**What is a Database?**

A *database* is a set of data stored in a computer. This data is usually structured in a way that makes the data easily accessible.

#### What is a Relational Database?

A relational database is a type of database. It uses a structure that allows us to identify and access data in relation to another piece of data in the database. Often, data in a relational database is organized into tables.

#### What is a Relational Database Management System (RDBMS)?

A relational database management system (RDBMS) is a program that allows you to create, update, and administer a relational database. Most relational database management systems use the SQL language to access the database.

#### What is SQL?

SQL (**S**tructured **Q**uery **L**anguage) is a programming language used to communicate with data stored in a relational database management system. SQL syntax is similar to the English language, which makes it relatively easy to write, read, and interpret.

Many RDBMSs use SQL (and variations of SQL) to access the data in tables. For example, SQLite is a relational database management system. SQLite contains a minimal set of SQL commands (which are the same across all RDBMSs). Other RDBMSs may use other variants.

(SQL is often pronounced in one of two ways. You can pronounce it by speaking each letter individually like “S-Q-L”, or pronounce it using the word “sequel”.)

#### Tables: Rows and Columns

Tables can have hundreds, thousands, sometimes even millions of rows of data. These rows are often called records.

SQL Server Version List

2019

2017

2016

2014

2012

2008 R2

2008

2005

2000

7.0

6.0

SQL Server Data Files

MDF -- Master data file

LDF -- Log Data File

NDF -- Second Data file

Database Objects:

Table

Stored Procedure

Function

Triggers

Views

Cursor

Index

Understanding the SQL Server Pages

Data is getting stored in Pages

Each page size: 8192 Bytes (8 KB)

1 byte = 1024 kb

1024 kb = 1mb

1024 MB = 1 GB

Each page contains three sections

Page Header -- 96 Bytes

Data section -- Required space by data

Row offset -- if table contains 2 records it wills 4 bytes

**Data types in SQL Server:**

**You will find the list of data types in another file.**

**For SQL Server, the max number of databases you can have on a single SQL Server instance is 32,767.**

**A single SQL Server can have 1 default instance and up to 15 named instances of the relational engine.**

**Database Creation:**

CREATE DATABASE Product -- Here Product is the Database Name.

DROP DATABASE Product

**Table in SQL Server:**

**Tables** are database objects that contain all the data in a database. In **tables**, data is logically organized in a row-and-column format similar to a spreadsheet. Each row represents a unique record, and each column represents a field in the record. ... A standard user-defined **table** can have up to 1,024 columns.

CREATE TABLE Demo

(

Empid INT,

FirstName VARCHAR(100),

LastName VARCHAR(50),

Salary INT,

Birthdate datetime

)

INSERT INTO Demo VALUES(1,'Venki','G',100,GETDATE())

INSERT INTO Demo VALUES(2,'Venki','G',100,GETDATE())

INSERT INTO Demo VALUES(3,'Venki','G',100,GETDATE())

INSERT INTO Demo VALUES(4,'Venki','G',100,GETDATE()),

(5,'Venki','G',100,GETDATE()),

(6,'Venki','G',100,GETDATE())

