is <= to a number |
| [LOG](https://www.w3schools.com/sql/func_sqlserver_log.asp) | Returns the natural logarithm of a number, or the logarithm of a number to a specified base |
| [MAX](https://www.w3schools.com/sql/func_sqlserver_max.asp) | Returns the maximum value in a set of values |
| [MIN](https://www.w3schools.com/sql/func_sqlserver_min.asp) | Returns the minimum value in a set of values |
| [POWER](https://www.w3schools.com/sql/func_sqlserver_power.asp) | Returns the value of a number raised to the power of another number |
| [RADIANS](https://www.w3schools.com/sql/func_sqlserver_radians.asp) | Converts a degree value into radians |
| [RAND](https://www.w3schools.com/sql/func_sqlserver_rand.asp) | Returns a random number (0 and 1) |
| [ROUND](https://www.w3schools.com/sql/func_sqlserver_round.asp) | Rounds a number to a specified number of decimal places |
| [SIGN](https://www.w3schools.com/sql/func_sqlserver_sign.asp) | Returns the sign of a number |
| [SQRT](https://www.w3schools.com/sql/func_sqlserver_sqrt.asp) | Returns the square root of a number |
| [SQUARE](https://www.w3schools.com/sql/func_sqlserver_square.asp) | Returns the square of a number |
| [SUM](https://www.w3schools.com/sql/func_sqlserver_sum.asp) | Calculates the sum of a set of values |

## Some Important / commonly used Aggregate functions in SQL Server :

1. The **ABS()** function returns the absolute value of a number.

## Exa - SELECT Abs(-243.5) AS AbsNum;

1. **The AVG()** function returns the average value of an expression.

**Note:** NULL values are ignored.

**Syntax :** AVG(expression)

**Example :** SELECT AVG(Price) AS AveragePrice FROM Products;

1. The **COUNT()** function returns the number of records returned by a select query.

**Note:** NULL values are not counted.

## Syntax : COUNT(expression)

## Example : SELECT COUNT(ProductID) AS NumberOfProducts FROM Products;

1. The **SUM()** function calculates the sum of a set of values.

**Note:** NULL values are ignored.

**Syntax :** SUM(expression)

**Example :** SELECT SUM(Quantity) AS TotalItemsOrdered FROM OrderDetails;

1. The **MIN()** function returns the minimum value in a set of values.

Syntax : MIN(expression)

Example : SELECT MIN(Price) AS SmallestPrice FROM Products;

1. The **MAX()** function returns the maximum value in a set of values.

Syntax : Max(expression)

Example : SELECT MAX(Price) AS LargestPrice FROM Products;

1. The **CEILING()** function returns the smallest integer value that is larger than or equal to a number.

## Syntax : CEILING(number)

## Example : SELECT CEILING(25.1) AS CeilValue; -- Returns 26

## The FLOOR() function returns the largest integer value that is smaller than or equal to a number.

## Syntax : FLOOR(number)

## Example : SELECT FLOOR(25.1) AS CeilValue; -- Returns 25

## The ROUND() function rounds a number to a specified number of decimal places.

## Syntax : ROUND(number, decimals, operation)

## Example : SELECT ROUND(235.415, 1, 2) AS RoundValue;

## The RAND() function returns a random number between 0 (inclusive) and 1 (exclusive).

## Syntax : RAND(seed)

## Example : SELECT RAND()

## 

## SQUARE ( Number ) - Returns the square of the given number. Example: Select SQUARE(9) -- Returns 81 SQRT ( Number ) - SQRT stands for Square Root. This function returns the square root of the given value. Example: Select SQRT(81) -- Returns 9

## The SQL GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

## Syntax :

## SELECT column\_name(s) FROM table\_name WHERE condition GROUP BY column\_name(s)ORDER BY column\_name(s);

## Example : SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country;

## The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

## Having Syntax :

## SELECT column\_name(s) FROM table\_name WHERE condition GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);

## Refer below table for Aggregate function and Group by:

## Table : PRODUCT\_MAST

## Structure : ProductMasterId int primary key identity column, Product varchar(15), Company varchar(15),

## Quantity int, Rate int, Cost int

## 

1. SELECT COUNT(\*)  FROM PRODUCT\_MAST
2. SELECT COUNT(\*)  FROM PRODUCT\_MAST WHERE RATE>=20
3. SELECT COUNT(DISTINCT COMPANY) FROM PRODUCT\_MAST
4. SELECT COMPANY, COUNT(\*) FROM PRODUCT\_MAST  GROUP BY COMPANY
5. SELECT COMPANY, COUNT(\*) FROM PRODUCT\_MAST  GROUP BY COMPANY

HAVING COUNT(\*)>2

1. SELECT SUM(COST)  FROM PRODUCT\_MAST
2. SELECT SUM(COST) FROM PRODUCT\_MAST  WHERE QTY>3;
3. SELECT SUM(COST) FROM PRODUCT\_MAST  WHERE QTY>3  GROUP BY COMPANY
4. SELECT COMPANY, SUM(COST) FROM PRODUCT\_MAST  GROUP BY COMPANY

HAVING SUM(COST)>=170;

1. SELECT AVG(COST)  FROM PRODUCT\_MAST
2. SELECT MAX(RATE) FROM PRODUCT\_MAST
3. SELECT MIN(RATE) FROM PRODUCT\_MAST

Table for Exercise with Sample Data TableName - EmpSalary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **employeenumber** | **Lastname** | **level** | **annual\_salary** | **department** |
| 1056 | Patterson | 10 | 10000 | Finance |
| 1076 | Firrel | 5 | 7000 | Marketing |
| 1088 | Patterson | 10 | 12500 | Finance |
| 1102 | Bondur | 2 | 5000 | Human Resources |
| 1143 | Bow | 2 | 5000 | Sales |
| 1165 | Jennings | 2 | 5000 | Sales |
| 1166 | Thompson | 10 | 10000 | Marketing |

* + - 1. Find employeenumber, Name & Max salary from above table

SELECT EmployeeNo, LastName, Max(AnnualSalary) AS MaxSalaary from EmpSalary Group by AnnualSalary,EmployeeNo, LastName

* + - 1. Find employeenumber, Name & min salary from above table

SELECT EmployeeNo, LastName, MIN(AnnualSalary) AS MaxSalaary from EmpSalary Group by AnnualSalary,EmployeeNo, LastName

* + - 1. Find employeenumber, Name & average salary from above table

SELECT EmployeeNo, LastName, AVG(AnnualSalary) AS MaxSalaary from EmpSalary Group by AnnualSalary,EmployeeNo, LastName

* + - 1. Find Count of employeenumber with respect to salary (Hint – Group by column salary)

SELECT count(EmployeeNo) EmployeeCount,AnnualSalary from EmpSalary group by AnnualSalary

* + - 1. Find average salary consumed by each department

select avg (AnnualSalary) as AvgSalaryByDept, Department from EmpSalary GROUP BY Department

* + - 1. Find sum of salary for each level in above table

SELECT SUM(AnnualSalary) AS SumOfSalary, Level FROM EmpSalary GROUP BY Level

* + - 1. Find count of employee whose average salary greater that 7000

SELECT Count(EmployeeNo) AS EmpCount, AVG(ANNUALSALARY) AvgSalary FROM EmpSalary Group BY AnnualSalary HAVING AVG(ANNUALSALARY)>7000

* + - 1. Find departments whose sum of salary greater that 10000

SELECT SUM(ANNUALSALARY) TotalSalary, Department FROM EmpSalary Group BY Department HAVING SUM(ANNUALSALARY)>10000

## SQL Server Date Functions :

|  |  |
| --- | --- |
| **Function** | **Description** |
| [CURRENT\_TIMESTAMP](https://www.w3schools.com/sql/func_sqlserver_current_timestamp.asp) | Returns the current date and time |
| [DATEADD](https://www.w3schools.com/sql/func_sqlserver_dateadd.asp) | Adds a time/date interval to a date and then returns the date |
| [DATEDIFF](https://www.w3schools.com/sql/func_sqlserver_datediff.asp) | Returns the difference between two dates |
| [DATEFROMPARTS](https://www.w3schools.com/sql/func_sqlserver_datefromparts.asp) | Returns a date from the specified parts (year, month, and day values) |
| [DATENAME](https://www.w3schools.com/sql/func_sqlserver_datename.asp) | Returns a specified part of a date (as string) |
| [DATEPART](https://www.w3schools.com/sql/func_sqlserver_datepart.asp) | Returns a specified part of a date (as integer) |
| [DAY](https://www.w3schools.com/sql/func_sqlserver_day.asp) | Returns the day of the month for a specified date |
| [GETDATE](https://www.w3schools.com/sql/func_sqlserver_getdate.asp) | Returns the current database system date and time |
| [GETUTCDATE](https://www.w3schools.com/sql/func_sqlserver_getutcdate.asp) | Returns the current database system UTC date and time |
| [ISDATE](https://www.w3schools.com/sql/func_sqlserver_isdate.asp) | Checks an expression and returns 1 if it is a valid date, otherwise 0 |
| [MONTH](https://www.w3schools.com/sql/func_sqlserver_month.asp) | Returns the month part for a specified date (a number from 1 to 12) |
| [SYSDATETIME](https://www.w3schools.com/sql/func_sqlserver_sysdatetime.asp) | Returns the date and time of the SQL Server |
| [YEAR](https://www.w3schools.com/sql/func_sqlserver_year.asp) | Returns the year part for a specified date |

**Below four Date functions used to find/fetch current date & time**

1. The **CURRENT\_TIMESTAMP** function returns the current date and time, in a 'YYYY-MM-DD hh:mm:ss.mmm' format.

SELECT CURRENT\_TIMESTAMP

1. The **GETDATE()** function returns the current database system date and time, in a 'YYYY-MM-DD hh:mm:ss.mmm' format.

SELECT GETDATE()

1. The **GETUTCDATE()** function returns the current database system UTC date and time, in a 'YYYY-MM-DD hh:mm:ss.mmm' format.

SELECT GETUTCDATE()

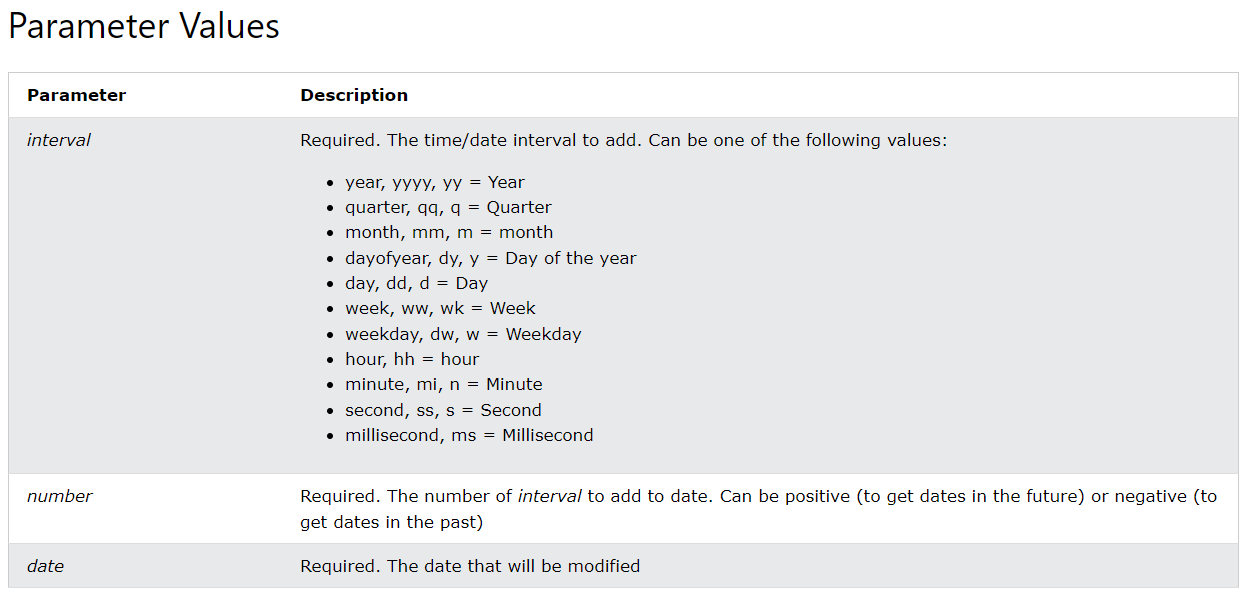
1. The **SYSDATETIME()** function returns the date and time of the computer where the SQL Server is running.

SELECT SYSDATETIME()

**Other Date functions in sql server :**

1. The **DATEADD()** function adds a time/date interval to a date and then returns the date.

Syntax : Select DATEADD(interval, number, date)



Example :

SELECT DATEADD(year, 1, '2017/08/25') AS YearAdd; ----It will add 1 year to date

SELECT DATEADD(month, 2, '2017/08/25') AS DateAdd; ----It will add 2 months to date

SELECT DATEADD(month, -2, '2017/08/25') AS DateAdd; ----It will add -2 months to date i.e. 2015

1. The **DATEDIFF()** function returns the difference between two dates.

Syntax : Select DATEDIFF(interval, date1, date2)

Example :

SELECT DATEDIFF(month, '2017/08/25', '2011/08/25') AS DateDiff; -- Displays diff between given dates in month

SELECT DATEDIFF(hour, '2017/08/25 07:00', '2017/08/25 12:45') AS DateDiff; --Diff in Hours

1. The DATEFROMPARTS() function returns a date from the specified parts (year, month, and day values).

Syntax : Select DATEFROMPARTS(year, month, day)

Example :

SELECT DATEFROMPARTS(2018, 10, 31) AS DateFromParts;

1. The DATENAME() function returns a specified part of a date.

This function returns the result as a string value.

Syntax : Select DATENAME(interval, date)

Example :

SELECT DATENAME(year, '2017/08/25') AS DatePartString;

SELECT DATENAME(yy, '2017/08/25') AS DatePartString;

SELECT DATENAME(month, '2017/08/25') AS DatePartString;

SELECT DATENAME(hour, '2017/08/25 08:36') AS DatePartString;

SELECT DATENAME(minute, '2017/08/25 08:36') AS DatePartString;

1. The DATEPART() function returns a specified part of a date.

This function returns the result as an integer value.

Syntax : Select DATEPART(interval, date)

Example :

SELECT DATEPART(yy, '2017/08/25') AS DatePartInt;

SELECT DATEPART(month, '2017/08/25') AS DatePartInt;

SELECT DATEPART(hour, '2017/08/25 08:36') AS DatePartInt;

SELECT DATEPART(minute, '2017/08/25 08:36') AS DatePartInt;

1. The ISDATE() function checks an expression and returns 1 if it is a valid date, otherwise 0.

Syntax : ISDATE(expression)

Example : SELECT ISDATE('2023-08-14')

SELECT ISDATE(‘Hello’)

1. Day, Month, Year : Syntax SELECT Day/Month/Year(Date)

The DAY() function returns the day of the month (from 1 to 31) for a specified date.

SELECT DAY('2017/08/25') AS DayOfMonth;

SELECT DAY('2017/08/13 09:08') AS DayOfMonth;

The MONTH() function returns the month part for a specified date (a number from 1 to 12).

SELECT MONTH('2017/08/25') AS Month;

SELECT MONTH('2017/05/25 09:08') AS Month;

The YEAR() function returns the year part for a specified date.

SELECT YEAR('2017/08/25') AS Year;

SELECT YEAR('1998/05/25 09:08') AS Year;

Exercise :

CREATE StudentData table with columns (Id,StName, DOB, City, JoiningDate )

Data type for DOB & JoiningDate should be Date

Insert below data in table

Id StName DOB City JoiningDate

1 Steward 2001-03-02 London 2022-05-02

2 Lucman 2000-08-19 New York 2022-05-02

3 Brathwaite 2001-01-07 Oslo 2022-05-03

4 Lourel 1999-11-05 Mumbai 2022-05-03

5 Joe Stephen 2003-08-25 China 2022-05-05

6 Jack 2000-08-19 New York 2022-05-02

7 Mike 2001-01-07 Oslo 2022-05-03

8 Paul Harry 1999-11-05 Mumbai 2022-05-03

9 Mike Hamper 1998-12-20 China 2022-05-05

10 Sara Jones 2000-08-12 Mumbai 2022-05-02

11 Johnson 1999-01-09 China 2022-05-02

12 David Stephen 2002-03-06 China 2022-05-03

13 Mark John 1997-10-12 New York 2022-05-02

14 Ken Villy 1997-11-30 New York 2022-05-02

15 Harry Teckor 2000-08-14 Oslo 2022-05-04

16 Karl Hoope 2001-03-19 China 2022-05-04

17 Jacky Laurel 1999-06-25 Mumbai 2022-05-06

1. Calculate Age of Student. Display name and age of student.

SELECT StName,DATEDIFF(YEAR,BirthDate,GETDATE()) AgeOfStudent FROM StudentData

1. Find number of students who were born before 2000.

SELECT COUNT(1) AS StudentCount FROM StudentData WHERE YEAR(BirthDate)>2000

1. Calculate average age of students from New York City

SELECT AVG(DATEDIFF(YEAR,BirthDate,GETDATE())) AS AverageAge FROM StudentData WHERE City='New York'

1. Find number of students joined for each day. Display count & day name.

SELECT COUNT(1) AS DayCount, DATENAME(WEEKDAY,JoiningDate) AS JoiningDay FROM StudentData GROUP BY DATENAME(WEEKDAY,JoiningDate)

1. Find average age of student for each city.

SELECT AVG(DATEDIFF(YEAR,BirthDate,GETDATE())) AverageAgeByCity, City FROM StudentData GROUP BY City

1. Find number of students from each city who were born after 2000.
2. SELECT COUNT(1) AS StudentCount FROM StudentData WHERE YEAR(BirthDate)<2000
3. Find count of Students from each city

SELECT Count(1) AS StudentCount , City FROM StudentData GROUP BY City

Also write SQL queries for below questions :

* 1. Display all the records by adding 25 years to its birthdate
  2. Display all the records who are born in January irrespective of birth year
  3. Display count of students & year born in particular year
  4. Display count & average age of students having average age more than 22 years.
  5. Display Count & joining date of all the students where count for a day is more than 2.

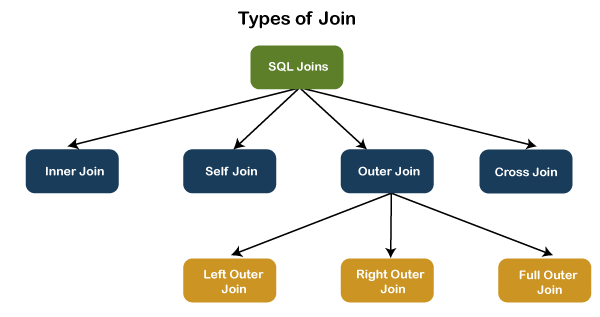
**SQL Server Joins:**

Summary : In a relational database, data is distributed in multiple logical tables. To get a complete meaningful set of data, you need to query data from these tables using joins. SQL Server supports many kinds of joins, including [inner join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-inner-join/), [left join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-left-join/), [right join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-right-join/), [full outer join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-full-outer-join/), self join and [cross join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-cross-join/). Each join type specifies how SQL Server uses data from one table to select rows in another table.

SQL Server (Transact-SQL) JOINS are used to retrieve data from multiple tables. A SQL Server JOIN is performed whenever two or more tables are joined in a SQL statement.

There are 4 different types of SQL Server joins:

* SQL Server INNER JOIN (or sometimes called simple join)
* SQL Server LEFT OUTER JOIN (or sometimes called LEFT JOIN)
* SQL Server RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
* SQL Server FULL OUTER JOIN (or sometimes called FULL JOIN)



**INNER JOIN (simple join)**

It is the most common type of join. SQL Server INNER JOINS return all rows from multiple tables where the join condition is met.

**Syntax**

The syntax for the INNER JOIN in SQL Server (Transact-SQL) is:

SELECT column\_list

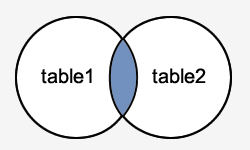
From Table1

INNER Join Table2

ON Table1.column = Table2.column

**Visual Illustration**

In this visual diagram, the SQL Server INNER JOIN returns the shaded area:



The SQL Server INNER JOIN would return the records where table1 and table2 intersect.

Let us first create two tables "**StudentDetails**" and "**FeeDetails**” & insert some sample records in those tables :

Table StudentDetails

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SdId** | **Admission\_No** | **First\_Name** | **Last\_Name** | **Age** | **City** |
| 1 | 3354 | Luisa | Evans | 13 | Texas |
| 2 | 2135 | Paul | Ward | 15 | Alaska |
| 3 | 4321 | Peter | Bennett | 14 | California |
| 4 | 4213 | Carlos | Patterson | 17 | New York |
| 5 | 5112 | Rose | Huges | 16 | Florida |
| 6 | 6113 | Marielia | Simmons | 15 | Arizona |
| 7 | 7555 | Antonio | Butler | 14 | New York |
| 8 | 8345 | Diego | Cox | 13 | California |

Table FeeDetails

|  |  |  |  |
| --- | --- | --- | --- |
| FdId | Admission\_no | Course | Amount\_Paid |
| 1 | 3354 | Java | 20000 |
| 2 | 7555 | Android | 22000 |
| 3 | 4321 | Python | 18000 |
| 4 | 8345 | SQL | 15000 |
| 5 | 5112 | Machine Learning | 30000 |

Now suppose we have to fetch result as all the columns from table StudentDetails who paid their fees & their fees paid amount from table FeeDetails :

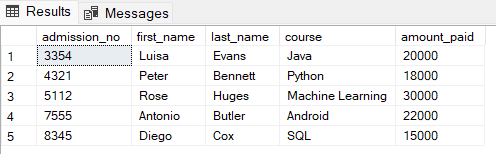
We can achieve this by joining two tables with inner join as below :

**SELECT** StudentDetails.admission\_no, StudentDetails.first\_name, StudentDetails.last\_name, FeeDetails.course, FeeDetails.amount\_paid  **FROM** StudentDetails

**INNER** JOIN FeeDetails

**ON** StudentDetails.admission\_no = FeeDetails.admission\_no;

Our output data will be as below,



Here have used the **admission\_no column** as a join condition to get the data from both tables. Depending on this table, we can see the information of the students who have paid their fee.

A join that displays only the rows that have a match in both joined tables. **Columns containing NULL do not match any values when you are creating an inner join and are therefore excluded from the result set**. Null values do not match other null values.

# **OUTER JOIN**

In the SQL outer JOIN all the content of the both tables are integrated together either they are matched or not.

**Outer join of two types:**

**1.Left outer join** (also known as left join): this join returns all the rows from left table combine with the matching rows of the right table. If you get no matching in the right table it returns NULL values.

**2.Right outer join** (also known as right join): this join returns all the rows from right table are combined with the matching rows of left table .If you get no column matching in the left table .it returns null value.

This diagram shows the different type of joins:

1. **Left Outer Join:**

The SQL left join returns all the values from the left table and it also includes matching values from right table, if there are no matching join value it returns NULL.

**Basic Syntax of Left Join:**

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

FROM table1

LEFTJOIN table2

ON table1.column\_field = table2.column\_field;

Lets consider above studentdetails & feedetails table:

SELECT StudentDetails.admission\_no, StudentDetails.first\_name, StudentDetails.last\_name, FeeDetails.course, FeeDetails.amount\_paid

FROM StudentDetails

left JOIN FeeDetails

ON StudentDetails.admission\_no = FeeDetails.admission\_no;

**Output of this query will contain matching records from both tables & non matching records from left table.**

1. **Right Outer Join:**

The SQL right join returns all the values from the rows of right table. It also includes the matched values from left table but if there is no matching in both tables, it returns NULL.

**Basic syntax for right join:**

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

**FROM** table1

RIGHT JOIN table2

**ON** table1.column\_field = table2.column\_field;

Lets consider above studentdetails & feedetails table:

SELECT StudentDetails.admission\_no, StudentDetails.first\_name, StudentDetails.last\_name, FeeDetails.course, FeeDetails.amount\_paid

FROM StudentDetails

RIGHT JOIN FeeDetails

ON StudentDetails.admission\_no = FeeDetails.admission\_no;

1. **Full Outer Join**

The SQL full join is the result of combination of both left and right outer join and the join tables have all the records from both tables. It puts NULL on the place of matches not found.

SQL full outer join and SQL join are same. generally it is known as SQL FULL JOIN.

SQL full outer join is used to combine the result of both left and right outer join and returns all rows (don't care its matched or unmatched) from the both participating tables.

**Syntax for full outer join:**

**SELECT** \*

**FROM** table1

**FULL** OUTER JOIN table2

**ON** table1.column\_name = table2.column\_name;

Let us take two tables to demonstrate full outer join:

**SELF Join**

A self join allows you to join a table to itself. It helps query hierarchical data or compare rows within the same table.

A self join uses the [inner join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-inner-join/) or [left join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-left-join/) clause. Because the query that uses the self join references the same table, the [table alias](https://www.sqlservertutorial.net/sql-server-basics/sql-server-alias/) is used to assign different names to the same table within the query.

Note that referencing the same table more than one in a query without using table aliases will result in an error.

The following shows the syntax of joining the table T to itself:

SELECT select\_list FROM T t1

[INNER | LEFT] JOIN T t2

ON join\_predicate;

The query references the table T twice. The table aliases t1 and t2 are used to assign the T table different names in the query.

Lets create a sample table to work with Self Join & insert some records:

CREATE TABLE [dbo].[tblEmployeeDetails](

[Emp\_id] [bigint] IDENTITY(1,1) PRIMARY KEY,

[Emp\_name] [nvarchar](200) NULL,

[Emp\_mgr\_id] [bigint] NULL )

SELECT \* FROM tblEmployeeDetails

Insert into tblEmployeeDetails (Emp\_name,Emp\_mgr\_id) values

('Rakesh', NULL)

,('Namam', 1)

,('Sanket', 2)

,('Vishal', 3)

,('Ram', 1)

,('Karan', 2)

,('Suhas', 3)

Now suppose in this table each employee is manager to some other person and same is mentioned using Emp\_mgr\_id column in table.

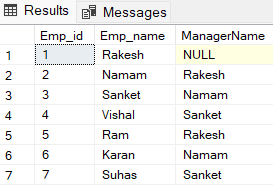
Then how to find which employee belongs to which manager??

Here we can use self-join as below query using allies names to same table & column

SELECT E.Emp\_id, E.Emp\_name, M.EMP\_NAME AS 'ManagerName' FROM tblEmployeeDetails E

LEFT JOIN tblEmployeeDetails M ON E.Emp\_mgr\_id=M.Emp\_id

Result will be:



If we use Inner join or Join in above query to join this two tables. Null values will be ignored.

**Cross Join:**

Join operation in SQL is used to combine multiple tables together into a single table.

If we use the cross join to combine two different tables, then we will get the Cartesian product of the sets of rows from the joined table. When each row of the first table is combined with each row from the second table, it is known as Cartesian join or cross join.

After performing the cross join operation, the total number of rows present in the final table will be equal to the product of the number of rows present in table 1 and the number of rows present in table 2.

**For example:**  
If there are two records in table 1 and three records in table 2, then after performing cross join operation, we will get six records in the final table.

Let us take a look at the syntax of writing a query to perform the cross join operation in SQL.

Syntax:

Select Column\_list From Table1 Cross Join Table2 (Full Join will not have ON condition)

**Sub Queries & Co-related Sub Queries in SQL:**

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

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow −

* Subqueries must be enclosed within parentheses.
* A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
* An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
* Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
* The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

## Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

WHERE column\_name OPERATOR

(SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

[WHERE])

**Students**

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_Id** | **S\_Name** | **S\_Address** | **S\_Phone** |
| S1 | RAM | Manali | 9455123451 |
| S2 | RAMESH | Delhi | 9652431543 |
| S3 | SUJIT | Rohtak | 9652431543 |
| S4 | AJIT | Jalandhar | 9156768971 |
|  |  |  |  |

**Courses**

|  |  |
| --- | --- |
| C\_Id | C\_Name |
| C1 | DSA |
| C2 | Programming |
| C3 | DBMS |

**Student\_Course**

|  |  |
| --- | --- |
| S\_Id | C\_Id |
| S1 | C1 |
| S1 | C3 |
| S2 | C1 |
| S3 | C2 |
| S4 | C2 |
| S4 | C3 |

**There are mainly two types of nested queries:**

* + **Independent Nested Queries:** In independent nested queries, query execution starts from innermost query to outermost queries. The execution of the inner query is independent of the outer query, but the result of the inner query is used in the execution of the outer query. Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.

IN: If we want to find out S\_ID who are enrolled in C\_NAME ‘DSA’ or ‘DBMS’, we can write it with the help of independent nested query and IN operator. From COURSE table, we can find out C\_ID for C\_NAME ‘DSA’ or DBMS’ and we can use these C\_IDs for finding S\_IDs from STUDENT\_COURSE TABLE.

STEP 1: Finding C\_ID for C\_NAME =’DSA’ or ‘DBMS’ Select C\_ID from COURSE where C\_NAME = ‘DSA’ or C\_NAME = ‘DBMS’

STEP 2: Using C\_ID of step 1 for finding S\_ID Select S\_ID from STUDENT\_COURSE where C\_ID IN (SELECT C\_ID from COURSE where C\_NAME = ‘DSA’ or C\_NAME=’DBMS’);

The inner query will return a set with members C1 and C3 and the outer query will return those S\_IDs for which C\_ID is equal to any member of the set (C1 and C3 in this case). So, it will return S1, S2 and S4.

Note: If we want to find out names of STUDENTs who have either enrolled in ‘DSA’ or ‘DBMS’, it can be done as:

Select S\_NAME from STUDENT where S\_ID IN (Select S\_ID from STUDENT\_COURSE where C\_ID IN (SELECT C\_ID from COURSE where C\_NAME=’DSA’ or C\_NAME=’DBMS’));

NOT IN: If we want to find out S\_IDs of STUDENTs who have neither enrolled in ‘DSA’ nor in ‘DBMS’, it can be done as:

Select S\_ID from STUDENT where S\_ID NOT IN (Select S\_ID from STUDENT\_COURSE where C\_ID IN (SELECT C\_ID from COURSE where C\_NAME=’DSA’ or C\_NAME=’DBMS’));

The innermost query will return a set with members C1 and C3. Second inner query will return those S\_IDs for which C\_ID is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those S\_IDs where S\_ID is not a member of set (S1, S2 and S4). So it will return S3.

* **Co-related Nested Queries:** In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query. e.g.; If we want to find out S\_NAME of STUDENTs who are enrolled in C\_ID ‘C1’, it can be done with the help of co-related nested query as:

Select S\_NAME from STUDENT S where EXISTS ( select \* from STUDENT\_COURSE SC where S.S\_ID=SC.S\_ID and SC.C\_ID=’C1’);

For each row of STUDENT S, it will find the rows from STUDENT\_COURSE where S.S\_ID = SC.S\_ID and SC.C\_ID=’C1’. If for a S\_ID from STUDENT S, atleast a row exists in STUDENT\_COURSE SC with C\_ID=’C1’, then inner query will return true and corresponding S\_ID will be returned as output.

**SQL Set Operators:**

A set operator in SQL is a keyword that lets you combine the results of two queries into a single query.

Sometimes when working with SQL, you’ll have a need to query data from two more tables. But instead of joining these two tables, you’ll need to list the results from both tables in a single result, or in different rows. That’s what set operators do.

Types Of Set Operators:

There are a few different set operators that can be used, depending on your needs, and which database vendor you’re using.

The different set operators are:

* UNION
* UNION ALL
* MINUS/ EXCEPT
* INTERSECT

Let’s take a look at each of these, using some sample data.

Standard Syntax for SET operators:

SELECT your\_select\_query

set\_operator

SELECT another\_select\_query;

It uses two (or more) SELECT queries, with a set operator in the middle.

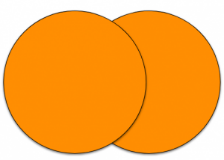
There are a few things to keep in mind though.

* When selecting your columns, the **number of columns needs to match** between queries, and the **data type of each column needs to be compatible**.
* So, if you select three columns in the first query, you need to select three columns in the second query. The data types also need to be compatible, so if you select a number and two-character types in the first query, you need to do the same in the second query.
* Also, if you want to order your results, the ORDER BY must go at the end of the last query. You can’t add ORDER BY inside each SELECT query before the set operator.

1. **Union (Combining Results)**

The UNION keyword or set operator will allow you to combine the results of two queries. It removes any duplicate results and shows you the combination of both.

Expressed as a Venn diagram, where each circle represents a query result, it looks like this:



Lets see Customer & Emp tables as below :

SELECT first\_name, last\_name

FROM customer

UNION

SELECT first\_name, last\_name

FROM employee;

We can use Where clause to filter data in all SET operators.

Union with where & order by:

SELECT ‘Customer’ AS record\_type, first\_name, last\_name

FROM customer

WHERE status = ‘Active’

UNION

SELECT ‘Employee’, first\_name, last\_name

FROM employee

WHERE emp\_status = ‘Current’

ORDER BY record\_type, first\_name, last\_name;

### **What’s the Difference Between a UNION and a JOIN?**

The UNION and JOIN keywords both combine results from two different tables or queries.

The difference is how they are combined.

* UNION combines data into separate rows, and JOIN combines data into separate columns.
* When performing a JOIN, there is a column that matches between the two tables, and additional data may be displayed.

1. **Union All:**

The UNION ALL set operator also combines the results from two queries.

It’s very similar to UNION, but it does not remove duplicates.

SELECT first\_name, last\_name

FROM customer

UNION ALL

SELECT first\_name, last\_name

FROM employee;

There are some records in this table that appear twice (Stephen Jones and Paula Johnson). Why is that?

This is because **UNION ALL does not remove duplicates**. So the same first name and last name values exist in both the customer and employee tables, and this query shows each of them.

### **What’s the Difference between UNION and UNION ALL?**

The major difference between UNION ALL and UNION in SQL is that UNION removes any duplicate results from the final result set, and UNION ALL does not. UNION performs a [DISTINCT](https://www.databasestar.com/sql-distinct/) on the result set to remove duplicates.

To remember this, consider that “ALL” in UNION ALL means “show all records”.

Therefore, UNION ALL will almost always show more results, as it does not remove duplicate records.

As a result of this, UNION is often slower than UNION ALL, because there is an operation to remove duplicate values (a.k.a DISTINCT), which is often a costly step in a query.

UNION ALL does not perform a distinct, so is usually faster.

So, if you don’t need to have unique rows in your result set, or if you’re sure the rows in the database or query are unique already, then use UNION ALL.

1. **Except**

Another set operator we can use is the MINUS keyword.

The MINUS set operator will return results that are found in the first query specified that don’t exist in the second query.

Using our example data, we could use the MINUS set operator to find all names in the customer table that don’t exist in the employee table.

Our query would look like this:

SELECT first\_name, last\_name

FROM customer

EXCEPT

SELECT first\_name, last\_name

FROM employee;

If a result exists in the employee table as well as the customer table, it is not shown. Only the results from the customer table that are not in the employee table are shown.

1. **Intersect**

The INTERSECT keyword allows you to find results that exist in both queries. Two SELECT statements are needed, and any results that are found in both of them are returned if INTERSECT is used.

Using our example data, we could use the INTERSECT set operator to find all names in the customer table that don’t exist in the employee table.

Our query would look like this:

SELECT first\_name, last\_name

FROM customer

INTERSECT

SELECT first\_name, last\_name

FROM employee;

**The SQL CASE Statement:**

The CASE statement goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

## CASE Syntax

CASE  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
    WHEN conditionN THEN resultN  
    ELSE result  
END;

Below is a selection from the "OrderDetails" table in the Northwind sample database:

SELECT Order\_No, Quantity,PRODUCT\_ID,

CASE

WHEN Quantity > 30 THEN 'The quantity is greater than 30'

WHEN Quantity = 30 THEN 'The quantity is 30'

ELSE 'The quantity is under 30'

END AS QuantityText

FROM Orders;

SELECT Order\_No, Quantity,PRODUCT\_ID,

FROM Orders;

ORDER BY

(CASE WHEN PRODUCT\_ID IS NULL THEN Quantity

ELSE PRODUCT\_ID END)

**Different ways to Replace NULL value:**

Null functions are required **to perform operations on the null values stored in our database**. We can perform functions on NULL values, which explicitly recognize if a value is null or not. Helps us to replace NULL values with the desired value.

Suppose in Product\_Master we need to calculate total cost of each product.

Our query will be like below:

SELECT product, Quantity \* (Rate + GST) AS 'TotalCost'

FROM Product\_Master;

If any value in GST column contains null value. Total cost will return as null. Which results in inaccurate data.

To avoid this, we need to replace null with sum other value:

In MS SQL we have ISNULL & Coalesce functions to replace null values.

The SQL Server [ISNULL()](https://www.w3schools.com/sql/func_sqlserver_isnull.asp) function lets you return an alternative value when an expression is NULL:

It contains two parameters. The first parameter is to evaluate the expression for NULL. If the first parameter is NULL, the function replaces the second parameter.

SELECT product, Quantity \* (Rate + ISNULL(GST,0)) AS 'TotalCost'

FROM Product\_Master;

or we can use the [COALESCE()](https://www.w3schools.com/sql/func_sqlserver_coalesce.asp) function, like this:

COALESCE evaluates the expression in order and returns the first non-null value.

COALESCE ( expression1, expression2 [, … expressionN] )

It can contain N no of parameters. It returns the first non-null parameter value.

SELECT product, Quantity \* (Rate + COALESCE(GST,NULL,NULL,0)) AS 'TotalCost'

FROM Product\_Master;

**SQL Server CAST() Function**

The CAST() function converts a value (of any DATA type) into a specified datatype.

Syntax:

CAST(*expression* AS *datatype(length)*)

SELECT CAST('2017-08-25' AS datetime);

SELECT CAST(25.65 AS int);

**SQL Server**CONVERT()**Function**

The CONVERT() function converts a value (of any type) into a specified datatype.

Syntax:

CONVERT(*data\_type(length)*,*expression*,*style*)

**The SQL SELECT INTO Statement**

The SELECT INTO statement copies data from one table into a new table.

**Copy all columns into a new table:**

SELECT \*  
INTO newtable [IN externaldb]  
FROM oldtable  
WHERE condition;

**Copy only some columns into a new table:**

SELECT column1, column2, column3, ...  
INTO newtable [IN externaldb]  
FROM oldtable  
WHERE condition;

The new table will be created with the column-names and types as defined in the old table. You can create new column names using the AS clause.

SQL SELECT INTO Examples

**The following SQL statement creates a backup copy of Customers:**

SELECT \* INTO CustomersBackup2017  
FROM Customers;

**The following SQL statement uses the IN clause to copy the table into a new table in another database**:

SELECT \* INTO CustomersBackup2017 IN 'Backup.mdb'  
FROM Customers;

**The following SQL statement copies only a few columns into a new table:**

SELECT CustomerName, ContactName INTO CustomersBackup2017  
FROM Customers;

**The following SQL statement copies only the German customers into a new table:**

SELECT \* INTO CustomersGermany  
FROM Customers  
WHERE Country = 'Germany';

**The following SQL statement copies data from more than one table into a new table:**

SELECT Customers.CustomerName, Orders.OrderID  
INTO CustomersOrderBackup2017  
FROM Customers  
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

**Tip:** SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

SELECT \* INTO *newtable*  
FROM *oldtable*  
WHERE 1 = 0;

**The SQL INSERT INTO SELECT Statement**

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

The INSERT INTO SELECT statement requires that the data types in source and target tables match.

**Note:** The existing records in the target table are unaffected.

INSERT INTO SELECT Syntax

Copy all columns from one table to another table:

INSERT INTO table2  
SELECT \* FROM table1WHERE condition;

Copy only some columns from one table into another table:

INSERT INTO table2 (column1, column2, column3, ...)  
SELECT column1, column2, column3, ...  
FROM table1  
WHERE condition;

The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

**Example**

INSERT INTO Customers (CustomerName, City, Country)  
SELECT SupplierName, City, Country FROM Suppliers;

The following SQL statement copies "Suppliers" into "Customers" (fill all columns):

**Example**

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)  
SELECT SupplierName, ContactName, Address, City, PostalCode, Country FROM Suppliers;

The following SQL statement copies only the German suppliers into "Customers":

**Example**

INSERT INTO Customers (CustomerName, City, Country)  
SELECT SupplierName, City, Country FROM Suppliers  
WHERE Country='Germany';

**INDEX in SQL Server:**

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

CREATE INDEX index\_name  
**ON** table\_name (column1, column2, ...);

#### **What is a SQL Server Clustered Index?**

A clustered index is one of the main index types in SQL Server.  A clustered index stores the index key in a B-tree structure along with the actual table data in each leaf node of the index.  Having a clustered index defined on a table eliminates the heap table structure we described in the previous section.  Since the rest of the table data (eg. non-key columns) is stored in the leaf nodes of the index structure, a table can only have one clustered index defined on it.

#### **SQL Server Clustered Index Benefits and Usage**

The are many benefits to having a clustered index defined on a table but the main benefit is speeding up query performance.  Queries that contain the index key columns in the WHERE clause use the index structure to go straight to the table data.  A clustered index also removes the need for an extra lookup, to get the rest of the column data, when querying based on the index key values.  This is something that is not true of other index types.  You can also eliminate the need to sort data.  If the ORDER BY clause of a query is based on the index key values then a sort is not required since the data is already ordered by these values.

#### **SQL Server Clustered Index Disadvantages**

There are a couple of disadvantages when it comes to clustered indexes.  There is some overhead in maintaining the index structure with respect to any DML operation (INSERT, UPDATE, DELETE).  This is especially true if you are updating the actual key values in the index as in this case all of the associated table data also has to be moved as it is stored in the leaf node of the index entry.  In each case there will be some performance impact to your DML query.

#### **SQL Server Clustered Index Basic Syntax**

CREATE non CLUSTERED INDEX CIX\_TestData\_TestId ON dbo.TestData (TestId);

ALTER INDEX IX\_TestData\_TestId ON TestData REBUILD WITH (ONLINE = ON);

DROP INDEX IX\_TestData\_TestId on TestData WITH (ONLINE = ON);

## SQL Server Non-Clustered Index

#### **What is a SQL Server Non-Clustered Index?**

A non-clustered index is the other main type of index used in SQL Server.  Similar to its clustered index counterpart, the index key columns are stored in a B-tree structure except in this case the actual data is not stored in the leaf nodes.  In this type of index, a pointer to the actual table data is stored in the leaf node.  This could point to the data value in the clustered index or in a heap structure depending on how the table data is stored.

#### **SQL Server Non-Clustered Index Benefits and Usage**

The benefits of a non-clustered index are similar to that of the clustered index we mentioned above, the main benefit being speeding up query performance.  There are however two differences.  The first is that you can have multiple non-clustered indexes defined on a single table.  This allows you to index different columns which can help queries with different columns in the WHERE clause allowing you to fetch data faster and in the ORDER BY clause to eliminate a need for a sort.  The second is that although there is overhead for a non-clustered index when it comes to DML operations there is less than its clustered counterpart.

#### **SQL Server Non-Clustered Index Disadvantages**

Similar to the clustered index the main disadvantage of a non-clustered index is the extra overhead required in maintaining the index during DML operations.  It can sometimes be tricky to balance query performance as having too many non-clustered indexes on a table, while they will help all of your SELECT queries, can sometimes really slow down DML performance.