SP\_HELP 'DEMO' -- Here Demo is the Table Name

SELECT \*

INTO Testing

FROM Demo

SELECT \* FROM Testing

SELECT \*

INTO Emp

FROM Demo

WHERE 1 = 2

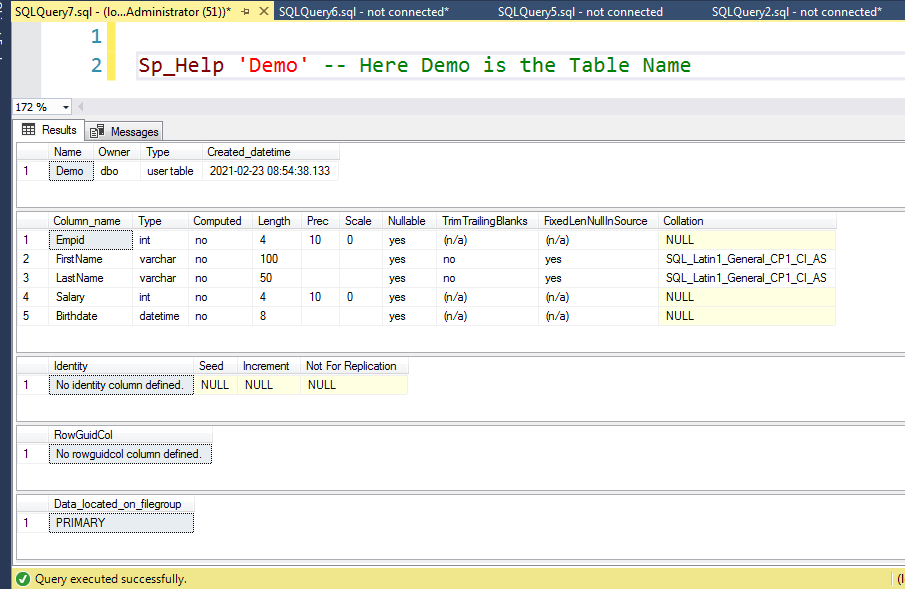
SELECT \* FROM Emp

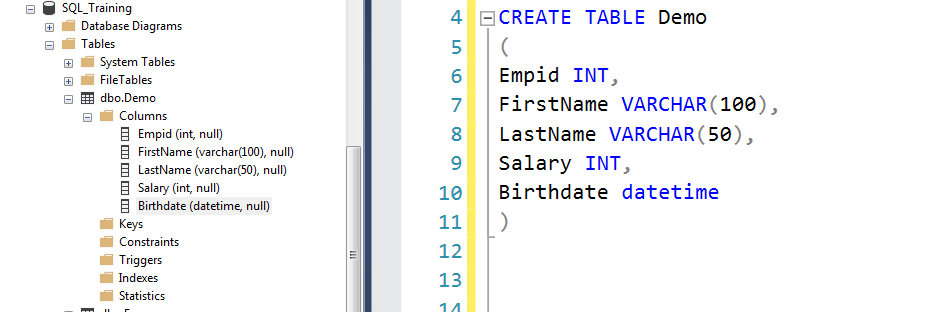
SELECT Empid,FirstName

--INTO Testing1

FROM Demo

SELECT \* FROM Testing1

****

****

**Operator:**

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations

SQL Arithmetic Operators

+ (Addition)

-- (Subtraction)

\* (Multiplication)

/ (Division)

Bitwise Operator

AND

OR

Comparison Operators

=

!=

<>

>

>=

<

<=

IN

NOT IN

BETWEEN

CREATE TABLE Operators

(

Id INT,

Num1 INT,

Num2 INT,

Num3 INT

)

SELECT \* FROM Operators

INSERT INTO Operators values(1,20,23,1)

INSERT INTO Operators values(2,6,90,90)

INSERT INTO Operators values(4,78,34,100)

SELECT \* FROM Operators

SELECT Id EmployyeId,Num1, Num2,Num1 + Num2 AS Addaation FROM Operators Op

SELECT Id ,Num1, Num2,Num1 - Num2 AS Subtraction FROM Operators Op

SELECT Id ,Num1, Num2,Num1 \* Num2 AS Multipliction FROM Operators Op

SELECT Id ,Num1, Num2,Num1 / Num2 AS Division FROM Operators Op

Select \* FROM Operators

SELECT \* from Operators --WHERE Id = 1

SELECT \*FROM Operators Where id = 2 AND Num1 = 87

SELECT \*FROM Operators Where id = 3 AND Num1 = 78 --AND Num2 = 1000

SELECT \* from Operators --WHERE Id = 1

--SELECT \*FROM Operators Where id = 2 OR Num1 = 87

SELECT \*FROM Operators Where id = 2 OR Num1 = 78

SELECT \*FROM Operators Where id = 3 AND Num1 = 78

SELECT \*FROM Operators Where id = 3 AND Num1 = 78

Select \* FROM Operators

SELECT \*FROM Operators Where (id = 2 OR Num1 = 78 )

AND (Num3 = 100)

Select \* FROM Operators WHERE Num2 = 34

Select \* FROM Operators

Select \* FROM Operators WHERE Num2 >= 34

SELECT \* FROM Operators WHERE Id IN (2,3,4)

Select \* FROM Operators

SELECT \* FROM Operators WHERE Id NOT IN (4)

SELECT \* FROM Operators

SELECT \* from Operators WHERE Id BETWEEN 3 AND 4

SELECT \* FROM Operators WHERE Id !=2

SELECT \* FROM Operators WHERE Id <> 2 OR Num1 > 100 AND Num2 = 100

**Datatype Example:**

CREATE TABLE Temp

(

Id INT,

FirstName VARCHAR(5),

LastName varchar(30),

Gender Char(1)

)

INSERT into Temp values(1,'Venki','venkat','M')

INSERT into Temp values(1,'Venki','venkat','Ma')

INSERT into Temp values(1,'G','venkat','M')

Select \* FROM Temp

CREATE TABLE Temp

(

Id INT,

FirstName VARCHAR(5),

LastName varchar(30),

Birthdate DATETIME

)

INSERT into Temp values(1,'Venki','venkat','1500-01-01')

INSERT into Temp values(1,'Venki','venkat','Ma')

INSERT into Temp values(1,'G','venkat','M')

Select \* FROM Temp

How to Delete Records in SQL Server:

SELECT \* FROM [dbo].[Testing]