#### **SQL Server Non-Clustered Index Basic Syntax**

CREATE INDEX IX\_TestData\_TestDate ON dbo.TestData (TestDate);

ALTER INDEX IX\_TestData\_TestDate ON TestData REBUILD WITH (ONLINE = ON);

DROP INDEX IX\_TestData\_TestDate on TestData;

TIP: <https://dotnettutorials.net/lesson/sql-server-indexes/>

**T-SQL:**

T-SQL, which stands for Transact-SQL and is sometimes referred to as TSQL, is an extension of the SQL language used primarily within Microsoft SQL Server. This means that it provides all the functionality of SQL but with some added extras.

**What is the difference between SQL and T-SQL?**

Now we have covered the basics of both, let’s take a look at the main differences:

**Difference #1**

The obvious difference is in what they are designed for: SQL is a​ query language used for manipulating data stored in a database. T-SQL is also a query language, but it’s an extension of SQL that is primarily used in Microsoft SQL Server databases and software.

**Difference #2**

SQL is open-source​. T-SQL is developed and owned by Microsoft.

**Difference #3**

SQL statements are executed one at a time, also known as “non-procedural.” T-SQL executes statements in a “procedural” way, meaning that the code will be processed as a block, logically and in a structured order.

There are advantages and disadvantages to each approach, but from a learner perspective, this difference isn’t too important. You’ll be able to get and work with the data you want in either language, ​it’s just that the way you go about doing that will vary a bit depending on which language you’re using and the specifics of your query.

**Difference #4**

On top of these more general differences, SQL and T-SQL also have some slightly different command key words. T-SQL also features functions that are not part of regular SQL.

An example of this is how in ​we ​select the top ​X number of rows.​ In standard SQL, ​we ​would use the LIMIT keyword​. In T-SQL, we use the TOP keyword.

**Difference #5**

Finally, and as referenced before, T-SQL offers functionality that does not appear in regular SQL. One example is the ISNULL function. This will replace NULL values coming from a specific column. The below would return an age of “0” for any rows that have a value of NULL in the age column.

**Variables in SQL Server:**

**What is a variable**

A variable is an object that holds a single value of a specific type e.g., [integer](https://www.sqlservertutorial.net/sql-server-basics/sql-server-int/), [date](https://www.sqlservertutorial.net/sql-server-basics/sql-server-date/), or [varying character string](https://www.sqlservertutorial.net/sql-server-basics/sql-server-varchar/).

**We typically use variables in the following cases:**

* As a loop counter to count the number of times a loop is performed.
* To hold a value to be tested by a control-of-flow statement such as WHILE.
* To store the value returned by a [stored procedure](https://www.sqlservertutorial.net/sql-server-stored-procedures/) or a function

**Declaring a variable**

To declare a variable, you use the DECLARE statement. For example, the following statement declares a variable named @model\_year:

DECLARE @model\_year AS SMALLINT;

We can declare as many variables inside the script:

DECLARE @model\_year SMALLINT,

@product\_name VARCHAR(MAX);

**Assigning a value to a variable**

To assign a value to a variable, you use the SET statement. For example, the following statement assigns 2018 to the @model\_year variable:

SET @model\_year = 2018;

**Using variables in a query**

The following [SELECT](https://www.sqlservertutorial.net/sql-server-basics/sql-server-select/) statement uses the @PaymentAmount variable in the [WHERE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-where/) clause to find the customers who pays less than 5000 payment amount.

DECLARE @PaymentAmount SMALLINT;

SET @PaymentAmount = 5000;

SELECT

cust\_name,

AGENT\_CODE,

OPENING\_AMT,

PAYMENT\_AMT

FROM

CUSTOMER

WHERE

PAYMENT\_AMT < @PaymentAmount

ORDER BY

AGENT\_CODE;

**Temporary Tables in SQL Server:**

Temporary tables are tables that exist temporarily on the SQL Server.

The temporary tables are useful for storing the immediate result sets that are accessed multiple times.

Temporary tables are very similar to normal tables in SQL Server with some limitations:

The name of the temporary table starts with a hash symbol (#). (Example #Employee)

These temporary tables are get stored in TempDb system database.

**Creating temporary tables:**

SQL Server provided two ways to create temporary tables via SELECT INTO and [CREATE TABLE](https://www.sqlservertutorial.net/sql-server-basics/sql-server-create-table/) statements.

Create temporary tables using **SELECT INTO** statement

The first way to create a temporary table is to use the SELECT INTO statement as shown below:

SELECT

select\_list

INTO

#temporary\_table

FROM

table\_name

Where Condition

Create temporary tables using **CREATE TABLE** statement

CREATE TABLE #haro\_products (

product\_name VARCHAR(MAX),

list\_price DEC(10,2)

);

This statement has the same syntax as creating a regular table. However, the name of the temporary table starts with a hash symbol (#)

After creating the temporary table, you can [insert data into this table](https://www.sqlservertutorial.net/sql-server-basics/sql-server-insert/) as a regular table:

Example:

SELECT AGENT\_CODE, AGENT\_NAME, WORKING\_AREA INTO #AgentData FROM AGENTS

SELECT \* FROM #AgentData

**Temporary Tables** are most likely as Permanent Tables. Temporary Tables are Created in TempDB and are automatically deleted as soon as the last connection is terminated. Temporary Tables helps us to store and process intermediate results. Temporary tables are very useful when we need to store temporary data. The Syntax to create a Temporary Table is given below:

**To Create Temporary Table:**

CREATE TABLE #EmpDetails (id INT, name VARCHAR(25))

**To Insert Values Into Temporary Table:**

INSERT INTO #EmpDetails VALUES (01, 'Lalit'), (02, 'Atharva')

**To Select Values from Temporary Table:**

SELECT \* FROM #EmpDetails

There are 2 types of Temporary Tables: Local Temporary Table, and Global Temporary Table. These are explained as following below.

**Local Temporary Table:**  
A Local Temp Table is available only for the session that has created it. It is automatically dropped (deleted) when the connection that has created it, is closed. To create Local Temporary Table Single “#” is used as the prefix of a table name.

Also, the user can drop this temporary table by using the “DROP TABLE #EmpDetails” query.

If the Temporary Table is created inside the stored procedure, it get dropped automatically upon the completion of stored procedure execution.

**Example:**

CREATE TABLE #EmpDetails

INSERT INTO #EmpDetails VALUES ( 01, 'Lalit'), ( 02, 'Atharva')

SELECT \* FROM #EmpDetails

**Global Temporary Table:**  
To create a Global Temporary Table, add the “##” symbol before the table name.

Global Temporary Tables are visible to all connections and dropped when the connection who created the table is closed.

**Example:**

CREATE TABLE ##EmpDetails (id INT, name VARCHAR(25))

SELECT \* FROM ##EmpDetails

**Table Variables in SQL Server:**

The table variable is a special type of the local variable that helps to store data temporarily, similar to the temp table in SQL Server. In fact, the table variable provides all the properties of the local variable, but the local variables have some limitations, unlike temp or regular tables.

The following syntax describes how to declare a table variable:

|  |  |
| --- | --- |
|  | DECLARE @LOCAL\_TABLEVARIABLE AS TABLE  (column\_1 DATATYPE,  column\_2 DATATYPE,  column\_N DATATYPE) |

If we want to declare a table variable, we have to start the DECLARE statement which is similar to local variables. The name of the local variable must start with at(@) sign. The TABLE keyword specifies that this variable is a table variable. After the TABLE keyword, we have to define column names and datatypes of the table variable in SQL Server.

In the following example, we will declare a table variable and insert the days of the week and their abbreviations to the table variable:

DECLARE @ListOWeekDays TABLE(DyNumber INT,DayAbb VARCHAR(40) , WeekName VARCHAR(40))

INSERT INTO @ListOWeekDays

VALUES

(1,'Mon','Monday')  ,

(2,'Tue','Tuesday') ,

(3,'Wed','Wednesday') ,

(4,'Thu','Thursday'),

(5,'Fri','Friday'),

(6,'Sat','Saturday'),

(7,'Sun','Sunday')

SELECT \* FROM @ListOWeekDays

What is the storage location of the table variables?

The lifecycle of the table variables starts in the declaration point and ends at the end of the batch. As a result, the table variable in SQL Server is automatically dropped at the end of the batch.

* You can update, delete data into table variable.
* Once declared, structure of table can’t change using alter command.
* Truncate will not work on table variables.
* Can not insert data into table variables using Inert into select command.

**Differences**:

|  |  |
| --- | --- |
| **Temp Table (#)** | **Table Variable (@)** |
| The temp table can be part of Transaction. | The table variable does not have any effect on the Transaction. |
| Temp table can have a clustered or non clustered index. | Table variable can have only clustered index. |
| It allows you **SELECT INTO** statements to insert data from the existing table. | It does not allow you **SELECT INTO** statements to insert data from the existing table. |
| Temp table can be accessed in nested stored procedure. For example you are working on stored procedure A, from stored proc A you are calling stored proc B. The temp table created in stored proc A can be accessed in stored proc B. | Table variable cannot be accessed in the nested stored procedure. |
| Temp table allows **TRUNCATE** or **ALTER** table statements. | Table variable does not allow **TRUNCATE** or **ALTER** table statements. |
| Temp table cannot be used in Used defined functions. | [Table variable can be used in User defined functions.](https://geeksarray.com/blog/user-defined-functions-in-sql-server) |

# **IF-Else statement in SQL Server:**

The IF...ELSE statement is a control-flow statement that allows you to execute or skip a [statement block](https://www.sqlservertutorial.net/sql-server-stored-procedures/sql-server-begin-end/) based on a specified condition.

**The IF statement**

The following illustrates the syntax of the IF statement:

IF boolean\_expression

BEGIN

statement\_block

END

In this syntax, if the Boolean\_expression evaluates to TRUE then the statement\_block in the [BEGIN...END](https://www.sqlservertutorial.net/sql-server-stored-procedures/sql-server-begin-end/) block is executed. Otherwise, the statement\_block is skipped and the control of the program is passed to the statement after the END keyword.

Note that if the Boolean expression contains a SELECT statement, you must enclose the SELECT statement in parentheses.

The following example first gets the sales amount from the AgentOrders table and then prints out a message if the sales amount is greater than 1 million

BEGIN

DECLARE @sales INT;

SELECT

@sales = SUM(ORD\_AMOUNT + ADVANCE\_AMOUNT)

FROM

AgentORDERS

WHERE

month(ORD\_DATE) = 07

--SELECT @sales;

IF @sales > 1000

BEGIN

PRINT 'Great! The sales amount in 2018 is greater than 1,000,000';

END

END

**The IF ELSE statement**

When the condition in the IF clause evaluates to FALSE and you want to execute another statement block, you can use the ELSE clause.

The following illustrates the IF ELSE statement:

IF Boolean\_expression

BEGIN

-- Statement block executes when the Boolean expression is TRUE

END

ELSE

BEGIN

-- Statement block executes when the Boolean expression is FALSE

END

Each IF statement has a condition. If the condition evaluates to TRUE then the statement block in the IF clause is executed. If the condition is FALSE, then the code block in the ELSE clause is executed.

See the following example:

BEGIN

DECLARE @sales INT;

SELECT @sales = SUM(ORD\_AMOUNT + ADVANCE\_AMOUNT) FROM AgentORDERS

WHERE month(ORD\_DATE) = 07

--SELECT @sales;

IF @sales > 100000

BEGIN

PRINT 'Great! The sales amount in 2018 is greater than 100000';

END

Else

BEGIN

PRINT 'Sales amount in July did not reach 100000';

END

END

# **Stored Procedures in SQL Server**

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

You can find the stored procedure in the Object Explorer, under **Programmability > Stored Procedures**

What are the Benefits of using a Stored Procedure in SQL?

Stored procedures provide some crucial benefits, which are:

* Reusable: As mentioned, multiple users and applications can easily use and reuse stored procedures by merely calling it.
* Easy to modify: You can quickly change the statements in a stored procedure as and when you want to, with the help of the ALTER proc command.
* Security: Stored procedures allow you to enhance the security of an application or a database by restricting the users from direct access to the table.
* Low network traffic: The server only passes the procedure name instead of the whole query, reducing network traffic.
* Increases performance: Upon the first use, a plan for the stored procedure is created and stored in the buffer pool for quick execution for the next time.

Stored Procedure Syntax:

CREATE PROCEDURE procedure\_name  
AS

Begin

sql\_statement (Code/Calculations as per project requirement)

End

Execute/Run a Stored Procedure

EXEC procedure\_name

EXECute procedure\_name

In the syntax mentioned above, the only thing to note here are the parameters, which can be the following three types:

* IN: It is the default parameter that will receive input value from the program
* OUT: It will send output value to the program
* IN OUT: It is the combination of both IN and OUT. Thus, it receives from, as well as sends a value to the program

We can write stored procedures for Select, Insert, Update, Delete functionalities & critical calculations.

**Parameters** are used to exchange data between stored procedures and functions and the application or tool that called the stored procedure or function: Input parameters allow the caller to pass a data value to the stored procedure or function.

A **variable** is an object that holds a single value of a specific type e.g., [integer](https://www.sqlservertutorial.net/sql-server-basics/sql-server-int/), [date](https://www.sqlservertutorial.net/sql-server-basics/sql-server-date/), or [varying character string](https://www.sqlservertutorial.net/sql-server-basics/sql-server-varchar/).

CREATE PROCEDURE Get\_AgentData

@AgentId VARCHAR(20)

As

Begin

SET NOCOUNT ON

Select \* from AGENTS A

INNER JOIN CUSTOMER C

ON A.AGENT\_CODE=C.AGENT\_CODE

WHERE A.AGENT\_CODE=@AgentId

End

EXEC Get\_AgentData 'A009'

We can modify stored procedure using alter command.

Alter Proc Proc\_Name

@Parameters

As

( Begin

End)

Also to rename proc we have system stored procedure sp\_Rename

Exec sp\_Rename ‘OldName’, ’New Name’

To delete procedure use DROP Proc “ProcName” command

**Stored Procedure with Input Parameters:**

Consider the following stored procedure example with the input parameters.

CREATE PROCEDURE uspUpdateEmpSalary

(

@empId int

,@salary money

)

AS

BEGIN

UPDATE dbo.Employee

SET Salary = @salary

WHERE EmployeeID = @empId

END

In the above stored procedure uspUpdateEmpSalary, the @empId and @Salary are INPUT parameters. By default, all the parameters are INPUT parameters in any stored procedure unless suffix with OUTPUT keyword. @empId is of int type and @salary is of money data type. You pass the INPUT parameters while executing a stored procedure, as shown below.

EXEC dbo.uspUpdateEmpSalary @EmpId = 4, @Salary = 25000

-- or

EXEC dbo.uspUpdateEmpSalary 4, 25000

**OUTPUT Parameters:**

The OUTPUT parameter is used when you want to return some value from the stored procedure. The calling program must also use the OUTPUT keyword while executing the procedure.

The following stored procedure contains INPUT and OUTPUT parameters.

CREATE PROCEDURE uspGetManagerID

@empId int,

@managerId int OUTPUT

AS

BEGIN

SELECT @managerId = ManagerID

FROM dbo.Employee

WHERE EmployeeID = @empId

END

In the above uspGetManagerID stored procedure, @manageId is an OUTPUT parameter. The value will be assigned in the stored procedure and returned to the calling statement. The following pass the OUTPUT parameter while executing the stored procedure.

DECLARE @managerID int

EXECUTE uspGetManagerID @empId = 2, @managerId OUTPUT

PRINT @managerId

Above, the uspGetManagerID is called by passing INPUT parameter @employeeID = 2 and @managerID OUTPUT as the output parameter. Notice that we have not assigned any value to an OUTPUT variable @managerID and also specified the OUTPUT keyword.

There are a total of three methods of returning data from a stored procedure: OUTPUT parameter, result sets, and return codes.

Result sets: If the body of the stored procedure has a SELECT statement, then the rows returned by the select statement are directly returned to the client.

Return code: A stored procedure can return an integer value called the Return code which will indicate the execution status of the procedure. You specify the return code using the RETURN keyword in the procedure.

## Optional Parameters

SQL Server allows you to specify the default values for parameters. It allows you to skip the parameters that have default values when calling a stored procedure.

The default value is used when no value is passed to the parameter or when the DEFAULT keyword is specified as the value in the procedure call.

Specify the default value when you declare parameters, as shown below.

CREATE PROCEDURE uspUpdateEmpSalary

(

@empId int

,@salary money = 1000

)

AS

BEGIN

UPDATE dbo.Employee

SET Salary = @salary

WHERE EmployeeID = @empId

END

Above, @empsalary money = 0 declares @salary parameter and assigns the default value. Now, you can call the above procedure without passing @salary parameter, as shown below.

EXEC uspUpdateEmpSalary 4

The above statement will update the Salary column with the default value 1000 for the EmployeeID 4. Thus, making @salary parameter as optional.

**Useful System Stored Procedures:**

1. sp\_depends: sp\_depends is a system stored procedure that displays information about all object types (e.g. procedures, tables, etc) that depend on the object specified in the input parameter as well as all objects that the specified object depends on.

Sample Call:

EXEC sp\_depends tablename

EXEC sp\_depends procname

EXEC sp\_depends uspUpdateEmpSalary

EXEC sp\_depends uspGetEmpDept

1. sp\_help: On Transact SQL language the sp\_help is part of Database Engine Stored Procedures and reports information about a database object or a data type.

Sp\_help example 1:

EXEC sp\_help;

Sp\_help example 2:  
EXEC sp\_help 'students';

**Error Handling in SQL Server:**

With the introduction of Try/Catch blocks in SQL Server 2005, error handling in sql server, is now similar to programming languages like C#, and java. Before understanding error handling using try/catch, let's step back and understand how error handling was done in SQL Server 2000, using system function **@@Error**. Sometimes, system functions that begin with two at signs (@@), are called as global variables. They are not variables and do not have the same behaviours as variables, instead they are very similar to functions.

The **RAISERROR** statement allows you to generate your own error messages and return these messages back to the application using the same format as a system error or warning message generated by SQL Server Database Engine. In addition, the RAISERROR statement allows you to set a specific message id, level of severity, and state for the error messages.

Syntax: RAISEERROR('Message Text',Severity,State)

**Severity**

The severity level is an integer between 0 and 25, with each level representing the seriousness of the error.

0–10 Informational messages

11–18 Errors

19–25 Fatal errors

**State**

The state is an integer from 0 through 255. If you raise the same user-defined error at multiple locations, you can use a unique state number for each location to make it easier to find which section of the code is causing the errors. For most implementations, you can use 1.

BEGIN TRY

Insert into tblProduct values(1, 'Laptops', 3000, 150)

END TRY

BEGIN CATCH

DECLARE @ErrorMessage VARCHAR(500)

SET @ErrorMessage=ERROR\_MESSAGE()

RAISERROR(@ErrorMessage,16,1)

END CATCH

You can take advantage of various functions inside the CATCH block to get detailed information about an error.

These functions include the following:

* ERROR\_MESSAGE() - you can take advantage of this function to get the complete error message.
* ERROR\_LINE() - this function can be used to get the line number on which the error occurred.
* ERROR\_NUMBER() - this function can be used to get the error number of the error.
* ERROR\_SEVERITY() - this function can be used to get the severity level of the error.
* ERROR\_STATE() - this function can be used to get the state number of the error.
* ERROR\_PROCEDURE() - this function can be used to know the name of the stored procedure or trigger that has caused the error.

Now let's create **tblProduct** and **tblProductSales**, that we will be using for the rest of this demo.  
  
**SQL script to create tblProduct**  
Create Table tblProduct  
(  
 ProductId int NOT NULL primary key,  
 Name nvarchar(50),  
 UnitPrice int,  
 QtyAvailable int  
)  
  
**SQL script to load data into tblProduct**  
Insert into tblProduct values(1, 'Laptops', 3000, 150), (2, 'Desktops', 3500, 120), (2, 'TV', 2900, 110)

**SQL script to create tblProductSales**  
Create Table tblProductSales  
(  
 ProductSalesId int primary key,  
 ProductId int,  
 QuantitySold int  
)

Now consider the following code snippet that illustrates how an error generated inside a TRY block is handled in the CATCH block and the relevant error metadata displayed.

BEGIN TRY

Insert into tblProduct values(1, 'Laptops', 3000, 150)

END TRY

BEGIN CATCH

SELECT ERROR\_MESSAGE() AS [Error Message]

,ERROR\_LINE() AS ErrorLine

,ERROR\_NUMBER() AS [Error Number]

,ERROR\_SEVERITY() AS [Error Severity]

,ERROR\_STATE() AS [Error State]

END CATCH

**Stored procedure -** usp\_SellProduct, has 2 parameters - **@ProductId** and **@QuantityToSell**. @ProductId specifies the product that we want to sell, and @QuantityToSell specifies, the quantity we would like to sell

Create Procedure usp\_SellProduct

@ProductId int,

@QuantityToSell int

as

Begin

-- Check the stock available, for the product we want to sell

Declare @StockAvailable int

Select @StockAvailable = QtyAvailable

from tblProduct where ProductId = @ProductId

-- Throw an error to the calling application, if enough stock is not available

IF(@StockAvailable < @QuantityToSell)

Begin

Raiserror('Not enough stock available',16,1)

End

-- If enough stock available

Else

Begin

Begin Tran

-- First reduce the quantity available

Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)

where ProductId = @ProductId

Declare @MaxProductSalesId int

-- Calculate MAX ProductSalesId

Select @MaxProductSalesId = Case When

MAX(ProductSalesId) IS NULL

Then 0 else MAX(ProductSalesId) end

from tblProductSales

-- Increment @MaxProductSalesId by 1, so we don't get a primary key violation

Set @MaxProductSalesId = @MaxProductSalesId + 1

Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)

Commit Tran

End

End

Exec usp\_SellProduct 1,10

–-There will not be any error. And operations in both the table will complete without any error.

In above stored proc try to insert some error. We can do that by commenting line where we are setting value in variable @ MaxProductSalesId, if we comment this line & alter proc.

Now if we execute above call again as below: it will throw error due to Primary key violation in tblProductSales table.