DELETE FROM Testing WHERE Id = 2

DELETE FROM Testing WHERE FirstName ='Gaddam' AND LastName = 'Rani'

SELECT \* FROM Testing1

DELETE FROM Testing1

How to Update the Records in SQL Server:

CREATE TABLE AddressTesting

(

Id INT,

FirstName VARCHAR(100),

Country varchar(100),

Location VARCHAR(100)

)

INSERT INTO AddressTesting Values(1,'Venki','India','AP')

INSERT INTO AddressTesting Values(2,'Ramu','India','AP')

INSERT INTO AddressTesting Values(3,'Srinu','India','AP')

INSERT INTO AddressTesting Values(4,'Mohit','UK','DT')

INSERT INTO AddressTesting Values(5,'Suresh','UK','RE')

INSERT INTO AddressTesting Values(6,'Somesh','UK','SE')

SELECT \* FROM [dbo].[AddressTesting]

UPDATE TableName

SET COLUMN = VALUE

WHERE ID = 1

UPDATE AddressTesting

SET FirstName = 'Suresh'

UPDATE AddressTesting

SET FirstName = 'Mahesh'

WHERE Id = 1

UPDATE AddressTesting

SET FirstName = 'Venkatesh',

Country = 'India',

Location = 'AP'

WHERE Id IN (6)

**Aggregate Functions in SQL Server:**

**An aggregate function performs a calculation one or more values and returns a single value.**

**Sum**

**Min**

**Max**

**Avg**

**Count**

SELECT SUM(Num1) AS Summaion FROM Operators

SELECT SUM(Num1+ Num2) AS Sumum FROM Operators

SELECT AVG(Num1) FROM Operators

SELECT MAX(Num2) AS MaxValue FROM Operators

SELECT MIN(Num2) AS Minvalue FROM Operators

SELECT COUNT(Num1) AS CountNumber,Max(Num1),Min(Num1),Avg(Num1),SUM(Num1)

FROM Operators

Order by Clause in SQL Server:

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.

SELECT \* FROM AddressTesting

ORDER BY id

SELECT \* FROM AddressTesting

ORDER BY id DESC

SELECT \* FROM AddressTesting

SELECT \* FROM AddressTesting

ORDER BY FirstName DESC

SELECT \* FROM AddressTesting

ORDER BY FirstName DESC,Country ASC

SELECT \* FROM AddressTesting

SELECT \* FROM AddressTesting

ORDER BY Country DESC,FirstName ASC,Id DESC

Group by Clause:

The **GROUP BY** clause in SQL Server allows grouping of rows of a query. Generally, GROUP BY is used with an aggregate SQL Server function, such as SUM, AVG, etc.

CREATE TABLE Employee

(

Id INT,

FirstName Varchar(100),

Gender varchar(100),

Department VARCHAR(100),

Salary INT

)

INSERT INTO Employee Values(1,'Venki','M','IT',430)

INSERT INTO Employee Values(2,'Rami','M','IT',43)

INSERT INTO Employee Values(3,'Srinu','F','IT',234)

INSERT INTO Employee Values(4,'Subbu','F','Hardware',675)

INSERT INTO Employee Values(5,'Josh','F','Hardware',234)

INSERT INTO Employee Values(6,'Mahesb','M','Software',678)

INSERT INTO Employee Values(7,'Suresh','M','Software',123)

INSERT INTO Employee Values(8,'Steve','F','Software',78)

INSERT INTO Employee Values(9,'Venki','M','IT',430)

INSERT INTO Employee Values(10,'Rami','M','IT',43)

INSERT INTO Employee Values(11,'Srinu','F','IT',234)

INSERT INTO Employee Values(12,'Subbu','F','Hardware',675)

INSERT INTO Employee Values(13,'Josh','F','Hardware',234)

INSERT INTO Employee Values(14,'Mahesb','M','Software',678)

SELECT Gender,SUM(Salary) AS TotalSalary FROM Employee

GROUP BY Gender

SELECT Department,COUNT(Id) AS EmployeeCount FROM Employee

GROUP BY Department

SELECT Department,Gender,COUNT(Id) AS EmployeeCount FROM Employee

GROUP BY Department,Gender

ORDER BY Department DESC

How to comment the code in SQL Server:

--Select \* FROM Employee (Singal line Commenting)

/\*

Select DISTINCT Department FROM Employee (Multi line commenting the Code)

Select distinct Department FROM Employee

Select distinct Department FROM Employee

\*/

Having Clause:

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

SELECT Department,COUNT(\*) AS TotalEmpCount FROM Employee

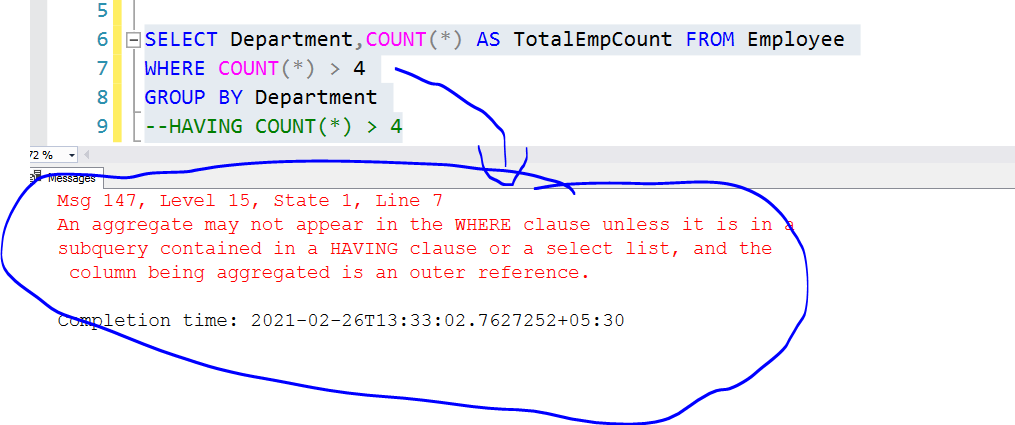
GROUP BY Department

HAVING COUNT(\*) > 4

SELECT Department,Gender,COUNT(\*) AS TotalEmpCount FROM Employee

GROUP BY Department,Gender

HAVING COUNT(\*) > 2

****

DISTINCT Keyword:

The SELECT DISTINCT statement is used to return only distinct (different) values.

SELECT DISTINCT Department FROM Employee

SELECT DISTINCT Department,Gender FROM Employee

SELECT DISTINCT Gender,COUNT(id) FROM Employee

GROUP BY Gender

**TOP Keyword:**

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

SELECT TOP 2 \* FROM Employee

SELECT TOP 5 \* FROM Employee

SELECT TOP 50 PERCENT \* FROM Employee

SELECT TOP 6 \* FROM Employee

How to Insert NULL values:

CREATE TABLE Emp

(

EmpId INT,

FirstName VARCHAR(100),

LastName VARCHAR(100),

MiddleName VARCHAR(100),

Salary INT

)

INSERT INTO Emp values(1,'Venki','G',NULL,300)

INSERT INTO Emp values(1,'Venki','G','',300)

INSERT INTO Emp values(1,'Venki','G',300)

INSERT INTO Emp

(

EmpId,

FirstName,

LastName

)

VALUES

(

13,'G','Ramu')

SELECT \* FROM Emp

WHERE Salary IS NOT NULL

SELECT \* FROM Emp

WHERE Salary IS NULL

Sp\_Help 'Emp'

Constraints in SQL Server:

Constraints in SQL Server are predefined rules and restrictions that are enforced in a single column or multiple columns, regarding the values allowed in the columns, to maintain the integrity, accuracy, and reliability of that column’s data. In other words, if the inserted data meets the constraint rule, it will be inserted successfully. If the inserted data violates the defined constraint, the insert operation will be aborted.

## Not Null Constraint: A Not null constraint restricts the insertion of null values into a column. If we are using a Not Null Constraint for a column then we cannot ignore the value of this column during an insert of data into the table.

CREATE TABLE Demo

(

EmpId INT,

FirstName varchar(100),

LastName varchar(100) NOT NULL,

Salary INT

)

Select \* FROM Demo

Sp\_Help 'Demo'

INSERT INTO Demo Values(1,'G','Rmu',3499)

INSERT INTO Demo Values(2,'G','ui',4566)

INSERT INTO Demo Values(2,'G','','Rmu')

## Primary Key Constraint:

## A Primary key uniquely identifies each row in a table. It cannot accept null and duplicate data. One or more of the columns of a table can contain a Primary key.

CREATE TABLE Demo

(

EmpId INT PRIMARY KEY,

FirstName varchar(100),

LastName varchar(100),

Salary INT

)

INSERT INTO Demo Values(1,'Venk','G',340)

INSERT INTO Demo Values(2,'Venk','G',340)

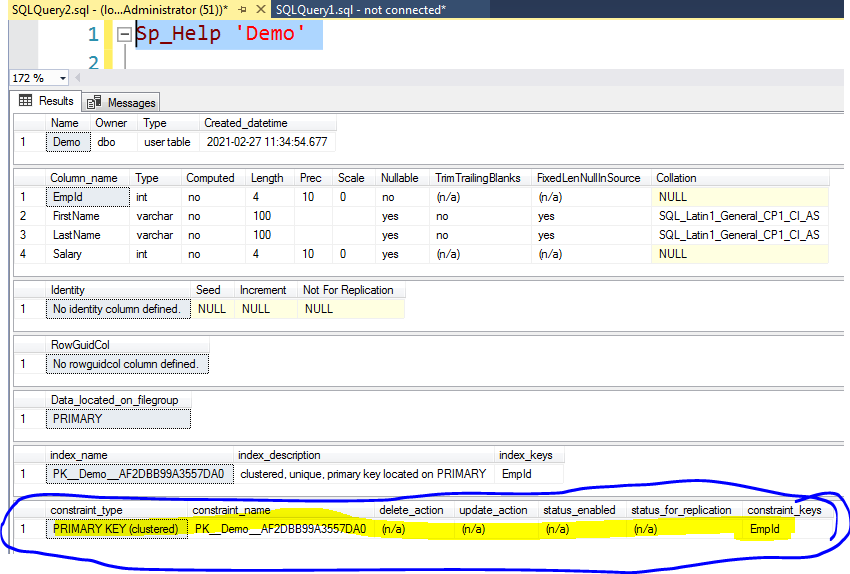
INSERT INTO Demo Values(NULL,'Venk','G',340)

INSERT INTO Demo Values('','Venk','G',340)

INSERT INTO Demo Values( ,'Venk','G',340)

Select\* FROM Demo

Sp\_Help 'Demo'



## Unique Constraint

It ensures that each row for a column must have a unique value. It is like a Primary key but it can accept only one null value. In a table one or more column can contain a Unique Constraint.

CREATE TABLE Demo

(

EmpId INT ,

FirstName varchar(100) UNIQUE,

LastName varchar(100),

Salary INT

)

INSERT INTO Demo Values(1,'Mike','Albert',2399)

INSERT INTO Demo Values(2,'Smith','E',200)

INSERT INTO Demo Values(3,NULL,'Eng',200)

INSERT INTO Demo Values(4,NULL,'Eng',200)

Select \* FROM Demo

Sp\_Help 'Demo'

Primary key:

It will allow only Unique Values

It will not accept any single null value also

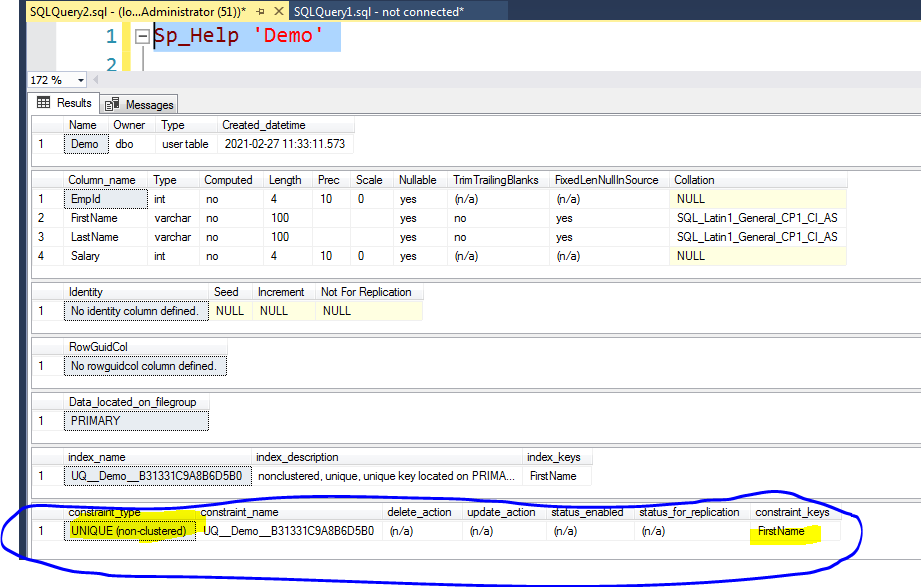
It will create Unique Clustered Index

Unique Key:

It will allow only Unique Values

It will accept one single null value.

It will create non clustered unique Index



## Default Constraint:

Specifies a default value for when a value is not specified for this column. If in an insertion query any value is not specified for this column then the default value will be inserted into the column.

CREATE TABLE Demo

(

EmpId INT ,

FirstName varchar(100) ,

LastName varchar(100),

Salary INT,

Country VARCHAR(100) DEFAULT 'India'

)

INSERT INTO Demo values (1,'Smith','Mike',234)