Exec usp\_SellProduct 1,10

Here table @tblProduct will get updated but data insertion will fail in table tblProductSales. Which in turn leads to inaccurate data. To overcome this, we must capture error & if any code inside transaction fails all the transaction should be rolled back.

SQL Server was using Try-Catch block to capture any error.

**Syntax:**  
BEGIN TRY  
     { Any set of SQL statements }  
END TRY  
BEGIN CATCH  
     [ Optional: Any set of SQL statements ]  
END CATCH  
[Optional: Any other SQL Statements]  
  
**Any set of SQL statements**, that can possibly throw an exception are wrapped between BEGIN TRY and END TRY blocks. If there is an exception in the TRY block, the control immediately, jumps to the CATCH block. If there is no exception, CATCH block will be skipped, and the statements, after the CATCH block are executed.

**Errors trapped by a CATCH block are not returned to the calling application**. If any part of the error information must be returned to the application, the code in the CATCH block must do so by using RAISERROR() function.  
  
1. **In procedure spSellProduct**, Begin Transaction and Commit Transaction statements are wrapped between Begin Try and End Try block. If there are no errors in the code that is enclosed in the TRY block, then COMMIT TRANSACTION gets executed and the changes are made permanent. On the other hand, if there is an error, then the control immediately jumps to the CATCH block. In the CATCH block, we are rolling the transaction back. So, it's much easier to handle errors with Try/Catch construct than with @@Error system function.  
  
2. Also notice that, in the scope of the CATCH block, there are several system functions, that are used to retrieve more information about the error that occurred, these functions return NULL if they are executed outside the scope of the CATCH block.  
  
3. TRY/CATCH cannot be used in a user-defined functions.

Create Procedure spSellProduct

@ProductId int,

@QuantityToSell int

as

Begin

Declare @StockAvailable int

Select @StockAvailable = QtyAvailable

from tblProduct where ProductId = @ProductId

if(@StockAvailable < @QuantityToSell)

Begin

Raiserror('Not enough stock available',16,1)

End

Else

Begin

Begin Try

Begin Transaction

Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)

where ProductId = @ProductId

Declare @MaxProductSalesId int

Select @MaxProductSalesId = Case When MAX(ProductSalesId) IS NULL

Then 0 else MAX(ProductSalesId)

end

from tblProductSales

Set @MaxProductSalesId = @MaxProductSalesId + 1

Insert into tblProductSales values(@MaxProductSalesId, ProductId,@QuantityToSell)

Commit Transaction

End Try

Begin Catch

Rollback Transaction

Select

ERROR\_NUMBER() as ErrorNumber,

ERROR\_MESSAGE() as ErrorMessage,

ERROR\_PROCEDURE() as ErrorProcedure,

ERROR\_STATE() as ErrorState,

ERROR\_SEVERITY() as ErrorSeverity,

ERROR\_LINE() as ErrorLine

End Catch

End

End

Earlier version of SQL server was using @@Error system function to capture the error.

**Note**: @@ERROR is cleared and reset on each statement execution. Check it immediately following the statement being verified, or save it to a local variable that can be checked later.  
  
In **tblProduct** table, we already have a record with **ProductId = 2**. So the insert statement causes a primary key violation error. @@ERROR retains the error number, as we are checking for it immediately after the statement that cause the error.

Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)  
if(@@ERROR <> 0)  
 Print 'Error Occurred'  
Else  
 Print 'No Errors'  
  
On the other hand, when you execute the code below, you get message**'No Errors'** printed. This is because the @@ERROR is cleared and reset on each statement execution.   
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)  
--At this point @@ERROR will have a NON ZERO value   
Select \* from tblProduct  
--At this point @@ERROR gets reset to ZERO, because the   
--select statement successfullyexecuted  
if(@@ERROR <> 0)  
 Print 'Error Occurred'  
Else  
 Print 'No Errors'  
  
In this example, we are storing the value of @@Error function to a local variable, which is then used later.  
Declare @Error int  
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)  
Set @Error = @@ERROR  
Select \* from tblProduct  
if(@Error <> 0)  
 Print 'Error Occurred'  
Else  
 Print 'No Errors'

Database🡪Programmability🡪Stored Procedures

**Functions in SQL Server:**

Functions in SQL Server are the database objects that contains a **set of SQL statements to perform a specific task**. A function accepts input parameters, perform actions, and then return the result. We should note that functions always return either a single value or a table. The main purpose of functions is to replicate the common task easily. We can build functions one time and can use them in multiple locations based on our needs. SQL Server does not allow to use of the functions for inserting, deleting, or updating records in the database tables.

**The following are the rules for creating SQL Server functions:**

* A function must have a name, and the name cannot begin with a special character such as @, $, #, or other similar characters.
* SELECT statements are the only ones that operate with functions.
* We can use a function anywhere such as AVG, COUNT, SUM, MIN, DATE, and other functions with the SELECT query in SQL.
* Whenever a function is called, it compiles.
* Functions must return a value or result.
* Functions use only input parameters.
* We cannot use TRY and CATCH statements in functions.

**Types of Functions**

SQL Server categorizes the functions into two types:

* System Functions
* User-Defined Functions

Let us describe both types in detail.

**System Functions**

Functions that are defined by the system are known as system functions. In other words, all the **built-in functions** supported by the server are referred to as System functions. The built-in functions save us time while performing the specific task. These types of functions usually work with the SQL SELECT statement to calculate values and manipulate data.

**Here is the list of some system functions used in the SQL Server:**

* String Functions (LEN, SUBSTRING, REPLACE, CONCAT, TRIM)
* Date and Time Functions (datetime, datetime2, smalldatetime)
* Aggregate Functions (COUNT, MAX, MIN, SUM, AVG)
* Mathematical Functions (ABS, POWER, SQUARE, SQRT, LOG)
* Ranking Functions (RANK, DENSE\_RANK, ROW\_NUMBER, NTILE)

You can find system functions in SSMS as below path:

Database🡪Programmability🡪Functions🡪System Functions

**Ranking or Window Functions in SQL Server:**

The Ranking functions in SQL Server return a ranking value for each row in a partition. Microsoft provides various Functions which allow us to assign different ranks. Depending on the function you select, they return a different number. The following table will show you the list of available Ranking Functions.

|  |  |
| --- | --- |
| **Functions** | **Description** |
| Row\_Number() | It will assign the sequential number to each unique record present in a partition. |
| Rank() | It will assign the rank number to each record present in a partition. |
| Dense\_Ranke() | It will assign the number to each record within a partition without skipping the numbers. |
| NTILE | This will assign the number to each record present in a partition. |
| Lag and Lead | It can often be useful to compare rows to preceding or following rows, |

# **SQL ROW\_NUMBER**

The ROW\_NUMBER Function is one of the Ranking functions. This SQL Server row\_number function assigns the sequential rank number to each unique record present in a partition.

If the SQL Server ROW\_NUMBER function encounters two equal values in the same partition, it will assign the different rank numbers to both values. Here rank numbers will depend upon the order they are displayed.

The syntax of the ROW\_NUMBER Function is:

SELECT ROW\_NUMBER() OVER (PARTITION\_BY\_Clause ORDER\_BY\_Clause) FROM [Source]

Partition\_By\_Clause: Divide the records into partitions.

* If you specify the Partition By Clause, ROW\_NUMBER Function will assign the rank number to each partition.
* If you haven’t defined the Partition By, the Function will consider all the records as a single partition. So, it will assign the rank numbers from top to bottom.

ROW\_NUMBER functions without Partition\_By clause:

SELECT Studentname, Subject, Marks, ROW\_NUMBER() OVER(ORDER BY Marks) RowNumber

FROM ExamResult;

By default, it sorts the data in ascending order and starts assigning ranks for each row. In the result, we get ROW number 1 for marks 50.

We can specify descending order with Order By clause, and it changes the RANK accordingly.

In the result we can see each row has given a sequential rank starting from 1.

ROW\_NUMBER functions with Partition\_By clause:

SELECT Studentname, Subject, Marks, ROW\_NUMBER() OVER(PARTITION BY StudentName ORDER BY Marks) RowNumber

FROM ExamResult;

When we use partition by, SQL will give sequential rank to each row in partition.

**RANK() SQL RANK Function**

The SQL RANK function will assign the rank number to each record present in a partition. If the SQL Server RANK function encounters two equal values in the same partition, then it will assign the same number to both values. And it skips the next number.

Syntax:

SELECT RANK() OVER (PARTITION\_BY\_Clause ORDER\_BY\_Clause) FROM [Source]

RANK functions without Partition\_By clause:

SELECT Studentname, Subject, Marks, RANK() OVER(ORDER BY Marks) Rank FROM ExamResult;

RANK functions with Partition\_By clause:

SELECT Studentname, Subject, Marks, RANK() OVER(PARTITION BY StudentName ORDER BY Marks) Rank FROM ExamResult;

# **SQL DENSE\_RANK Function**

The SQL DENSE\_RANK Function is one of the Ranking functions. The Sql Server DENSE\_RANK function will assign the rank number to each record present in a partition without skipping the rank numbers.

If the DENSE\_RANK function encounters two equal values in the same partition, it will assign the same rank number to both values. In this article, we will show you, How to write Sql DENSE\_RANK Function with an example.

Syntax:

SELECT DENSE\_RANK() OVER (PARTITION\_BY\_Clause ORDER\_BY\_Clause) FROM [Source]

-- DENSE\_RANK functions without Partition\_By clause:

SELECT Studentname, Subject, Marks, dense\_RANK() OVER(ORDER BY Marks) DenseRank FROM ExamResult;

-- DENSE\_RANK functions with Partition\_By clause:

SELECT Studentname, Subject, Marks, dense\_RANK() OVER(PARTITION BY StudentName ORDER BY Marks) DenseRank FROM ExamResult;

**SQL NTILE Function**

The SQL Server NTILE Function is one of the ranking functions. This NTILE function will assign the rank number to each record present in a partition, and the syntax of it is:

SELECT NTILE(Interger\_Value) OVER (PARTITION\_BY\_Clause ORDER\_BY\_Clause)

FROM [Source]

Integer\_Value: It will use this integer value to decide the number of ranks it has to assign for each partition. For instance, If we specify 2, the NTILE Function will assign 2 rank numbers for each category.

-- NTILE functions without Partition\_By clause:

SELECT Studentname, Subject, Marks, NTILE(2) OVER(ORDER BY Marks) DenseRank FROM ExamResult;

--NTILE functions with Partition\_By clause:

SELECT Studentname, Subject, Marks, NTILE(2) OVER(PARTITION BY StudentName ORDER BY Marks) DenseRank FROM ExamResult;

**SQL Server User-defined Functions**

In this section, you will learn about SQL Server user-defined functions including scalar-valued functions which return a single value and table-valued function which return rows of data.

The SQL Server user-defined functions help you simplify your development by encapsulating complex business logic and make them available for reuse in every query.

1. User Defined Functions in SQL Server
2. Types of User Defined Functions
3. Creating a Scalar User Defined Function
4. Calling a Scalar User Defined Function
5. Places where we can use Scalar User Defined Function
6. Altering and Dropping a User Defined Function

**In SQL Server there are 3 types of User Defined functions**  
1. Scalar functions  
2. Inline table-valued functions  
3. Multi statement table-valued functions

**Scalar functions** may or may not have parameters, but always return a single (scalar) value. The returned value can be of any data type, except **text, ntext, image, cursor, and timestamp**.

1. **Scalar Functions:**

**To create a function, we use the following syntax: (Scalar function will always return a single value)**  
CREATE FUNCTION Function\_Name(@Parameter1 DataType, @Parameter2 DataType,..@Parametern Datatype)  
RETURNS Return\_Datatype  
AS  
BEGIN  
    Function Body  
    Return Return\_Datatype  
END

Let us now create a function which calculates and returns the age of a person. To compute the age we require, date of birth. So, let's pass date of birth as a parameter. So, AGE() function returns an integer and accepts date parameter.

CREATE FUNCTION Age(@DOB Date)    
RETURNS INT    
AS    
BEGIN    
  DECLARE @Age INT    
  SET @Age = DATEDIFF(YEAR, @DOB, GETDATE()) - CASE WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR (MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) > DAY(GETDATE())) THEN 1 ELSE 0 END    
RETURN @Age    
END

**When calling a scalar user-defined function**, you must supply a two-part name,  **OwnerName.FunctionName**. **dbo** stands for database owner.  
Select dbo.Age ('10/08/1982')

**You can also invoke it using the complete 3 part name**, DatabaseName.OwnerName.FunctionName.  
Select SampleDB.dbo.Age('10/08/1982')

**Scalar user defined functions can be used in the Select clause** as shown below.  
Select EmployeeNo,LastName,Department, dbo.Age(dob) as Age from EmpSalary

**Scalar user defined functions can be used in the Where clause**, as shown below.  
Select EmployeeNo,LastName,Department,dbo.Age(DOB) as Age

from EmpSalary Where dbo.Age(DOB) > 30

**A stored procedure** also can accept DateOfBirth and return Age, but you cannot use stored procedures in a **select or where clause**. This is just one difference between a function and a stored procedure. There are several other differences, which we will talk about in a later session.  
  
To alter a function we use ALTER FUNCTION FuncationName statement and to delete it, we use DROP FUNCTION FuncationName.

1. **Inline Table Valued Functions**

A scalar function, returns a **single**value. on the other hand, an Inline Table Valued function, return a **table**.

**Syntax for creating an inline table valued function**  
CREATE FUNCTION Function\_Name(@Param1 DataType, @Param2 DataType..., @ParamN DataType)  
RETURNS TABLE  
AS  
RETURN (Select\_Statement)

CREATE FUNCTION fn\_EmployeesByGender(@Gender nvarchar(10))

RETURNS TABLE

AS

RETURN (Select Id, Name, DateOfBirth, Gender, DepartmentId

from tblEmployees

where Gender = @Gender)

**If you look at the way we implemented this function**, it is very similar to SCALAR function, with the following differences  
1. We specify **TABLE**as the return type, instead of any **scalar** data type  
2. The **function body** is not enclosed between **BEGIN and END** block. Inline table valued function body, cannot have BEGIN and END block.  
3. The **structure of the table** that gets returned, is determined by the SELECT statement with in the function.

**Calling the user defined function**  
Select \* from fn\_EmployeesByGender('Male')

As the inline user defined function, is returning a table, issue the select statement against the function, as if you are selecting the data from a TABLE.  
  
**Where can we use Inline Table Valued functions**  
1. Inline Table Valued functions can be used to achieve the functionality of parameterized views. We will talk about views, in a later session.  
2. The table returned by the table valued function, can also be used in joins with other tables.  
  
**Consider the Departments Table**

**Joining the Employees returned by the function, with the Departments table**  
Select Name, Gender, Dept\_name

from fn\_EmployeesByGender('Male') E

Join Department D on D.Dept\_id = E.DepartmentId

1. **Multi Statement Table Valued Functions**

Multi statement table valued functions are very similar to Inline Table valued functions, with a few differences. Let's look at an example, and then note the differences.

**Let's write an Inline and multi-statement Table Valued functions that can return the Id, Name & DOB from employees table in output**

**Inline Table Valued function(ILTVF):**  
Create Function fn\_ILTVF\_GetEmployees()  
Returns **Table**  
as  
Return (Select Id, Name, Cast(DateOfBirth as Date) as DOB  
        From tblEmployees)

**Multi-statement Table Valued function(MSTVF):**  
Create Function fn\_MSTVF\_GetEmployees()  
Returns @Table Table (Id int, Name varchar(20), DOB Date)  
as  
Begin  
 Insert into @Table  
 Select Id, Name, Cast(DateOfBirth as Date)  
 From tblEmployees  
   
 Return  
End  
  
**Calling the Inline Table Valued Function:**  
Select \* from fn\_ILTVF\_GetEmployees()  
  
**Calling the Multi-statement Table Valued Function:**  
Select \* from fn\_MSTVF\_GetEmployees()

**Now let's understand the differences between Inline Table Valued functions and Multi-statement Table Valued functions**  
1. In an Inline Table Valued function, the RETURNS clause cannot contain the structure of the table, the function returns. Where as, with the multi-statement table valued function, we specify the structure of the table that gets returned  
2. Inline Table Valued function cannot have BEGIN and END block, where as the multi-statement function can have.  
3. Inline Table valued functions are better for performance, than multi-statement table valued functions. If the given task, can be achieved using an inline table valued function, always prefer to use them, over multi-statement table valued functions.  
4. It's possible to update the underlying table, using an inline table valued function, but not possible using multi-statement table valued function.  
  
**Updating the underlying table using inline table valued function:**  
This query will change **Sam** to **Sam1**, in the underlying table **tblEmployees**. When you try do the same thing with the multi-statement table valued function, you will get an error stating 'Object 'fn\_MSTVF\_GetEmployees' cannot be modified.'  
Update fn\_ILTVF\_GetEmployees() set Name='Sam1' Where Id = 1  
  
**Reason for improved performance of an inline table valued function:**  
Internally, SQL Server treats an inline table valued function much like it would a view and treats a multi-statement table valued function similar to how it would a stored procedure.

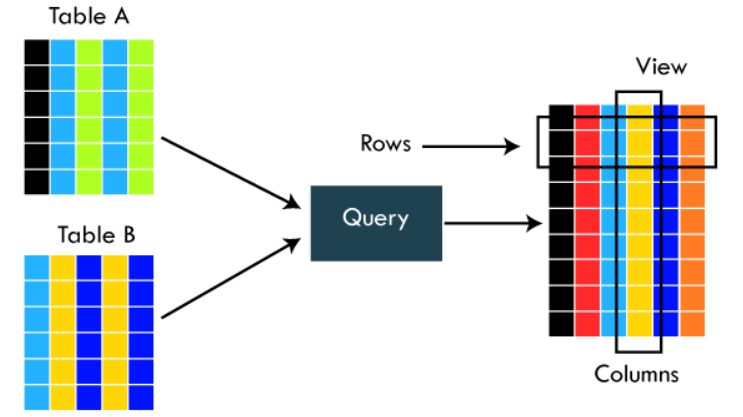
**Key Differences between SQL server store procedure and function.**

Let’s discuss some of the major **differences between SQL server store procedure and function**.

|  |  |
| --- | --- |
| **Stored Procedures** | **Functions** |
| A stored procedure in SQL Server can have input as well as [output parameters](https://sqlserverguides.com/sql-server-stored-procedure-output-parameter/). | A function, on the other hand, can only have input parameters. |
| A stored procedure can return numerous parameters | A function can only return one value or defined table |
| Stored procedure may or may not return value (it is optional) | A function in SQL Server must return a value |
| We cannot use a procedure in a function | We can use a function within a stored procedure |
| We can handle errors in a stored procedure by using the TRY-CATCH block | We cannot use TRY-CATCH in functions |
| We can use SELECT as well as DML (INSERT/UPDATE /DELETE) statements in a stored procedure | We can only use SELECT statements within a function, as DML statements are not supported in functions |
| We cannot use a stored procedure in a SELECT statement | We can easily use functions in a SELECT statement |
| A procedure is not supported to be used in WHERE/HAVING/SELECT statements | We can use the functions in WHERE/HAVING/SELECT section |
| We can use the transactions in the stored procedure | We cannot use it in functions |

**View in SQL Server**

A view is a database object that has no values. **It is a virtual table, which is created according to the result set of an SQL query**. However, it looks similar to an actual table containing rows and columns. Therefore, we can say that its contents are based on the base table. It is operated similarly to the base table but does not contain any data of its own. **Its name is always unique, like tables**. The views differ from tables as they are definitions that are created on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the views also.

****

**Types Of Views:**

**1. User-Defined Views**

**Users define these views to meet their specific requirements**. It can also divide into two types one is the simple view, and another is the complex view. The **simple view** is based on the single base table without using any complex queries. The **complex view** is based on more than one table along with group by clause, order by clause, and join conditions.

**2. System-Defined Views**

**System-defined views are predefined and existing views stored in SQL Server**, such as Tempdb, Master, and temp. Each system views has its own properties and functions. They can automatically attach to the user-defined databases. We can divide the System-defined views in SQL Server into three types: Information Schema, Catalog View, and Dynamic Management View.

**Syntax for Views:**

**Create View ViewName**

**AS**

**Select statement**

**Let's understand views with an example**. We will base all our examples on **tblEmployees** and **tblDepartment** tables.

**Now, let's write a Query which returns Id,Name,Salary,Gender,DepartmentName in output:**

Select E.Id, E.Name, E.Salary, E.Gender, D.DeptName from tblEmployees E

join tblDepartment D on E.DepartmentId = D.DeptId

Now let's create a view, using the JOINS query, we have just written.

Create View vWEmployeesByDepartment

as

Select Id, Name, Salary, Gender, DeptName from tblEmployees

join tblDepartment on tblEmployee.DepartmentId = tblDepartment.DeptId

**To select data from the view**, SELECT statement can be used the way, we use it with a table.  
SELECT \* from vWEmployeesByDepartment

**When this query is executed**, the database engine actually retrieves the data from the underlying base tables, **tblEmployees and tblDepartments**. The View itself, does not store any data by default. However, we can change this default behaviour, which we will talk about in a later session. So, this is the reason, a view is considered, as just, a stored query or a virtual table.  
  
**Advantages of using views:**  
1. Views can be used to reduce the **complexity of the database schema**, for non-IT users. The sample view, **vWEmployeesByDepartment**, hides the complexity of joins. Non-IT users, finds it easy to query the view, rather than writing complex joins.

2. Views can be used as a mechanism to implement **row and column level security**.  
**Row Level Security:**  
For example, I want an end user, to have access only to IT Department employees. If I grant him access to the underlying tblEmployees and tblDepartments tables, he will be able to see, every department employees. To achieve this, I can create a view, which returns only IT Department employees, and grant the user access to the view and not to the underlying table.

**View that returns only IT department employees: Create view in SSMS**

**Column Level Security:**  
Salary is confidential information and I want to prevent access to that column. To achieve this, we can create a view, which excludes the Salary column, and then grant the end user access to this views, rather than the base tables.  
  