(

EmpId,

FirstName,

LastName,

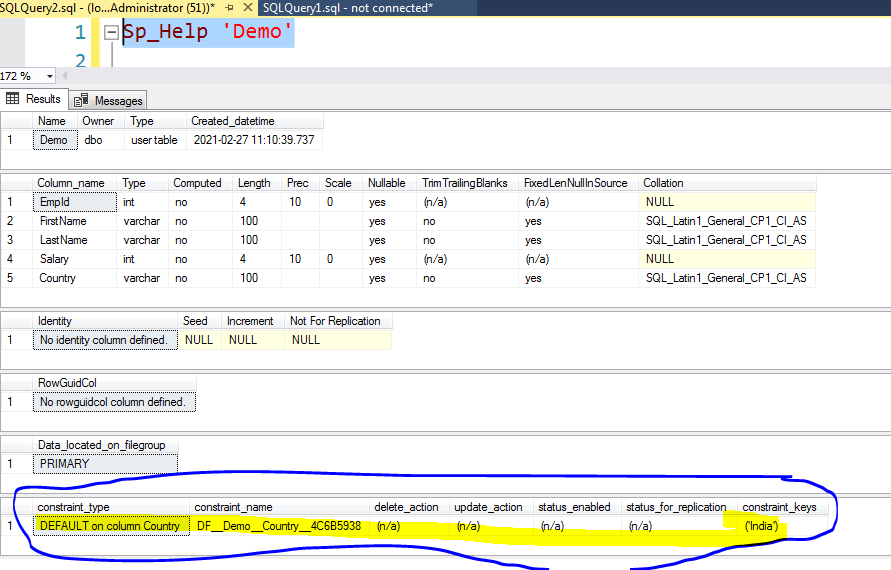
Salary

)

Values

(1,'Smith','Mike',234)

Sp\_Help 'Demo'

****

## Check Constraint:

A Check constraint checks for a specific condition before inserting data into a table. If the data passes all the Check constraints then the data will be inserted into the table otherwise the data for insertion will be discarded. The CHECK constraint ensures that all values in a column satisfies certain conditions.

CREATE TABLE Demo

(

EmpId INT ,

FirstName varchar(100) ,

LastName varchar(100),

Salary INT CHECK(Salary > 1000)

)

Sp\_Help 'Demo'

INSERT INTO Demo Values(1,'Smti','Mi',500)

INSERT INTO Demo Values(1,'Smti','Mi','')

INSERT INTO Demo

(

EmpId,

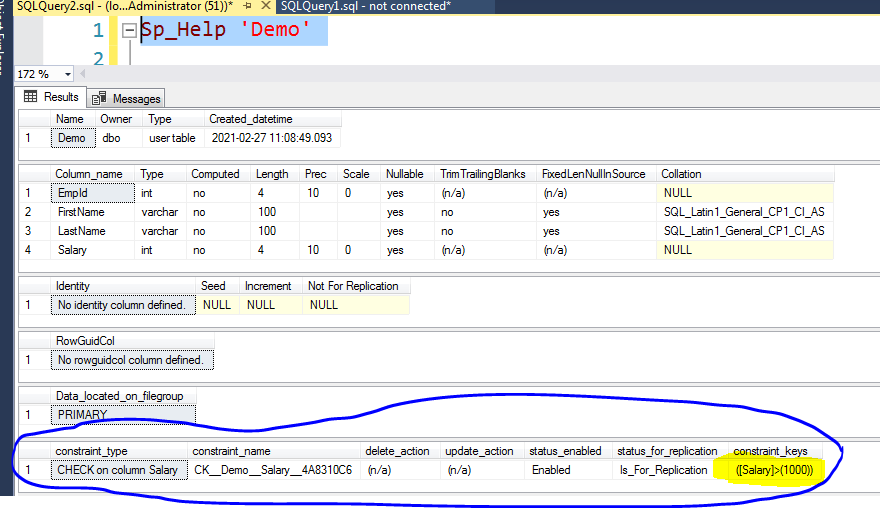
FirstName,

LastName,

Salary

)values(1,'Smith','Mike')

Select \* FROM Demo



**Identity in SQL Server:**

A SQL Server IDENTITY column is a special type of column that is used to automatically generate key values based on a provided seed (starting point) and increment. SQL Server provides us with a number of functions that work with the IDENTITY column. In this tip, we will go through these functions with examples.

CREATE TABLE Demo

(

EmpId INT IDENTITY(1,1),

FirstName varchar(100) ,

LastName varchar(100),

Salary INT

)

Sp\_Help 'Demo'

Select \* FROM Demo

INSERT INTO Demo Values('Dummu','Sravn',4500)