**View that returns all columns except Salary column: Create view in SSMS**

To look at view definition - sp\_helptext vWName  
To modify a view - ALTER VIEW statement   
To Drop a view - DROP VIEW vWName

To Rename view - SP\_RENAME View\_Old\_Name, View\_New\_Name

**Updatable Views:**

**Let's create a view**, which returns all the columns from the tblEmployees table, except Salary column.  
Create view vWEmployeesDataExceptSalary  
as  
Select Id, Name, Gender, DepartmentId from tblEmployees

Select \* from vWEmployeesDataExceptSalary  
  
**Is it possible to Insert, Update and delete rows**, from the underlying tblEmployees table, using view vWEmployeesDataExceptSalary?  
**Yes**, SQL server views are updateable.  
  
**The following query updates, Name column from Mike to Mikey**. Though, we are updating the view, SQL server, correctly updates the base table tblEmployees. To verify, execute, SELECT statement, on tblEmployee table.  
Update vWEmployeesDataExceptSalary Set Name = 'Mikey' Where Id = 2  
  
**Along the same lines**, it is also possible to insert and delete rows from the base table using views.  
Delete from vWEmployeesDataExceptSalary where Id = 2

**Now, let us see, what happens if our view is based on multiple base tables**.

**View that joins tblEmployee and tblDepartment**  
Create view vwEmployeeDetailsByDepartment  
as  
Select Id, Name, Salary, Gender, DeptName from tblEmployees  
join tblDepartment on tblEmployees.DepartmentId = tblDepartment.DeptId  
**Select Data from view vwEmployeeDetailsByDepartment**  
Select \* from vwEmployeeDetailsByDepartment

**Now, let's update, John's department, from HR to IT**. At the moment, there are 2 employees (Ben, and John) in the HR department.  
Update vwEmployeeDetailsByDepartment   
set DeptName='IT' where Name = 'John'

**Notice, that Ben's department is also changed to IT**. To understand the reasons for incorrect UPDATE, select Data from tblDepartment and tblEmployee base tables.

**The UPDATE statement, updated DeptName from HR to IT in tblDepartment table**, instead of upadting **DepartmentId** column in **tblEmployee** table. So, the conclusion - If a view is based on multiple tables, and if you update the view, it may not update the underlying base tables correctly. To correctly update a view, that is based on multiple tables, INSTEAD OF triggers are used.  
We will discuss about triggers and correctly updating a view that is based on multiple tables, in a later session.

**Indexed View**

**What is an Indexed View or What happens when you create an Index on a view?**  
A **standard** or **Non-indexed** view, is just a stored SQL query. When, we try to retrieve data from the view, the data is actually retrieved from the underlying base tables. So, a view is just a virtual table it does not store any data, by default.  
  
**However, when you create an index**, on a view, the view gets materialized. This means, the view is now, capable of storing data. In SQL server, we call them Indexed views and in Oracle, Materialized views.  
  
**Let's now, look at an example of creating an Indexed view**. For the purpose of this video, we will be using **Product** and **ProductSales** tables.

Script to create view vWTotalSalesByProduct

Create view vWTotalSalesByProduct

with SchemaBinding

as

Select Name,SUM(ISNULL((QuantitySold \* UnitPrice), 0)) as TotalSales, COUNT\_BIG(\*) as TotalTransactions

from dbo.ProductSales

join dbo.Product on dbo. Product.ProductId = dbo.ProductSales.ProductId

group by Name

**If you want to create an Index**, on a view, the following rules should be followed by the view. For the complete list of all rules, please check [MSDN](http://msdn.microsoft.com/en-us/library/ms191432(v=sql.105).aspx).  
1. The view should be created with SchemaBinding option  
  
2. If an Aggregate function in the SELECT LIST, references an expression, and if there is a possibility for that expression to become NULL, then, a replacement value should be specified. In this example, we are using, ISNULL() function, to replace NULL values with ZERO.  
  
3. If GROUP BY is specified, the view select list must contain a COUNT\_BIG(\*) expression  
  
4. The base tables in the view, should be referenced with 2 part name. In this example, tblProduct and tblProductSales are referenced using dbo.tblProduct and dbo.tblProductSales respectively.

**Now, let's create an Index on the view:**  
The first index that you create on a view, must be a unique clustered index. After the unique clustered index has been created, you can create additional nonclustered indexes.  
Create Unique Clustered Index UIX\_vWTotalSalesByProduct\_Name  
on vWTotalSalesByProduct(Name)  
  
**Since, we now have an index on the view, the view gets materialized**. The data is stored in the view. So when we execute Select \* from vWTotalSalesByProduct, the data is retrurned from the view itself, rather than retrieving data from the underlying base tables.  
Indexed views, can significantly improve the performance of queries that involves JOINS and Aggeregations. The cost of maintaining an indexed view is much higher than the cost of maintaining a table index.  
  
Indexed views are ideal for scenarios, where the underlying data is not frequently changed. Indexed views are more often used in OLAP systems, because the data is mainly used for reporting and analysis purposes. Indexed views, may not be suitable for OLTP systems, as the data is frequently added and changed.

**Advantages of views in SQL Server**

1. Hiding the complexity of a Complex SQL Query
2. Implementing Row Level Security
3. Implementing Column Level Security
4. Presenting the Aggregated data by Hiding Detailed data

**Limitations and Dis-Advantages of Views in SQL Server**

1. We cannot pass parameters to SQL Server views
2. Cannot use Order By clause with views without specifying FOR XML, OFFSET, or TOP
3. The Views cannot be created based on Temporary Tables in SQL Server
4. We cannot associate Rules and Defaults with SQL Server views

**Triggers in SQL Server**

A trigger is a set of SQL statements that reside in system memory with unique names. It is a specialized category of stored procedure that is called automatically when a database server event occurs. Each trigger is always associated with a table.

A **trigger is called a special procedure** because it cannot be called directly like a stored procedure. The key distinction between the trigger and procedure is that a trigger is called automatically when a data modification event occurs against a table. A stored procedure, on the other hand, must be invoked directly.

The following are the main characteristics that distinguish triggers from stored procedures:

* We cannot manually execute/invoked triggers.
* Triggers have no chance of receiving parameters.
* A transaction cannot be committed or rolled back inside a trigger.

**Syntax of Trigger**

We can create a trigger in [SQL Server](https://www.javatpoint.com/sql-server-tutorial) by using the **CREATE TRIGGER** statement as follows:

CREATE TRIGGER schema.trigger\_name

ON table\_name

AFTER  {INSERT, UPDATE, DELETE}

[NOT FOR REPLICATION]

AS

{SQL\_Statements}

**The parameter descriptions of this syntax illustrate below:**

**schema:** It is an optional parameter that defines which schema the new trigger belongs to.

**trigger\_name:** It is a required parameter that defines the name for the new trigger.

**table\_name:** It is a required parameter that defines the table name to which the trigger applies. Next to the table name, we need to write the AFTER clause where any events like INSERT, UPDATE, or DELETE could be listed.

**NOT FOR REPLICATION:** This option tells that [SQL](https://www.javatpoint.com/sql-tutorial) Server does not execute the trigger when data is modified as part of a replication process.

**SQL\_Statements:** It contains one or more SQL statements that are used to perform actions in response to an event that occurs.

When we use triggers?

Triggers will be helpful when we need to execute some events automatically on certain desirable scenarios. **For example**, we have a constantly changing table and need to know the occurrences of changes and when these changes happen. If the primary table made any changes in such scenarios, we could create a trigger to insert the desired data into a separate table.

**DML stands for Data Manipulation Language.** INSERT, UPDATE, and DELETE statements are DML statements. DML triggers are fired, whenever data is modified using INSERT, UPDATE, and DELETE events.  
  
**DML triggers can be again classified into 2 types.**  
1. After triggers (Sometimes called as FOR triggers)  
2. Instead of triggers  
  
**After triggers, as the name says, fires after the triggering action**. The INSERT, UPDATE, and DELETE statements, causes an after trigger to fire after the respective statement’s complete execution.  
  
**On the other hand, as the name says, INSTEAD of triggers, fires instead of the triggering action**. The INSERT, UPDATE, and DELETE statements, can cause an INSTEAD OF trigger to fire INSTEAD OF the respective statement execution.

**We will use tblEmployees and tblEmployeeAudit**tables for our examples

**Whenever, a new Employee is added**, we want to capture the ID and the date and time, the new employee is added in tblEmployeeAudit table. The easiest way to achieve this, is by having an AFTER TRIGGER for INSERT event.  
  
**Example for AFTER TRIGGER for INSERT event on tblEmployee table:**  
CREATE TRIGGER tr\_tblEmployee\_ForInsert  
ON tblEmployees  
FOR INSERT  
AS  
BEGIN  
 Declare @Id int  
 Select @Id = Id from inserted  
   
 insert into tblEmployeeAudit   
 values('New employee with Id  = ' + Cast(@Id as nvarchar(5)) + ' is added at ' + cast(Getdate() as nvarchar(20)))  
END  
  
**In the trigger, we are getting the id from inserted table.** So, what is this inserted table? INSERTED table, is a special table used by DML triggers. When you add a new row into tblEmployees table, a copy of the row will also be made into inserted table, which only a trigger can access. You cannot access this table outside the context of the trigger. The structure of the inserted table will be identical to the structure of tblEmployees table.

**So, now if we execute the following INSERT statement on tblEmployee.** Immediately, after inserting the row into tblEmployee table, the trigger gets fired (executed automatically), and a row into tblEmployeeAudit, is also inserted.

**Insert into tblEmployee values (7,'Tan', 2300, 'Female', 3)**  
  
**Along, the same lines, let us now capture audit information, when a row is deleted**from the table, tblEmployees.

**Example for AFTER TRIGGER for DELETE event on tblEmployees table:**  
CREATE TRIGGER tr\_tblEMployee\_ForDelete  
ON tblEmployees  
FOR DELETE  
AS  
BEGIN  
 Declare @Id int  
 Select @Id = Id from deleted  
   
 insert into tblEmployeeAudit values('An existing employee with Id  = ' + Cast(@Id as nvarchar(5)) + ' is deleted at ' + Cast(Getdate() as nvarchar(20)))  
END  
**The only difference here is that**, we are specifying, the triggering event as **DELETE** and retrieving the deleted row ID from **DELETED** table. DELETED table, is a special table used by DML triggers. When you delete a row from tblEmployees table, a copy of the deleted row will be made available in DELETED table, which only a trigger can access. Just like INSERTED table, DELETED table cannot be accessed, outside the context of the trigger and, the structure of the DELETED table will be identical to the structure of tblEmployees table.

**AFTER TRIGGER for Update**

**Triggers make use of 2 special tables**, INSERTED and DELETED. The inserted table contains the updated data and the deleted table contains the old data. The After trigger for UPDATE event, makes use of both inserted and deleted tables.   
  
**Create AFTER UPDATE trigger script:**  
ALTER trigger tr\_tblEmployee\_ForUpdate

on tblEmployees

for Update

as

Begin

Select \* INTO DeletedAudit from deleted

Select \* INTO InsertedAudit from inserted

End  
  
**Now, execute this query:**  
Update tblEmployee set Name = 'Tods', Salary = 2000,   
Gender = 'Female' where Id = 4

**Immediately after the UPDATE statement execution**, the AFTER-UPDATE trigger gets fired, and you should see the contents of INSERTED and DELETED tables.

**Instead Of Triggers:**

We know that, AFTER triggers are fired after the triggering event(INSERT, UPDATE or DELETE events), where as, INSTEAD OF triggers are fired instead of the triggering event(INSERT, UPDATE or DELETE events). In general, INSTEAD OF triggers are usually used to correctly update views that are based on multiple tables.

**Instead Of Insert Triggers:**

let's create a view based on tblEmployees  & tblDepartment tables. The view should return Employee Id, Name, Gender and DepartmentName columns. So, the view is obviously based on multiple tables.  
  
**Script to create the view:**  
Create view vWEmployeeDetails  
as  
Select Id, Name, Gender, DeptName  
from tblEmployees   
join tblDepartment  
on tblEmployee.DepartmentId = tblDepartment.DeptId

Select \* from vWEmployeeDetails

**Now, let's try to insert a row into the view, vWEmployeeDetails**, by executing the following query. At this point, an error will be raised stating 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables.'  
Insert into vWEmployeeDetails values(7, 'Valarie', 'Female', 'IT')  
  
**So, inserting a row into a view that is based on multiple tables**, raises an error by default. Now, let's understand, how INSTEAD OF TRIGGERS can help us in this situation. Since, we are getting an error, when we are trying to insert a row into the view, let's create an INSTEAD OF INSERT trigger on the view **vWEmployeeDetails.**

**Script to create INSTEAD OF INSERT trigger:**  
Create trigger tr\_vWEmployeeDetails\_InsteadOfInsert  
on vWEmployeeDetails  
Instead Of Insert  
as  
Begin  
 Declare @DeptId int  
   
 --Check if there is a valid DepartmentId  
 --for the given DepartmentName  
 Select @DeptId = DeptId   
 from tblDepartment   
 join inserted  
 on inserted.DeptName = tblDepartment.DeptName  
   
 --If DepartmentId is null throw an error  
 --and stop processing  
 if(@DeptId is null)  
 Begin  
 Raiserror('Invalid Department Name. Statement terminated', 16, 1)  
 return  
 End  
   
 --Finally insert into tblEmployee table  
 Insert into tblEmployees(Id, Name, Gender, DepartmentId)  
 Select Id, Name, Gender, @DeptId  
 from inserted  
End  
  
**Now, let's execute the insert query:**  
Insert into vWEmployeeDetails values(7, 'Valarie', 'Female', 'IT')  
  
**The instead of trigger correctly inserts**, the record into tblEmployees table. Since, we are inserting a row, the **inserted** table, contains the newly added row, whereas the **deleted** table will be empty.

**Instead Of Update Triggers:**

An INSTEAD OF UPDATE triggers gets fired instead of an update event, on a table or a view. For example, let's say we have, an INSTEAD OF UPDATE trigger on a view or a table, and then when you try to update a row with in that view or table, instead of the UPDATE, the trigger gets fired automatically. INSTEAD OF UPDATE TRIGGERS, are of immense help, to correctly update a view, that is based on multiple tables.

**Now, let's try to update the view**, in such a way that, it affects, both the underlying tables, and see, if we get the same error. The following UPDATE statement changes **Name column** from **tblEmployee** and **DeptName column** from **tblDepartment**. So, when we execute this query, we get the same error.  
Update vWEmployeeDetails   
set Name = 'Johny', DeptName = 'IT'  
where Id = 1  
  
**Now, let's try to change, just the department of John from HR to IT**. The following UPDATE query, affects only one table, tblDepartment. So, the query should succeed. But, before executing the query, please note that, employees **JOHN** and **BEN** are in **HR** department.  
Update vWEmployeeDetails   
set DeptName = 'IT'  
where Id = 1  
  
**After executing the query**, select the data from the view, and notice that **BEN's** **DeptName** is also changed to **IT**. We intended to just change **JOHN's DeptName**. So, the UPDATE didn't work as expected. This is because, the UPDATE query, updated the **DeptName from HR to IT**, in tblDepartment table. For the UPDATE to work correctly, we should change the **DeptId** of **JOHN** from 3 to 1.

**Record with Id = 3, has the DeptName changed from 'HR' to 'IT'**

**We should have actually updated, JOHN's DepartmentId from 3 to 1**

**So, the conclusion is that, if a view is based on multiple tables**, and if you update the view, the UPDATE may not always work as expected. To correctly update the underlying base tables, thru a view, INSTEAD OF UPDATE TRIGGER can be used.

**Before, we create the trigger, let's update the DeptName to HR for record with Id = 3.**

Update tblDepartment set DeptName = 'HR' where DeptId = 3  
  
**Script to create INSTEAD OF UPDATE trigger:**  
Create Trigger tr\_vWEmployeeDetails\_InsteadOfUpdate  
on vWEmployeeDetails  
instead of update  
as  
Begin  
 -- if EmployeeId is updated  
 if(Update(Id))  
 Begin  
 Raiserror('Id cannot be changed', 16, 1)  
 Return  
 End  
   
 -- If DeptName is updated  
 if(Update(DeptName))   
 Begin  
 Declare @DeptId int  
  
 Select @DeptId = DeptId  
 from tblDepartment  
 join inserted  
 on inserted.DeptName = tblDepartment.DeptName  
   
 if(@DeptId is NULL )  
 Begin  
 Raiserror('Invalid Department Name', 16, 1)  
 Return  
 End  
   
 Update tblEmployee set DepartmentId = @DeptId  
 from inserted  
 join tblEmployee  
 on tblEmployee.Id = inserted.id  
 End  
   
 -- If gender is updated  
 if(Update(Gender))  
 Begin  
 Update tblEmployee set Gender = inserted.Gender  
 from inserted  
 join tblEmployee  
 on tblEmployee.Id = inserted.id  
 End  
   
 -- If Name is updated  
 if(Update(Name))  
 Begin  
 Update tblEmployee set Name = inserted.Name  
 from inserted  
 join tblEmployee  
 on tblEmployee.Id = inserted.id  
 End  
End  
  
**Now, let's try to update JOHN's Department to IT.**  
Update vWEmployeeDetails   
set DeptName = 'IT'  
where Id = 1  
  
**The UPDATE query works as expected.** The INSTEAD OF UPDATE trigger, correctly updates, JOHN's DepartmentId to 1, in tblEmployee table.  
  
**Now, let's try to update Name, Gender and DeptName.** The UPDATE query, works as expected, without raising the error - 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables.'  
Update vWEmployeeDetails   
set Name = 'Johny', Gender = 'Female', DeptName = 'IT'   
where Id = 1  
  
**Update**() function used in the trigger, returns true, even if you update with the same value. For this reason, I recomend to compare values between inserted and deleted tables, rather than relying on Update() function. The Update() function does not operate on a per row basis, but across all rows.

**Instead Of Delete Triggers:**

**INSTEAD OF DELETE trigger**. An INSTEAD OF DELETE trigger gets fired instead of the DELETE event, on a table or a view. For example, let's say we have, an INSTEAD OF DELETE trigger on a view or a table, and then when you try to update a row from that view or table, instead of the actual DELETE event, the trigger gets fired automatically. INSTEAD OF DELETE TRIGGERS, are used, to delete records from a view, that is based on multiple tables.

**Now, let's try to delete a row from the view, and we get the same error.**  
Delete from vWEmployeeDetails where Id = 1  
  
**Script to create INSTEAD OF DELETE trigger:**  
Create Trigger tr\_vWEmployeeDetails\_InsteadOfDelete  
on vWEmployeeDetails  
instead of delete  
as  
Begin  
 Delete tblEmployee   
 from tblEmployee  
 join deleted  
 on tblEmployee.Id = deleted.Id  
   
 --Subquery  
 --Delete from tblEmployee   
 --where Id in (Select Id from deleted)  
End  
  
**Notice that, the trigger tr\_vWEmployeeDetails\_InsteadOfDelete**, makes use of DELETED table. DELETED table contains all the rows, that we tried to DELETE from the view. So, we are joining the DELETED table with tblEmployee, to delete the rows. You can also use sub-queries to do the same. In most cases JOINs are faster than SUB-QUERIEs. However, in cases, where you only need a subset of records from a table that you are joining with, sub-queries can be faster.  
  
**Upon executing the following DELETE statement**, the row gets DELETED as expected from tblEmployee table  
Delete from vWEmployeeDetails where Id = 1

|  |  |
| --- | --- |
| **Trigger** | **INSERTED or DELETED?** |
| Instead of Insert | DELETED table is always empty and the INSERTED table contains the newly inserted data. |
| Instead of Delete | INSERTED table is always empty and the DELETED table contains the rows deleted |
| Instead of Update | DELETED table contains OLD data (before update), and inserted table contains NEW data(Updated data) |

**What are DDL triggers**  
**DDL triggers fire in response to DDL events** - CREATE, ALTER, and DROP (Table, Function, Index, Stored Procedure etc...). For the list of all DDL events please visit https://msdn.microsoft.com/en-us/library/bb522542.aspx  
  
**Certain system stored procedures** that perform DDL-like operations can also fire DDL triggers.

Example - sp\_rename system stored procedure  
  
**What is the use of DDL triggers**

* If you want to execute some code in response to a specific DDL event
* To prevent certain changes to your database schema
* Audit the changes that the users are making to the database structure

**Syntax for creating DDL trigger**

CREATE TRIGGER [Trigger\_Name]

ON [Scope (Server|Database)]

FOR [EventType1, EventType2, EventType3, ...],

AS

BEGIN

   -- Trigger Body

END

**DDL triggers scope :** DDL triggers can be created in a specific database or at the server level.   
  
**The following trigger will fire in response to CREATE\_TABLE DDL event.**

CREATE TRIGGER trMyFirstTrigger

ON Database

FOR CREATE\_TABLE

AS

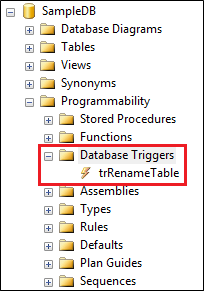
BEGIN

   Print 'New table created'

END

**To check if the trigger has been created**

1. In the Object Explorer window, expand the **SampleDB**database by clicking on the plus symbol.
2. Expand **Programmability**folder
3. Expand **Database Triggers** folder

  
  
**Please note :** If you can't find the trigger that you just created, make sure to refresh the Database Triggers folder.  
  
When you execute the following code to create the table, the trigger will automatically fire and will print the message - New table created  
Create Table Test (Id int)  
  
The above trigger will be fired only for one DDL event CREATE\_TABLE. If you want this trigger to be fired for multiple events, for example when you alter or drop a table, then separate the events using a comma as shown below.

ALTER TRIGGER trMyFirstTrigger

ON Database

FOR CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE

AS

BEGIN

   Print 'A table has just been created, modified or deleted'

END

Now if you create, alter or drop a table, the trigger will fire automatically and you will get the message - A table has just been created, modified or deleted.   
  
The 2 DDL triggers above execute some code in response to DDL events

Now let us look at an example of how to prevent users from creating, altering or dropping tables. To do this modify the trigger as shown below.

ALTER TRIGGER trMyFirstTrigger

ON Database

FOR CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE

AS

BEGIN

   Rollback

   Print 'You cannot create, alter or drop a table'

END

To be able to create, alter or drop a table, you either have to disable or delete the trigger.  
  
**To disable trigger**  
**1.** Right click on the trigger in object explorer and select **"Disable"**from the context menu   
**2.** You can also disable the trigger using the following T-SQL command

DISABLE TRIGGER trMyFirstTrigger ON DATABASE

**To enable trigger**  
**1.** Right click on the trigger in object explorer and select "Enable" from the context menu   
**2.** You can also enable the trigger using the following T-SQL command  
ENABLE TRIGGER trMyFirstTrigger ON DATABASE  
  
**To drop trigger**  
**1.** Right click on the trigger in object explorer and select "Delete" from the context menu   
**2.** You can also drop the trigger using the following T-SQL command  
DROP TRIGGER trMyFirstTrigger ON DATABASE  
  
Certain system stored procedures that perform DDL-like operations can also fire DDL triggers. The following trigger will be fired when ever you rename a database object using sp\_rename system stored procedure.

CREATE TRIGGER trRenameTable

ON DATABASE

FOR RENAME

AS

BEGIN

    Print 'You just renamed something'

END

The following code changes the name of the TestTable to NewTestTable. When this code is executed, it will fire the trigger trRenameTable:

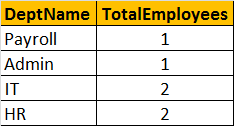
sp\_rename 'TestTable', 'NewTestTable'

The following code changes the name of the Id column in NewTestTable to NewId. When this code is executed, it will fire the trigger trRenameTable

sp\_rename 'NewTestTable.Id' , 'NewId', 'column'

### **Common Table Expressions**

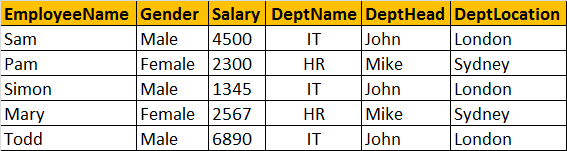
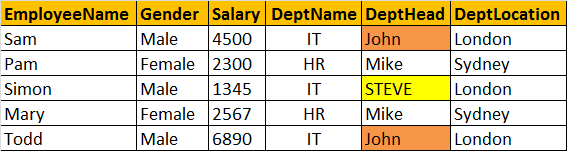
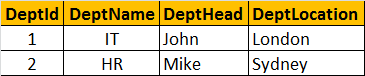
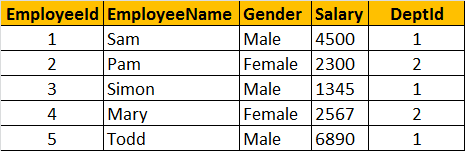
A **CTE** is a temporary result set, that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement, that immediately follows the **CTE.**

**Write a query using CTE,**to display the total number of Employees by Department Name. The output should be as shown below.  
  
  
**Before we write the query, let's look at the syntax for creating a CTE.**

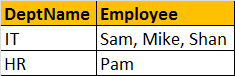
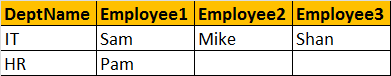
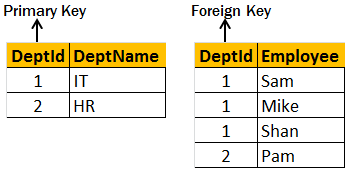
**WITH cte\_name (Column1, Column2,)**  
**AS**  
**( CTE\_query )**  
  
**SQL query using CTE:**  
With EmployeeCount(DepartmentId, TotalEmployees)  
as  
( Select DepartmentId, COUNT(\*) as TotalEmployees  
 from tblEmployee  
 group by DepartmentId)  
Select DeptName, TotalEmployees  
from tblDepartment  
join EmployeeCount  
on tblDepartment.DeptId = EmployeeCount.DepartmentId  
order by TotalEmployees  
  
**We define a CTE**, using **WITH** keyword, followed by the name of the CTE. In our example, **EmployeeCount**is the name of the CTE. Within parentheses, we specify the columns that make up the CTE. **DepartmentId** and **TotalEmployees** are the columns of **EmployeeCount** CTE. These 2 columns map to the columns returned by the **SELECT CTE query**. The CTE column names and CTE query column names can be different. Infact, CTE column names are optional. However, if you do specify, the number of **CTE columns** and the **CTE SELECT query** columns should be same. Otherwise you will get an error stating - 'EmployeeCount has fewer columns than were specified in the column list'. The column list, is followed by the as keyword, following which we have the CTE query within a pair of parentheses.  
  
**EmployeeCount CTE** is being joined with **tblDepartment** table, in the SELECT query, that immediately follows the CTE. Remember, a CTE can only be referenced by a SELECT, INSERT, UPDATE, or DELETE statement, **that immediately follows the CTE**. If you try to do something else in between, we get an error stating - 'Common table expression defined but not used'. The following SQL, raise an error.  
  
With EmployeeCount(DepartmentId, TotalEmployees)  
as  
( Select DepartmentId, COUNT(\*) as TotalEmployees  
 from tblEmployee  
 group by DepartmentId)  
  
Select 'Hello'  
  
Select DeptName, TotalEmployees  
from tblDepartment  
join EmployeeCount  
on tblDepartment.DeptId = EmployeeCount.DepartmentId  
order by TotalEmployees  
  
**It is also, possible to create multiple CTE's using a single WITH clause.**  
With EmployeesCountBy\_Payroll\_IT\_Dept(DepartmentName, Total)  
as  
(  
 Select DeptName, COUNT(Id) as TotalEmployees  
 from tblEmployee  
 join tblDepartment   
 on tblEmployee.DepartmentId = tblDepartment.DeptId  
 where DeptName IN ('Payroll','IT')  
 group by DeptName  
),  
EmployeesCountBy\_HR\_Admin\_Dept(DepartmentName, Total)  
as  
(  
 Select DeptName, COUNT(Id) as TotalEmployees  
 from tblEmployee  
 join tblDepartment   
 on tblEmployee.DepartmentId = tblDepartment.DeptId  
 group by DeptName   
)  
Select \* from EmployeesCountBy\_HR\_Admin\_Dept   
UNION  
Select \* from EmployeesCountBy\_Payroll\_IT\_Dept

### **Database Normalization**

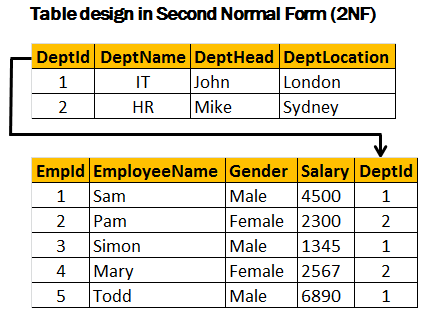
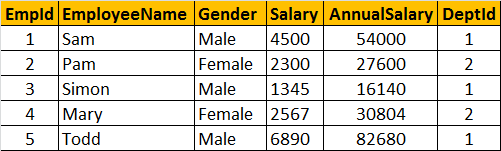
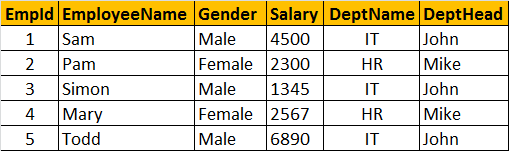
**What are the goals of database normalization**?  
or **Why do we normalize databases**?  
or **What is database normalization**?  
  
**Database normalization** is the process of organizing data to minimize data redundancy (data duplication), which in turn ensures data consistency.

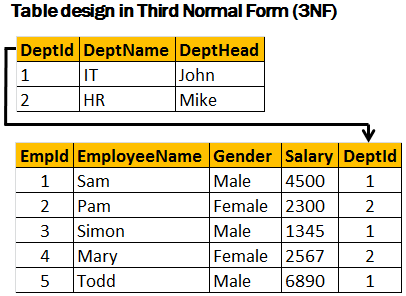
**Let's understand with an example**, how**redundant data**can cause **data inconsistency**. Consider **Employees**table below. For every employee with in the same department, we are repeating, all the 3 columns (DeptName, DeptHead and DeptLocation). Let's say for example, if there 50 thousand employees in the IT department, we would have unnecessarily repeated all the 3 department columns (DeptName, DeptHead and DeptLocation) data 50 thousand times. The obvious problem with redundant data is the disk space wastage.  
  
**Another common problem, is that data can become inconsistent.** For example, let's say, JOHN has resigned, and we have a new department head (STEVE) for IT department. At present, there are 3 IT department rows in the table, and we need to update all of them. Let's assume I updated only one row and forgot to update the other 2 rows, then obviously, the data becomes inconsistent.  
  
**Another problem**, DML queries (Insert, update and delete), **could become slow**, as there could many records and columns to process.  
  
**So, to reduce the data redundancy**, we can divide this large badly organised table into two (Employees and Departments), as shown below. Now, we have reduced redundant department data. So, if we have to update department head name, we only have one row to update, even if there are 10 million employees in that department.  
  
**Normalized Departments Table**  
  
  
**Normalized Employees Table**  


**Database normalization is a step by step process.** There are 6 normal forms, First Normal form (1NF) thru Sixth Normal Form (6NF). Most databases are in third normal form (3NF). There are certain rules, that each normal form should follow.  
  
**Now, let's explore the first normal form** (1NF). A table is said to be in 1NF, if

1. The data in each column should be **atomic**. No multiple values, separated by comma.  
2. The table does not contain any **repeating column groups**  
3. Identify each record **uniquely using primary key**.  
  
**In the table below, data in Employee column is not atomic**. It contains multiple employees separated by comma. From the data you can see that in the IT department, we have 3 employees - Sam, Mike, Shan. Now, let's say I want to change just, SHAN name.**It is not possible; we have to update the entire cell.** Similarly, it is not possible to select or delete just one employee, as the data in the cell is not atomic.  
  
  
**The 2nd rule of the first normal form is that, the table should not contain any repeating column groups**. Consider the Employee table below. We have repeated the Employee column, from Employee1 to Employee3. The problem with this design is that, if a department is going to have more than 3 employees, then we have to **change the table structure** to add Employee4 column. Employee2 and Employee3 columns in the HR department are NULL, as there is only employee in this department. The **disk space is simply wasted.**  
  
  
**To eliminate the repeating column groups, we are dividing the table into 2**. The repeating Employee columns are moved into a seperate table, with a foreign key pointing to the primary key of the other table. We also, introduced primary key to uniquely identify each record.  


**A table is said to be in 2NF, if**  
1. The table meets all the **conditions of 1NF**  
2. Move **redundant** data to a separate table  
3. Create **relationship** between these tables using foreign keys.

**Now, to put this table in the second normal form**, we need to break the table into 2, and move the redundant department data (**DeptName, DeptHead and DeptLocation**) into it's own table. To link the tables with each other, we use the **DeptId** foreign key. The tables below are in 2NF.  
  
  
**Third Normal Form (3NF):**  
**A table is said to be in 3NF, if the table**  
1. Meets all the conditions of **1NF and 2NF**  
2. Does not contain columns (attributes) that are not fully **dependent upon the primary key**  
  
**The table below, violates third normal form**, because **AnnualSalary** column is not fully dependent on the primary key **EmpId**. The **AnnualSalary** is also dependent on the **Salary** column. In fact, to compute the **AnnualSalary**, we multiply the **Salary** by **12**. Since **AnnualSalary** is not fully dependent on the primary key, and it can be computed, we can remove this column from the table, which then, will adhere to 3NF.  
  
  
**Let's look at another example of Third Normal Form violation**. In the table below, **DeptHead** column is not fully dependent on **EmpId** column. **DeptHead** is also dependent on **DeptName**. So, this table is not in **3NF**.  
  
  
**To put this table in 3NF, we break this down into 2**, and then move all the columns that are not fully dependent on the primary key to a separate table as shown below. This design is now in 3NF.



### **Pivot operator in SQL Server**

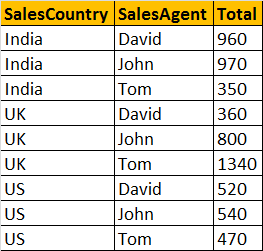
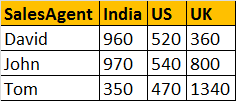
**Pivot is a SQL** **server operator**that can be used to turn **unique values from one column**, into **multiple columns**in the output, there by effectively **rotating a table**.

**Let's understand the power of PIVOT operator with an example:**

**Consider Sales Table-**

**Select \* from Sales**: As you can see, we have 3 sales agents selling in 3 countries

**Now, let's write a query which returns TOTAL SALES**, grouped by **SALESCOUNTRY**and **SALESAGENT**. The output should be as shown below.

  
**A simple GROUP BY query can produce this output.**  
Select SalesCountry, SalesAgent, SUM(SalesAmount) as Total  
from Sales group by SalesCountry, SalesAgent  
order by SalesCountry, SalesAgent  
  
**At, this point, let's try to present the same data in different format** using PIVOT operator.  
  
  
**Query using PIVOT operator:**  
Select SalesAgent, India, US, UK  
from Sales  
Pivot  
(  
   Sum(SalesAmount) for SalesCountry in ([India],[US],[UK])  
) as PivotTable

SELECT PIVOTED VALUES FROM TABLE

PIVOT

(

FOR PIVOTED COLUMN NAME IN (VALUES)

)

**This PIVOT query is converting the unique column values**(India, US, UK) in **SALESCOUNTRY** column, **into Columns**in the output, along with performing aggregations on the **SALESAMOUNT**column. The Outer query, simply, selects **SALESAGENT**column from **tblProductSales**table, along with pivoted columns from the PivotTable.  
  
**Having understood the basics of PIVOT**, let's look at another example. Let's create **SalesRev**, a slight variation of **Sales**, that we have already created. The table, that we are creating now, has got an additional**Id** column.

**Now, run the same PIVOT query**that we have already created, just by changing the name of the table to **tblProductsSale**instead of **tblProductSales**  
Select SalesAgent, India, US, UK  
from tblProductsSale  
Pivot  
(  
   Sum(SalesAmount) for SalesCountry in ([India],[US],[UK])  
)as PivotTable

**This output is not what we have expected.**

This is because of the presence of Id column in tblProductsSale, which is also considered when performing pivoting and group by. To eliminate this from the calculations, we have used derived table, which only selects, SALESAGENT, SALESCOUNTRY, and SALESAMOUNT. The rest of the query is very similar to what we have already seen.  
Select SalesAgent, India, US, UK  
from  
(  
   Select SalesAgent, SalesCountry, SalesAmount from tblProductsSale  
) as SourceTable  
Pivot  
(  
 Sum(SalesAmount) for SalesCountry in (India, US, UK)  
) as PivotTable

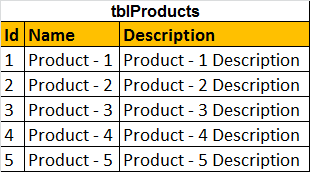
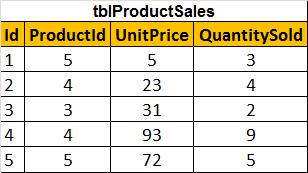
**UNPIVOT** performs the opposite operation to PIVOT by rotating columns of a table-valued expression into column values.  
  
The syntax of PIVOT operator :

SELECT <non-pivoted column>,  
    [first pivoted column] AS <column name>,  
    [second pivoted column] AS <column name>,  
    ...  
    [last pivoted column] AS <column name>  
FROM  
    (<SELECT query that produces the data>)  
    AS <alias for the source query>  
PIVOT  
(  
    <aggregation function>(<column being aggregated>)  
FOR  
    [<column that contains the values that will become column headers>]  
    IN ( [first pivoted column], [second pivoted column], ... [last pivoted column])  
)  
AS <alias for the pivot table>  
<optional ORDER BY clause>;

### **Cursors in SQL server:**

**Relational Database Management Systems, including sql server are very good at handling data in SETS.** For example, the following "UPDATE" query, updates a set of rows that matches the condition in the "WHERE" clause at the same time.   
**Update tblProductSales1 Set UnitPrice = 50 where ProductId = 101**

**However, if there is ever a need to process the rows, on a row-by-row basis**, then cursors are your choice. Cursors are very bad for performance, and should be avoided always. Most of the time, cursors can be very easily replaced using joins.  
  
There are different types of cursors in sql server as listed below. We will talk about the differences between these cursor types in a later video session.   
**1.** Forward-Only  
**2.** Static  
**3.** Keyset  
**4.** Dynamic

**Let us now look at a simple example of using sql server cursor to process one row at time.** We will be using tblProducts1 and tblProductSales1 tables, for this example. The tables here show only 5 rows from each table. However, on my machine, there are 400,000 records in tblProducts1 and 600,000 records in tblProductSales1 tables. If you want to learn about generating huge amounts of random test data  
   
  
**Cursor Example:** Let us say, I want to update the UNITPRICE column in **tblProductSales1** table, based on the following criteria  
**1.** If the ProductName = 'Product - 55', Set Unit Price to 55  
**2.** If the ProductName = 'Product - 65', Set Unit Price to 65  
**3.** If the ProductName is like 'Product - 100%', Set Unit Price to 1000  
  
Declare @ProductId int  
-- Declare the cursor using the declare keyword  
Declare ProductIdCursor CURSOR FOR   
Select ProductId from tblProductSales1  
-- Open statement, executes the SELECT statment  
-- and populates the result set  
Open ProductIdCursor  
-- Fetch the row from the result set into the variable  
Fetch Next from ProductIdCursor into @ProductId  
-- If the result set still has rows, @@FETCH\_STATUS will be ZERO  
While(@@FETCH\_STATUS = 0)  
Begin  
 Declare @ProductName nvarchar(50)  
 Select @ProductName = Name from tblProducts1 where Id = @ProductId  
   
 if(@ProductName = 'Product - 55')  
 Begin  
 Update tblProductSales1 set UnitPrice = 55 where ProductId = @ProductId  
 End  
 else if(@ProductName = 'Product - 65')  
 Begin  
 Update tblProductSales1 set UnitPrice = 65 where ProductId = @ProductId  
 End  
 else if(@ProductName like 'Product - 100%')  
 Begin  
 Update tblProductSales1 set UnitPrice = 1000 where ProductId = @ProductId  
 End  
   
 Fetch Next from ProductIdCursor into @ProductId   
End  
-- Release the row set  
CLOSE ProductIdCursor   
-- Deallocate, the resources associated with the cursor  
DEALLOCATE ProductIdCursor  
  
The cursor will loop thru each row in tblProductSales1 table. As there are 600,000 rows, to be processed on a row-by-row basis, it takes around 40 to 45 seconds on my machine. We can achieve this very easily using a join, and this will significantly increase the performance. We will discuss about this in our next video session.  
  
**To check if the rows have been correctly updated, please use the following query.**

Select  Name, UnitPrice   
from tblProducts1 join  
tblProductSales1 on tblProducts1.Id = tblProductSales1.ProductId  
where (Name='Product - 55' or Name='Product - 65' or Name like 'Product - 100%')

**While Loop In SQL**

The WHILE statement is a control-flow statement that allows you to execute a statement block repeatedly as long as a specified condition is TRUE.

The following illustrates the syntax of the WHILE statement:

WHILE Boolean\_expression

{ sql\_statement | statement\_block}

First, the Boolean\_expression is an expression that evaluates to TRUE or FALSE.

Second, sql\_statement | statement\_block is any Transact-SQL statement or a set of Transact-SQL statements. A statement block is defined using the [BEGIN...END](https://www.sqlservertutorial.net/sql-server-stored-procedures/sql-server-begin-end/) statement.

If the Boolean\_expression evaluates to FALSE when entering the loop, no statement inside the WHILE loop will be executed.

Inside the WHILE loop, you must change some [variables](https://www.sqlservertutorial.net/sql-server-stored-procedures/variables/) to make the Boolean\_expression returns FALSE at some points. Otherwise, you will have an indefinite loop.

Note that if the Boolean\_expression contains a [SELECT](https://www.sqlservertutorial.net/sql-server-basics/sql-server-select/) statement, it must be enclosed in parentheses.

To exit the current iteration of the loop immediately, you use the [BREAK](https://www.sqlservertutorial.net/sql-server-stored-procedures/sql-server-break/) statement. To skip the current iteration of the loop and start the new one, you use the [CONTINUE](https://www.sqlservertutorial.net/sql-server-stored-procedures/sql-server-continue/) statement.

**SQL Server WHILE example**

Let’s take an example of using the SQL Server WHILE statement to understand it better.

The following example illustrates how to use the WHILE statement to print out numbers from 1 to 5:

DECLARE @counter INT = 1;

WHILE @counter <= 5

BEGIN

PRINT @counter;

SET @counter = @counter + 1;

END

CREATE Table tblAuthors

(

Id int identity primary key,

Author\_name nvarchar(50),

country nvarchar(50)

)

Declare @Id int

Set @Id = 1

While @Id <= 12000

Begin

Insert Into tblAuthors values ('Author - ' + CAST(@Id as nvarchar(10)),

'Country - ' + CAST(@Id as nvarchar(10)) + ' name')

Print @Id

Set @Id = @Id + 1

End

select \* from tblAuthors

# **Types of Transactions in SQL Server**

What are Transactions / Transaction Management??