Select \* FROM Demo

INSERT INTO Demo

(

EmpId,

FirstName,

LastName,

Salary

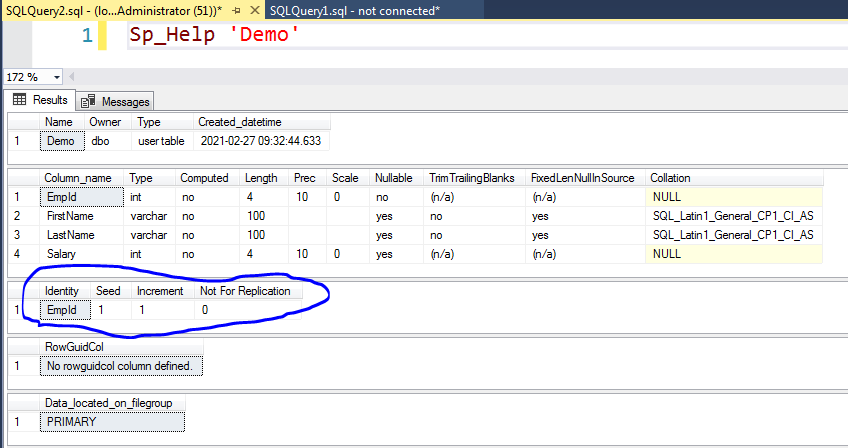
)

Values(15,'Dummu','Sravn',4500)

SET IDENTITY\_INSERT Demo ON

INSERT INTO Demo Values(15,'Dummu','Sravn',4500)

SET IDENTITY\_INSERT Demo OFF

****

## SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

CREATE TABLE Dept

(

DeptId INT PRIMARY KEY,

DeptName VARCHAR(100)

)

INSERT INTO Dept values(1,'IT')

INSERT INTO Dept values(2,'Hardware')

INSERT INTO Dept values(3,'Software')

SELECT \* FROM Dept

CREATE TABLE Employee

(

EmpId INT,

FirstName varchar(100),

Gender Varchar(100),

Salary INT,

DeptId INT FOREIGN KEY REFERENCES Dept(Deptid)

)

DROP Table Dept

DROP Table Employee

Select \* FROM Dept

Select \* FROM EMployee

INSERT INTO Employee values(1,'Venki','M',1200,1)

INSERT INTO Employee values(2,'Venki','M',1200,10)

INSERT INTO Employee values(3,'Venki','M',1200,3)

Sp\_Help 'Dept'

Sp\_Help 'Employee'

**String Functions in SQL Server:**

LEFT:

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

LEFT(string, number\_of\_chars)

Right

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

RIGHT(string, number\_of\_chars)

Lower:

LOWER(text)

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

Len:

LEN(string)

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

Replace:

REPLACE(string, old\_string, new\_string)

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

Replicate:

REPLICATE(string, integer)

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

Reverse:

The REVERSE() function reverses a string and returns the result.

REVERSE(string)

Upper:

UPPER(text)

The UPPER() function converts a string to upper-case.

LTRIM:

LTRIM(string)

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

RTIM:

RTRIM(string)

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

SELECT \*,UPPER(FirstName) AS UpperCase FROM Employee

SELECT \*,LOWER(FirstName) AS UpperCase FROM Employee

SELECT FirstName,LEFT(FirstName,4) AS UpperCase FROM Employee

SELECT FirstName,RIGHT(FirstName,4) AS Rightfun FROM Employee

SELECT FirstName,Reverse(FirstName) AS Reversefun FROM Employee

SELECT FirstName,LEN(FirstName) AS Reversefun FROM Employee

SELECT FirstName,REPLICATE(FirstName,3) AS Reversefun FROM Employee

DECLARE @Test VARCHAR(100)

SET @Test = ' Gaddam Venki'

Select LTRIM(@Test)

DECLARE @Test1 VARCHAR(100)

SET @Test1 = 'Gaddam Venki '

Select RTRIM(@Test1)

## The SQL LIKE Operator

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

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

% - The percent sign represents zero, one, or multiple characters

\_ - The underscore represents a single character

Select \* FROM Employee where FirstName LIKE 's%' OR Department LIKE 'H%'

Select \* FROM Employee where FirstName LIKE 's%' AND Department LIKE 'H%'

Select \* FROM Employee where FirstName LIKE '%s%'

Select \* FROM Employee where FirstName LIKE '%s'

Select \* FROM Employee where FirstName LIKE '\_\_e%'

Date Time Functions in SQL Server:

CREATE TABLE Demo select

upper(Name) upper\_nm,

lower(Name) lower\_nm,

right(Name,3) right\_a,

left(name,4) left\_a,

len(name) length\_a,

replace(groupname,' ','/') replace\_value,

replicate(Name,2) replicate\_nm,

reverse(name) reverse\_nm,

stuff(name,2,3,'\*&') stuff\_a,

(select upper(string\_agg(left(value,1),' ')) from string\_split(GroupName,space(1))) abbrevation,

substring(GroupName,1,charindex(' ',GroupName)) first\_name,

substring(GroupName,charindex(' ',GroupName),charindex(substring(GroupName,1,charindex(' ',GroupName)),GroupName)) as second\_name

from [BCMPQA].[Department]

(

EmpId INT,

FirstName VARCHAR(100),

Salary INT,

DateofBirth datetime

)