The process of combining a set of related operations into a single unit and executing those operations by applying to do everything or do-nothing principle is called transaction management. For example, the transfer money task is the combination of two operations

1. **Withdraw money from the Senders account**
2. **Deposit Money into the Receivers account.**

We need to execute these two operations by applying the “do everything or nothing principle” which is nothing but performing the transaction management. So, every transaction has two boundaries

##### **Types of Transactions in SQL Server:**

The SQL Server Transactions are classified into three types, they are as follows

1. **Auto Commit Transaction Mode (default)**
2. **Implicit Transaction Mode**
3. **Explicit Transaction Mode**

Lets create below table for understanding each of the type:

CREATE TABLE Customers

(

CustomerID INT PRIMARY KEY,

CustomerCode VARCHAR(10),

CustomerName VARCHAR(50)

)

##### **Auto Commit Transaction Mode in SQL Server:**

This is the default transaction mode in SQL Server. In this transaction mode, each SQL statement is treated as a separate transaction. In this Transaction Mode, as a developer, we are not responsible for either beginning the transaction (i.e. Begin Transaction) or ending a transaction (i.e. either Commit or Roll Back). Whenever we execute any DML statement, the SQL Server will automatically begin the transaction as well as end the transaction with a Commit or Rollback i.e. if the transaction is completed successfully, it is committed. If the transaction faces any error, it is rolled back. So, the programmer does not have any control over them.

Let us understand Auto Commit Transaction Mode with some examples. Please execute the below insert statement.  
INSERT INTO Customer VALUES (1, ‘CODE\_1’, ‘David’)

When you execute the above statement, the SQL Server will automatically begin the transaction and end the

transaction with commit. Now, try to execute the below Insert query.

**INSERT INTO Customer VALUES (1, ‘CODE\_2’, ‘John’)**

When you try to execute the above Insert Query, the insert failed as we are trying to insert a duplicate value into the primary key table, so the SQL Server will automatically begin the transaction and end the transaction with a Rollback. And when you execute the query you will get the below Primary Key Violation error.

##### **Implicit Transaction Mode in SQL Server:**

In the Implicit mode of transaction, the SQL Server is responsible for beginning the transaction implicitly before the execution of any DML statement and the developers are responsible to end the transaction with a commit or rollback. Once the transaction is ended ie. once the developer executes either the commit or rollback command, then automatically a new transaction will start by SQL Server. That means, in the case of implicit mode, a new transaction will start automatically by SQL Server after the current transaction is committed or rolled back by the programmer.

In order to use implicit transaction mode in SQL Server, first, we need to set the implicit transaction mode to **ON** using the **SET IMPLICIT\_TRANSACTIONS** statement.

The value for **IMPLICIT\_TRANSACTIONS can be ON or OFF**. When the value for implicit transaction mode is set to ON then a new transaction is automatically started by SQL Server whenever we execute any SQL statement (Insert, Select, Delete, and Update).

**Examples to understand Implicit Mode of Transactions in SQL Server:**

Before going to do anything, first, DELETE all the data from the Customer table by executing the below DELETE statement.

**DELETE FROM Customer;**

**Step1:** Set the Implicit transaction mode to ON

**SET IMPLICIT\_TRANSACTIONS ON**

**Step2:** Execute the DML Statement

Now let us try to insert two records using the implicit mode of transaction.

**INSERT INTO Customer VALUES (1, ‘CODE\_1’, ‘David’);**

**INSERT INTO Customer VALUES (2, ‘CODE\_2’, ‘John’);**

**Step3:** Commit the transaction

**COMMIT TRANSACTION**

When you execute the Commit Transaction statement, then data is saved permanently into the database, and after that, a new transaction will automatically be started by SQL Server.

**Step4:** Now execute the following DML Statements

**INSERT INTO Customer VALUES (3, ‘CODE\_3’, ‘Pam’);**

**UPDATE Customer SET CustomerName = ‘John Changed’ WHERE CustomerID = 2;**

**SELECT \* FROM Customer;**

When you execute the above statement and you will get the below data.

**Step5:** Rollback the transaction

As of now, we have not either committed or rollback the transaction, so let roll back the transaction and see the table data.

**ROLLBACK TRANSACTION**

Once you roll back the transaction, issue a select query against the customer table and you will see the following data.

**Note:** If you don’t want to use implicit transaction mode, then you can turn it off by executing the below statement.

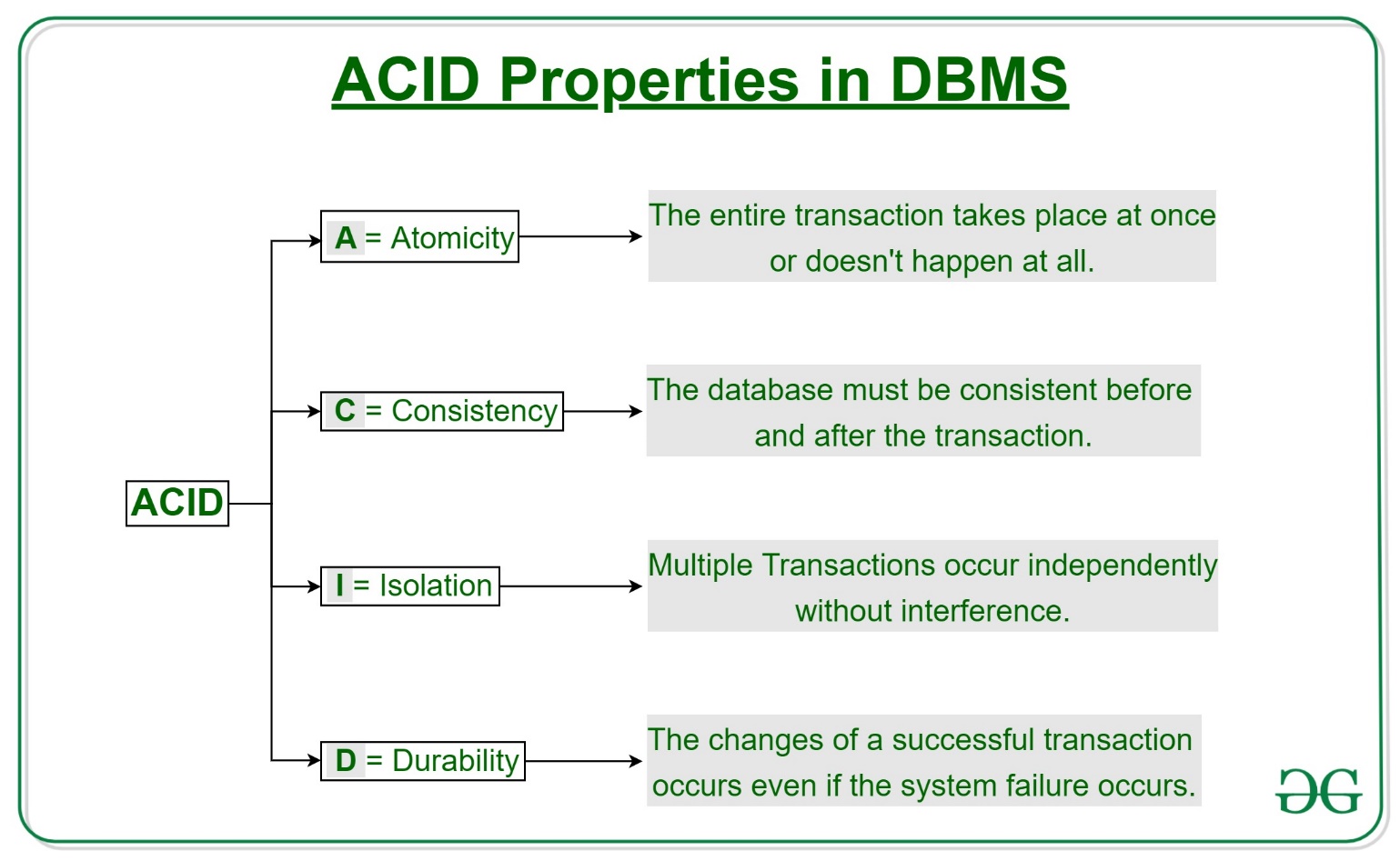
**SET IMPLICIT\_TRANSACTIONS OFF**

##### **Explicit Transaction Mode in SQL Server:**

In the Explicit mode of transaction, the developer is only responsible for beginning the transaction as well as ending the transaction. In other words, we can say that the transactions that have a START and END explicitly written by the programmer are called explicit transactions.

Here, every transaction should start with the **BEGIN TRANSACTION** statement and ends with either a **ROLLBACK TRANSACTION** statement (when the transaction does not complete successfully) or a **COMMIT TRANSACTION** statement (when the transaction completes successfully). The Explicit Transaction Mode is most commonly used in triggers, stored procedures, and application programs.

A [**transaction**](https://www.geeksforgeeks.org/sql-transactions/) is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.   
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.



**Atomicity:**

By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.   
—**Abort**: If a transaction aborts, changes made to the database are not visible.   
—**Commit**: If a transaction commits, changes made are visible.   
Atomicity is also known as the ‘All or nothing rule’.

**Consistency:**

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database.

**Isolation:**

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

**Durability:**

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

# **All about locking in SQL Server**

Locking is essential to successful SQL Server transactions processing and it is designed to allow SQL Server to work seamlessly in a multi-user environment. Locking is the way that SQL Server manages transaction concurrency. To understand better the locking in SQL Server, it is important to understand that locking is designed to ensure the integrity of the data in the database, as it forces every SQL Server transaction to pass the ACID test.

While objects are locked, SQL Server will prevent other transactions from making any change of data stored in objects affected by the imposed lock. Once the lock is released by committing the changes or by rolling back changes to initial state, other transactions will be allowed to make required data changes.

Translated into the SQL Server language, this means that when a transaction imposes the lock on an object, all other transactions that require the access to that object will be forced to wait until the lock is released and that wait will be registered with the adequate wait type

**Lock modes**

Lock mode considers various lock types that can be applied to a resource that has to be locked:

Exclusive (X)

Shared (S)

Update (U)

Intent (I)

Schema (Sch)

Bulk update (BU)

**Exclusive lock (X)** – This lock type, when imposed, will ensure that a page or row will be reserved exclusively for the transaction that imposed the exclusive lock, as long as the transaction holds the lock.

The exclusive lock will be imposed by the transaction when it wants to modify the page or row data, which is in the case of DML statements DELETE, INSERT and UPDATE. An exclusive lock can be imposed to a page or row only if there is no other shared or exclusive lock imposed already on the target. This practically means that only one exclusive lock can be imposed to a page or row, and once imposed no other lock can be imposed on locked resources

**Shared lock (S)**– this lock type, when imposed, will reserve a page or row to be available only for reading, which means that any other transaction will be prevented to modify the locked record as long as the lock is active. However, a shared lock can be imposed by several transactions at the same time over the same page or row and in that way several transactions can share the ability for data reading since the reading process itself will not affect anyhow the actual page or row data. In addition, a shared lock will allow write operations, but no DDL changes will be allowed

**Update lock (U)** – this lock is similar to an exclusive lock but is designed to be more flexible in a way. An update lock can be imposed on a record that already has a shared lock. In such a case, the update lock will impose another shared lock on the target row. Once the transaction that holds the update lock is ready to change the data, the update lock (U) will be transformed to an exclusive lock (X). It is important to understand that update lock is asymmetrical in regards of shared locks. While the update lock can be imposed on a record that has the shared lock, the shared lock cannot be imposed on the record that already has the update lock

**Intent locks (I)** – this lock is a means used by a transaction to inform another transaction about its intention to acquire a lock. The purpose of such lock is to ensure data modification to be executed properly by preventing another transaction to acquire a lock on the next in hierarchy object. In practice, when a transaction wants to acquire a lock on the row, it will acquire an intent lock on a table, which is a higher hierarchy object. By acquiring the intent lock, the transaction will not allow other transactions to acquire the exclusive lock on that table (otherwise, exclusive lock imposed by some other transaction would cancel the row lock).

This is an important lock type from the performance aspect as the SQL Server database engine will inspect intent locks only at the table level to check if it is possible for transaction to acquire a lock in a safe manner in that table, and therefore intent lock eliminates need to inspect each row/page lock in a table to make sure that transaction can acquire lock on entire table

There are three regular intent locks and three so-called conversion locks:

Regular intent locks:

**Intent exclusive (IX)** – when an intent exclusive lock (IX) is acquired it indicates to SQL Server that the transaction has the intention to modify some of lower hierarchy resources by acquiring exclusive (X) locks individually on those lower hierarchy resources

**Intent shared (IS)** – when an intent shared lock (IS) is acquired it indicates to SQL Server that the transaction has the intention to read some lower hierarchy resources by acquiring shared locks (S) individually on those resources lower in the hierarchy

**Intent update (IU)** – when an intent shared lock (IS) is acquired it indicates to SQL Server that the transaction has the intention to read some of lower hierarchy resources by acquiring shared locks (S) individually on those resources lower in the hierarchy. The intent update lock (IU) can be acquired only at the page level and as soon as the update operation takes place, it converts to the intent exclusive lock (IX)

Conversion locks:

**Shared with intent exclusive (SIX)** – when acquired, this lock indicates that the transaction intends to read all resources at a lower hierarchy and thus acquire the shared lock on all resources that are lower in hierarchy, and in turn, to modify part of those, but not all. In doing so, it will acquire an intent exclusive (IX) lock on those lower hierarchy resources that should be modified. In practice, this means that once the transaction acquires a SIX lock on the table, it will acquire intent exclusive lock (IX) on the modified pages and exclusive lock (X) on the modified rows.

Only one shared with intent exclusive lock (SIX) can be acquired on a table at a time and it will block other transactions from making updates, but it will not prevent other transactions to read the lower hierarchy resources they can acquire the intent shared (IS) lock on the table

**Shared with intent update (SIU)** – this is a bit more specific lock as it is a combination of the shared (S) and intent update (IU) locks. A typical example of this lock is when a transaction is using a query executed with the PAGELOCK hint and query, then the update query. After the transaction acquires an SIU lock on the table, the query with the PAGELOCK hint will acquire the shared (S) lock while the update query will acquire intent update (IU) lock

**Update with intent exclusive (UIX)** – when update lock (U) and intent exclusive (IX) locks are acquired at lower hierarchy resources in the table simultaneously, the update with intent exclusive lock will be acquired at the table level as a consequence

**Schema locks (Sch)** – The SQL Server database engine recognizes two types of the schema locks: **Schema modification lock (Sch-M)** and **Schema stability lock (Sch-S)**

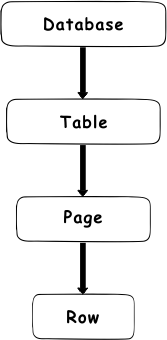
A **Schema modification lock (Sch-M)** will be acquired when a DDL statement is executed, and it will prevent access to the locked object data as the structure of the object is being changed. SQL Server allows a single schema modification lock (Sch-M) lock on any locked object. In order to modify a table, a transaction must wait to acquire a Sch-M lock on the target object. Once it acquires the schema modification lock (Sch-M), the transaction can modify the object and after the modification is completed and the lock will be released. A typical example of the Sch-M lock is an index rebuild, as an index rebuild is table modification process. Once the index rebuild ID is issued, a schema modification lock (Sch-M) will be acquired on that table and will be released only after the index rebuild process is completed (when used with ONLINE option, index rebuild will acquire Sch-M lock shortly at the end of the process)

A **Schema stability lock (Sch-S)** will be acquired while a schema-dependent query is being compiled and executed and execution plan is generated. This particular lock will not block other transactions to access the object data and it is compatible with all lock modes except with the schema modification lock (Sch-M). Essentially, Schema stability locks will be acquired by every DML and select query to ensure the integrity of the table structure (ensure that table doesn’t change while queries are running).

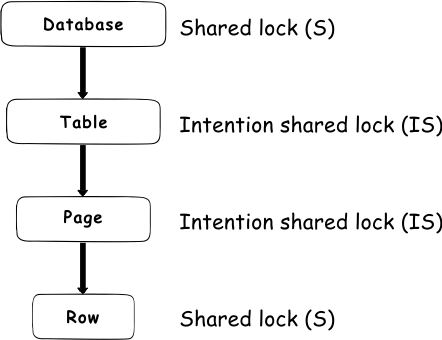
**Bulk Update locks (BU)** – this lock is designed to be used by bulk import operations when issued with a TABLOCK argument/hint. When a bulk update lock is acquired, other processes will not be able to access a table during the bulk load execution. However, a bulk update lock will not prevent another bulk load to be processed in parallel. But keep in mind that using TABLOCK on a clustered index table will not allow parallel bulk importing.

**Locking hierarchy**

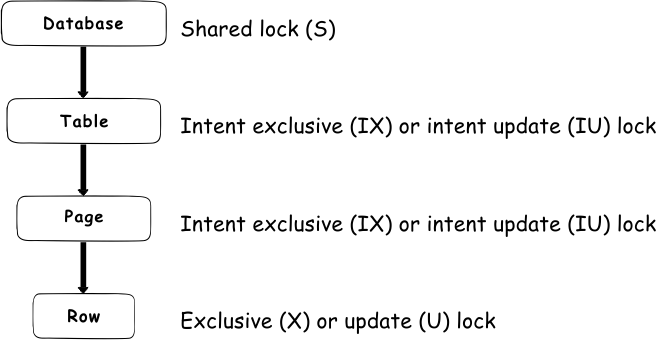
SQL Server has introduced the locking hierarchy that is applied when reading or changing of data is performed. The lock hierarchy starts with the database at the highest hierarchy level and down via table and page to the row at the lowest level



Essentially, there is always a shared lock on the database level that is imposed whenever a transaction is connected to a database. The shared lock on a database level is imposed to prevent dropping of the database or restoring a database backup over the database in use. For example, when a SELECT statement is issued to read some data, a shared lock (S) will be imposed on the database level, an intent shared lock (IS) will be imposed on the table and on the page level, and a shared lock (S) on the row itself



In case of a DML statement (i.e. insert, update, delete) a shared lock (S) will be imposed on the database level, an intent exclusive lock (IX) or intent update lock (IU) will be imposed on the table and on the page level, and an exclusive or update lock (X or U) on the row



**Deadlock definition in SQL Server**

In terms of SQL Server, a deadlock occurs when two (or more) processes lock the separate resource. Under these circumstances, each process cannot continue and begins to wait for others to release the resource. However, the SQL engine understands that this contention would never end with the help of the lock manager warning and then it decides to kill one process to solve this conflict problem so that the other process can be completed. The killed process is called the deadlock victim.

DROP TABLE IF EXISTS Table\_A

  CREATE TABLE Table\_A (Id INT PRIMARY KEY, FruitName VARCHAR(100))

  GO

  INSERT INTO Table\_A VALUES(1,'Lemon')

  INSERT INTO Table\_A VALUES(2,'Apple')

  GO

  DROP TABLE IF EXISTS Table\_B

  CREATE TABLE Table\_B (Id INT PRIMARY KEY, FruitName VARCHAR(100))

  GO

  INSERT INTO  Table\_B VALUES(1,'Banana')

  INSERT INTO Table\_B VALUES(2,'Orange')

GO

As we explained in the deadlock definition, we need at least two processes for the deadlock, so that we will execute the following queries at the same time in the separated query windows.

  BEGIN TRAN

  UPDATE Table\_A SET FruitName ='Mango' WHERE Id=1

  --WAITFOR DELAY '00:00:08'

  UPDATE Table\_B SET FruitName ='Avacado' WHERE Id=1

  COMMIT TRAN

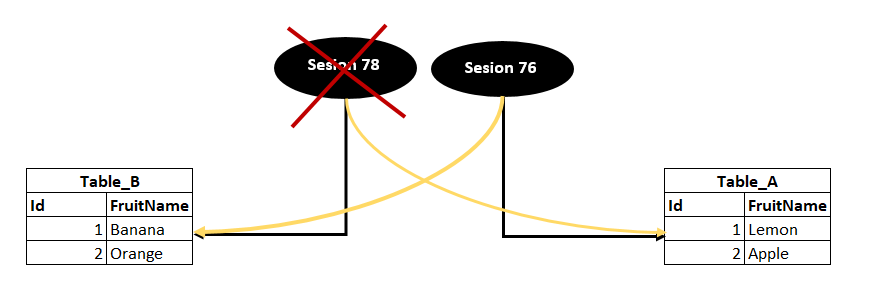
  BEGIN TRAN

  UPDATE Table\_B SET FruitName ='Papaya' WHERE Id=1

  --WAITFOR DELAY '00:00:08'

  UPDATE Table\_A SET FruitName ='Kiwi' WHERE Id=1

COMMIT TRAN



SQL chooses the victim according to the cost of the rollback. It means that the victim of the process has been decided based on the minimum resource consumption.

### **SQL Server deadlock analysis and prevention**

No we will discuss **how to read and analyse sql server deadlock information captured in the error log**, so we can understand what's causing the deadlocks and take appropriate actions to prevent or minimize the occurrence of deadlocks.

**The deadlock information in the error log has three sections**

|  |  |
| --- | --- |
| **Section** | **Description** |
| Deadlock Victim | Contains the ID of the process that was selected as the deadlock victim and killed by SQL Server. |
| Process List | Contains the list of the processes that participated in the deadlock. |
| Resource List | Contains the list of the resources (database objects) owned by the processes involved in the deadlock |

**Process List :** The process list has lot of items. Here are some of them that are particularly useful in understanding what caused the deadlock.

|  |  |
| --- | --- |
| **Node** | **Description** |
| loginname | The loginname associated with the process |
| isolationlevel | What isolation level is used |
| procname | The stored procedure name |
| Inputbuf | The code the process is executing when the deadlock occured |

**Resource List :** Some of the items in the resource list that are particularly useful in understanding what caused the deadlock.

|  |  |
| --- | --- |
| **Node** | **Description** |
| objectname | Fully qualified name of the resource involved in the deadlock |
| owner-list | Contains (owner id) the id of the owning process and the lock mode it has acquired on the resource. lock mode determines how the resource can be accessed by concurrent transactions. S for Shared lock, U for Update lock, X for Exclusive lock etc |
| waiter-list | Contains (waiter id) the id of the process that wants to acquire a lock on the resource and the lock mode it is requesting |

To prevent the deadlock that we have in our case, we need to ensure that database objects (Table A & Table B) are accessed in the same order every time.

### Capturing Deadlocks:

1. Using SQL Profiler
2. Using Extended Events

On the **Select Events To Capture** screen, we will add the following events from the **Event library** to **Selected events**list.

* database\_xml\_deadlock\_report
* lock\_deadlock
* lock\_deadlock\_chain
* scheduler\_monitor\_deadlocks\_ring\_buffer\_recorded
* xml\_deadlock\_report
* xml\_deadlock\_report\_filtered

On the **Capture Global Fields**screen, we will select global events that will be captured with the events:

* client app name
* client connection id
* client hostname
* database id
* database name
* nt username
* sql text
* username

When we click the **xml\_report** field on the **xml\_deadlock\_report**event, the XML report of the deadlock will be opened. This report can be very helpful in understanding the details of the deadlock.

Also, in the **xml\_deadlock\_report** event, we can see the [deadlock graph](https://www.sqlshack.com/understanding-graphical-representation-sql-server-deadlock-graph/) and it offers a virtual representation of the deadlock.

**Preventing Deadlocks in SQL Server**

There is no exact and clear resolving formula for the deadlocks because all deadlocks might have unique characteristics. However, it is significant to understand the circumstances and the situation under which these deadlocks have occurred because this approach will broadly help to resolve them. After then, the following solution recommendations might help.

* Access the resources in the same order
* Write the shortest transactions as much as possible and lock the resource for the minimum duration
* Limiting the usage of the cursors
* Design more normalized databases
* Avoid poorly-optimized queries

**Conclusion**

Resolving the deadlock can be more complicated and struggling, so as a first step, we should clearly understand the deadlock definition and then set to work capturing and handling the deadlocks. In the final step, we can work on preventing or minimizing the deadlock.

### **Cross apply and Outer apply in SQL server**

Consider below two tables for understanding these concepts:

SELECT \* FROM Department

SELECT \* FROM Employee

Now to fetch all the matching records we will use Inner join as below:

Select D. Dept\_name, E.Name, E.Gender, E.Salary

from Department D

Inner Join Employee E

On D. Dept\_Id  = E.DepartmentId

Now if we want to retrieve all the matching rows between **Department**and **Employee**tables + the non-matching rows from the LEFT table (**Department**)

This can be very easily achieved using a Left Join as shown below.

Select D. Dept\_name, E.Name, E.Gender, E.Salary

from Department D

Left Join Employee E

On D.Dept\_Id = E.DepartmentId

Now let's assume we do not have access to the Employee table. Instead, we have access to the following Table Valued function, that returns all employees belonging to a department-by-Department Id.

fn\_GetEmployeesByDepartmentId

Create function fn\_GetEmployeesByDepartmentId(@DepartmentId int)

Returns Table

as

Return

(

    Select Id, Name, Gender, Salary, DepartmentId

    from Employee where DepartmentId = @DepartmentId

)

Go

The following query returns the employees of the department with Id =1.

Select \* from fn\_GetEmployeesByDepartmentId(1)

Now if you try to perform an Inner or Left join between **Department**table and **fn\_GetEmployeesByDepartmentId**() function you will get an error.

Select D.Dept\_name, E.Name, E.Gender, E.Salary

from Department D

Inner Join fn\_GetEmployeesByDepartmentId(D.Dept\_id) E

On D.Dept\_id = E.DepartmentId

If you execute the above query you will get the error

This is where we use **Cross Apply** and **Outer Apply** operators. **Cross Apply** is semantically equivalent to **Inner Join**and **Outer Apply**is semantically equivalent to **Left Outer Join**.

Just like Inner Join, Cross Apply retrieves only the matching rows from the Department table and fn\_GetEmployeesByDepartmentId() table valued function.

Select D.Dept\_name, E.Name, E.Gender, E.Salary

from Department D

Cross Apply fn\_GetEmployeesByDepartmentId(D.Dept\_id) E

Just like Left Outer Join, Outer Apply retrieves all matching rows from the Department table and fn\_GetEmployeesByDepartmentId() table valued function + non-matching rows from the left table (Department)

Select D.Dept\_name, E.Name, E.Gender, E.Salary

from Department D

Outer Apply fn\_GetEmployeesByDepartmentId(D.Dept\_id) E

**How does Cross Apply and Outer Apply work**

* The APPLY operator introduced in SQL Server 2005, is used to join a table to a table-valued function.
* The Table Valued Function on the right-hand side of the APPLY operator gets called for each row from the left (also called outer table) table.
* Cross Apply returns only matching rows (semantically equivalent to Inner Join)
* Outer Apply returns matching + non-matching rows (semantically equivalent to Left Outer Join). The unmatched columns of the table valued function will be set to NULL.

**Query Optimization and Performance Improvement Techniques:**

Interview Question: What are the query optimization or performance improvement techniques?

### 1. **Use Indexes Effectively**

Indexes speed up data retrieval by allowing SQL Server to quickly locate rows without scanning the entire table. Indexing columns involved in WHERE, JOIN, and ORDER BY clauses can significantly improve performance.

### 2. **Avoid SELECT \* (Select Only Required Columns)**

Retrieving all columns using SELECT \* increases the amount of data transferred and processed, even if all columns are not needed. Always select only the necessary columns.

### 3. **Use EXISTS Instead of IN for Sub queries**

EXISTS can be more efficient than IN, especially when dealing with subqueries that return large result sets. EXISTS stops searching as soon as it finds a match, while IN processes the entire subquery result set.

-- Less efficient use of IN

SELECT CustomerID, CustomerName

FROM Customers

WHERE CustomerID IN (SELECT CustomerID FROM Orders WHERE OrderDate = '2023-09-01');

-- More efficient use of EXISTS

SELECT CustomerID, CustomerName

FROM Customers

WHERE EXISTS (SELECT 1 FROM Orders WHERE Customers.CustomerID = Orders.CustomerID AND OrderDate = '2023-09-01');

### 4. **Use JOINs Instead of Subqueries**

JOINs are generally more efficient than subqueries because they allow SQL Server to process relationships between tables directly rather than running separate queries.

-- Less efficient use of subquery

SELECT CustomerID, CustomerName

FROM Customers

WHERE CustomerID = (SELECT CustomerID FROM Orders WHERE OrderDate = '2023-09-01');

-- More efficient JOIN usage

SELECT C.CustomerID, C.CustomerName

FROM Customers C

INNER JOIN Orders O ON C.CustomerID = O.CustomerID

WHERE O.OrderDate = '2023-09-01';

### 5. **Use Indexed Views for Complex Aggregations**

Indexed views store precomputed aggregations or joins, which speeds up queries that use complex calculations repeatedly.

-- Creating an aggregation view without an index

CREATE VIEW OrderSummary AS

SELECT CustomerID, COUNT(OrderID) AS TotalOrders, SUM(TotalAmount) AS TotalSales

FROM Orders

GROUP BY CustomerID;

Optimization

-- Create an indexed view to precompute the aggregation

CREATE VIEW OrderSummary WITH SCHEMABINDING AS

SELECT CustomerID, COUNT\_BIG(\*) AS TotalOrders, SUM(TotalAmount) AS TotalSales

FROM dbo.Orders

GROUP BY CustomerID;

-- Create an index on the view for better performance

CREATE UNIQUE CLUSTERED INDEX IX\_OrderSummary ON OrderSummary(CustomerID);

### 6. **Use Proper Data Types**

Using appropriate data types for columns reduces unnecessary conversions and storage, improving performance.

### 7. **Use Temporary Tables, CTE’s and Table Variables Appropriately**

For intermediate results, use temporary tables or table variables wisely. Temporary tables are better for larger datasets, while table variables are efficient for smaller sets of data.

Few More:

### 1. **Partitioning Large Tables**

**Technique**: Partitioning divides a large table into smaller, more manageable pieces based on a specified column (often a date). This helps reduce the amount of data scanned and improves query performance, especially for large datasets.

### 2. **Optimize Joins and Join Order**

SQL Server performs better when smaller result sets are joined first. This reduces the number of rows in intermediate results and reduces processing time.

**Optimization**: Arrange tables in such a way that smaller or more selective tables are joined first.

-- Less optimized join (larger table joined first)

SELECT \*

FROM LargeTable A

INNER JOIN SmallTable B ON A.ID = B.ID;

-- Optimized join (smaller table joined first)

SELECT \*

FROM SmallTable B

INNER JOIN LargeTable A ON A.ID = B.ID;

### 3. **Reduce Locking and Blocking (Use NOLOCK)**

SQL Server uses locks to maintain data consistency. However, this can cause blocking issues. Using the WITH (NOLOCK) hint (also known as a **read uncommitted** isolation level) can prevent locks from being acquired when only reading data.

-- Query using NOLOCK to prevent locking

SELECT \* FROM Orders WITH (NOLOCK)

WHERE OrderDate = '2023-09-01';

**Caution**: NOLOCK can return dirty (uncommitted) data, so use it carefully.

### 4. **Optimize Network Traffic (Batch Queries)**

Reduce the number of round trips between the client application and SQL Server by batching multiple queries or operations into a single request.

-- Instead of sending multiple individual insert statements:

INSERT INTO Orders (OrderID, OrderDate, TotalAmount) VALUES (1, '2023-09-01', 100.00);

INSERT INTO Orders (OrderID, OrderDate, TotalAmount) VALUES (2, '2023-09-02', 200.00);

-- Send them in a single batch:

INSERT INTO Orders (OrderID, OrderDate, TotalAmount)

VALUES (1, '2023-09-01', 100.00),

(2, '2023-09-02', 200.00);

**SQL Injection:**

**SQL Injection** is a type of security vulnerability that occurs when an attacker is able to insert or manipulate SQL queries sent to the database via an application's input fields. It allows attackers to execute arbitrary SQL code, potentially accessing, modifying, or deleting data in the database without proper authorization.

In a typical SQL injection attack, an attacker provides malicious input that is then incorporated into an SQL query without proper validation or sanitization. This causes the SQL query to execute unintended commands.

Consider a simple example of an online login form where users enter their username and password. The application executes an SQL query to check if the provided credentials are valid.

Here’s an example of vulnerable code:

-- Vulnerable SQL query (user input directly concatenated)

SELECT \* FROM Users

WHERE Username = 'user\_input'

AND Password = 'password\_input';

If the application accepts user inputs directly without any validation or escaping, an attacker can manipulate the inputs.

Example: An attacker enters the following values in the **username** field:

' OR 1=1 –

Exaplaination:

' closes the Username string.

OR 1=1 always evaluates to **true**, meaning the condition will always be met, regardless of the password.

-- comments out the rest of the SQL query, so the Password condition is ignored.

SELECT \* FROM Users

WHERE Username = '' OR 1=1 -- '

AND Password = 'password\_input';

This query will return all rows in the Users table because 1=1 is always true. The attacker gains access without needing a valid username or password.

CREATE TABLE Users (

UserID INT,

Username NVARCHAR(50),

Password NVARCHAR(50)

);

-- Insert some data

INSERT INTO Users (UserID, Username, Password) VALUES (1, 'admin', 'adminpassword');

INSERT INTO Users (UserID, Username, Password) VALUES (2, 'user', 'userpassword');

DECLARE @username NVARCHAR(50), @password NVARCHAR(50);

SET @username = 'user\_input';

SET @password = 'password\_input';

-- Vulnerable query

EXEC('SELECT \* FROM Users WHERE Username = ''' + @username + ''' AND Password = ''' + @password + '''');

SELECT \* FROM Users

WHERE Username = '' OR 1=1 -- ' AND Password = 'password\_input';

### SQL Injection Consequences

1. **Authentication Bypass**: As shown, the attacker can log in without knowing valid credentials.
2. **Data Leakage**: An attacker can modify the SQL query to extract sensitive information.
3. **Data Modification/Deletion**: The attacker can inject malicious code to modify or delete data.
4. **Database Corruption**: In the worst case, the attacker can corrupt or drop entire tables.

### Example of SQL Injection Prevention

#### 1. **Using Parameterized Queries**

With parameterized queries, user input is treated as data rather than part of the SQL command. This prevents attackers from injecting SQL code.

#### 2. **Using Stored Procedures**

Another method is to use **stored procedures**, where SQL logic is predefined and does not directly concatenate user inputs.

Stored procedures protect against SQL injection because the query plan is precompiled, and user inputs are treated as parameters, not code.

#### 3. **Input Validation and Escaping**

Ensure that user input is validated to conform to expected formats (e.g., username should only contain alphanumeric characters). Also, escape special characters where necessary.

**SQL Collation:**

**SQL Collation** refers to a set of rules that determine how data is sorted and compared in a SQL Server database. These rules define how characters are treated in terms of case sensitivity, accent marks, and language or locale-specific sorting conventions. Collation affects string comparison, sorting, and indexing behavior in SQL Server, and is an important consideration for databases that handle multilingual or regional data.

### Key Components of SQL Collation

1. **Character Set**: The set of characters (like Latin, Cyrillic, etc.) supported by the collation.
2. **Sort Order**: Defines how characters are sorted. For example, whether "A" comes before or after "a", or how characters with accents (like é and e) are ordered.
3. **Case Sensitivity**:
   * **Case-Sensitive (CS)**: Treats A and a as different characters.
   * **Case-Insensitive (CI)**: Treats A and a as the same character.
4. **Accent Sensitivity**:
   * **Accent-Sensitive (AS)**: Treats accented characters like é and e as different.
   * **Accent-Insensitive (AI)**: Treats accented characters like é and e as the same.
5. **Kana Sensitivity**:
   * **Kana-Sensitive (KS)**: Differentiates between Japanese Kana characters Hiragana and Katakana.
   * **Kana-Insensitive (KI)**: Does not differentiate between Hiragana and Katakana.
6. **Width Sensitivity**:
   * **Width-Sensitive (WS)**: Differentiates between single-byte and multi-byte character sets.
   * **Width-Insensitive (WI)**: Treats single-byte and multi-byte characters as equal.

**SQL Collations**: These are collations defined by SQL Server, typically used for backward compatibility with older versions of SQL Server. Example: SQL\_Latin1\_General\_CP1\_CI\_AS.

#### 1. **Setting Collation at Database Creation**

You can specify the collation of a database when creating it:

CREATE DATABASE Sample COLLATE Latin1\_General\_CI\_AS;

This sets the collation of the Sample to Latin1\_General\_CI\_AS (case-insensitive, accent-sensitive).

#### 2. **Setting Collation on a Column**

You can also set collation at the column level for specific string columns:

CREATE TABLE MyTable (

Name NVARCHAR(50) COLLATE Latin1\_General\_CS\_AS, -- Case-sensitive and accent-sensitive

Description NVARCHAR(255) -- Uses the default collation of the database

);

#### 3. **Collation in Queries**

You can override the default or column-level collation in a query using the COLLATE clause:

SELECT \*

FROM MyTable

WHERE Name = 'John' COLLATE Latin1\_General\_CS\_AS; -- Case-sensitive search

### Changing Collation

#### 1. **Changing Database Collation**

You can change the collation of an existing database, but this does not automatically change the collation of existing columns:

**ALTER DATABASE MyDatabase COLLATE Latin1\_General\_CS\_AS;**

#### 2. **Changing Column Collation**

If you need to change the collation of an individual column:

ALTER TABLE MyTable

ALTER COLUMN Name NVARCHAR(50) COLLATE Latin1\_General\_CI\_AS;

### Default Collation in SQL Server

By default, SQL Server uses the collation SQL\_Latin1\_General\_CP1\_CI\_AS. However, this can be changed during the installation of SQL Server or later when creating databases and columns.

### Summary

* **Collation** controls how strings are compared and sorted in SQL Server.
* It defines rules for **case sensitivity**, **accent sensitivity**, and **locale-specific sorting**.
* Collation can be applied at the database, table, or column level, and can be changed as needed.
* Proper collation choices are important for handling **multilingual data** and **regional variations** in applications.

Collation plays a critical role in SQL Server, particularly in databases handling multiple languages or requiring specific sorting and comparison behaviors.

**Testing Methods:**

In SQL Server (or in general software testing), **Black Box Testing** and **White Box Testing** refer to different approaches for testing the functionality and behavior of the system. Each approach has its own focus and testing methods.

### 1. **Black Box Testing in SQL Server**

**Black Box Testing** involves testing the functionality of the SQL queries, stored procedures, triggers, and other database components without knowledge of the internal code structure. The tester focuses on providing input and validating the output, ensuring that the database behaves as expected without delving into the underlying logic or code.

#### Example of Black Box Testing in SQL Server:

Consider you have a stored procedure GetCustomerOrders that retrieves orders for a specific customer based on their CustomerID.

CREATE PROCEDURE GetCustomerOrders

@CustomerID INT

AS

BEGIN

SELECT OrderID, OrderDate, TotalAmount

FROM Orders

WHERE CustomerID = @CustomerID;

END;

In **black box testing**, you would test this procedure by:

1. Providing different inputs (valid and invalid CustomerID values).
2. Verifying that the output matches the expected result (correct number of orders, correct data).
3. Testing boundary cases (e.g., what happens if no orders exist for a customer).
4. Ensuring proper error handling (e.g., what happens if an invalid CustomerID is passed).

### 2. **White Box Testing in SQL Server**

**White Box Testing** (also known as clear box, glass box, or structural testing) involves testing the internal workings of the SQL code. In this method, the tester has access to the internal logic and can test specific branches, conditions, loops, and error handling mechanisms in the SQL code. The focus is on testing the implementation, ensuring that the code behaves as expected and is optimized.

#### Example of White Box Testing in SQL Server:

Consider the same stored procedure GetCustomerOrders:

CREATE PROCEDURE GetCustomerOrders

@CustomerID INT

AS

BEGIN

-- Step 1: Check if CustomerID exists

IF EXISTS (SELECT 1 FROM Customers WHERE CustomerID = @CustomerID)

BEGIN

-- Step 2: Fetch orders

SELECT OrderID, OrderDate, TotalAmount

FROM Orders

WHERE CustomerID = @CustomerID;

END

ELSE

BEGIN

-- Step 3: Return an error message

PRINT 'Customer does not exist';

END

END;

In **white box testing**, you would:

1. **Analyze the internal logic**: Check the logic flow, such as the IF EXISTS check, and the structure of the query inside the BEGIN block.
2. **Test different branches**: Verify both the case when the customer exists and when the customer does not exist.
3. **Check SQL performance**: Optimize and test the performance of the queries (e.g., indexing, query plan analysis).
4. **Test for SQL vulnerabilities**: Verify that the stored procedure is safe from issues like SQL injection.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

NORMALIZATION

Normalization is a database design technique used to reduce data redundancy and improve data integrity.

It involves organizing data into multiple related tables based on rules called normal forms (NF).

1NF (First Normal Form)

Rule: Each table cell must contain a single value, and each record needs to be unique.

Example: Unnormalized Table

Let’s assume we have a table Customers with repeating groups in a single row (unnormalized):

CREATE TABLE Customers\_Unnormalized

(

CustomerID INT,

CustomerName VARCHAR(100),

Address VARCHAR(255),

Orders VARCHAR(255) -- Multiple orders stored as a single value, separated by commas

);

EXAMPLE:

(1, 'John Doe', '123 Maple St', '101,103,104'),

(2, 'Sam Brown', '789 Pine St', '106,107')

(1, 'John Doe', '123 Maple St', '101'),

(1, 'John Doe', '123 Maple St', '103'),

(1, 'John Doe', '123 Maple St', '104'),

Conversion to 1NF:

To convert to 1NF, separate repeating groups into individual rows. Each order will have its own row.

CREATE TABLE Customers\_1NF (

CustomerID INT,

CustomerName VARCHAR(100),

Address VARCHAR(255),

OrderID INT

);

2NF (Second Normal Form)

Rule: It should be in 1NF, and all non-key attributes must depend on the whole primary key, not part of it.

Example: 1NF Table with Partial Dependencies

Assume the Orders table in 1NF is:

CREATE TABLE Orders\_1NF (

CustomerID INT,

OrderID INT,

ProductName VARCHAR(100),

CustomerName VARCHAR(100),

Address VARCHAR(255) -- This depends only on CustomerID

);

Conversion to 2NF:

To convert to 2NF, remove partial dependencies. In this case, CustomerName and Address depend only on CustomerID,

so we need to create a separate Customers table.

-- Customers table

CREATE TABLE Customers\_2NF (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

Address VARCHAR(255)

);

-- Orders table

CREATE TABLE Orders\_2NF (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductName VARCHAR(100),

FOREIGN KEY (CustomerID) REFERENCES Customers\_2NF(CustomerID)

);

3NF (Third Normal Form)

Rule: It should be in 2NF, and there should be no transitive dependencies (i.e., non-key attributes should not depend on other non-key attributes).

Example: 2NF Table with Transitive Dependencies

Let’s assume the Customers\_2NF table is:

Example: 2NF Table with Transitive Dependencies

Let’s assume the Customers\_2NF table is:

CREATE TABLE Customers\_2NF (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

Address VARCHAR(255),

City VARCHAR(100),

ZipCode VARCHAR(20)

);

In this table, City and ZipCode have a transitive dependency, as City is determined by ZipCode.

Conversion to 3NF:

To achieve 3NF, we create a new table for ZipCode and move City to that table.

-- ZipCodes table

CREATE TABLE ZipCodes\_3NF (

ZipCode VARCHAR(20) PRIMARY KEY,

City VARCHAR(100)

);

-- Customers table

CREATE TABLE Customers\_3NF (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR(100),

Address VARCHAR(255),

ZipCode VARCHAR(20),

FOREIGN KEY (ZipCode) REFERENCES ZipCodes\_3NF(ZipCode)

);

Summary of the Tables Created in Normalization Process:

1NF - Separate repeating groups into individual rows.

2NF - Remove partial dependencies by creating separate tables for data depending on only part of the primary key.

3NF - Remove transitive dependencies by isolating non-key attributes that depend on other non-key attributes.

Each step of normalization ensures that data is structured more efficiently, reducing redundancy and ensuring that data dependencies make sense.