INSERT INTO Demo VALUES(1,'Ramu',4500,'2015-12-08 17:19:16')

INSERT INTO Demo VALUES(2,'Venki',3433,'2014-09-15 15:18:16')

INSERT INTO Demo VALUES(3,'Subbu',1233,'2013-05-06 10:20:16')

INSERT INTO Demo VALUES(4,'Srinu',7655,'2015-04-26 18:38:16')

SELECT \*,

YEAR(DateofBirth) AS YearValue ,

MONTH(DateofBirth) AS MonthValue,

DAY(DateofBirth) AS DayNumber,

DATENAME(YEAR, DateofBirth) AS 'Year',

DATENAME(QUARTER, DateofBirth) AS 'Quarter',

DATENAME(MONTH, DateofBirth) AS 'Month',

DATENAME(DAYOFYEAR, DateofBirth) AS 'DayOfYear',

DATENAME(DAY, DateofBirth) AS 'Day',

DATENAME(WEEK, DateofBirth) AS 'Week',

DATENAME(WEEKDAY, DateofBirth) AS 'WeekDay',

DATENAME(HOUR, DateofBirth) AS 'Hour',

DATENAME(MINUTE, DateofBirth) AS 'Minute',

DATENAME(SECOND, DateofBirth) AS 'Second'

FROM Demo

SELECT \*,

DATEADD(YYYY,1,dateofBirth) AS DateaddYear,

DATEADD(Q,1,DateofBirth) DateaddQuarter,

DATEADD(MM,1,DateofBirth) AS DateaddMonth,

DATEADD(D,1,dateofbirth) AS DateaddDay,

DATEADD(D,-1,dateofbirth) AS Daysubtraction

FROM Demo

SELECT \*,

DATEADD(YYYY,1,dateofBirth) AS DateaddYear,

DATEADD(Q,1,DateofBirth) DateaddQuarter,

DATEADD(MM,1,DateofBirth) AS DateaddMonth,

DATEADD(D,1,dateofbirth) AS DateaddDay,

DATEADD(D,-1,dateofbirth) AS Daysubtraction

FROM Demo

SELECT \*,

GETDATE() AS Today,

DATEDIFF(YYYY,DateofBirth,GETDATE()) YearDifference,

DATEDIFF(MM,DateofBirth,GETDATE()) MonthDifference,

DATEDIFF(DD,DateofBirth,GETDATE()) AS DayDifference

FROM Demo

SELECT \*,

DATEPART(WEEKDAY,DateofBirth) AS DayNumber,

DATEPART(YYYY,DateofBirth) AS YearDate,

DATEPART(MM,DateofBirth) AS MonthDate

FROM Demo

SELECT \*,

DATEPART(WEEKDAY,DateofBirth) AS DayNumber,

DATENAME(WEEKDAY,DateofBirth) AS DayName

FROM Demo

SELECT \*,

EOMONTH(DateofBirth) AS Endofmonth,

ISDATE(DateofBirth) AS IsDateornot,

ISDATE('2019-31-31') AS IsDateCheck

FROM Demo

How to alter the Table Name:

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

LastName VARCHAR(100),

Salary INT

)

INSERT INTO Demo Values(1,'Venki','Gaddam',3400)

INSERT INTO Demo Values(2,'Ramu','Pavani',36200)

SELECT \* FROM Demo

ALTER TABLE Demo

ADD Price INT

ALTER TABLE Demo

DROP COLUMN Price

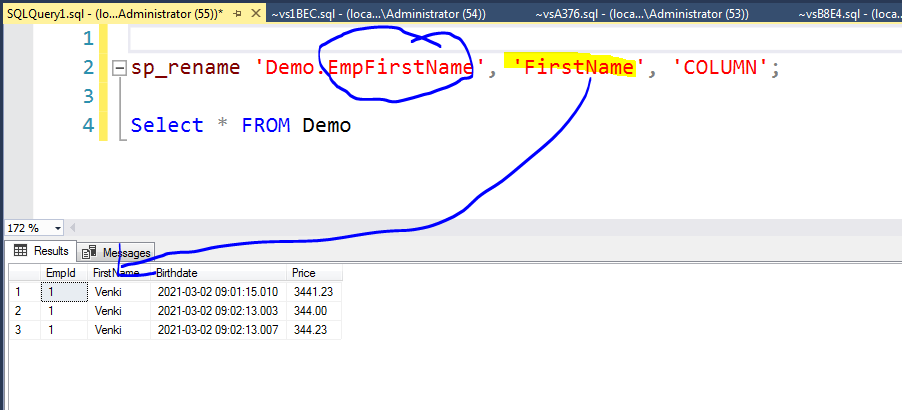
ALTER TABLE Demo

ALTER COLUMN FirstName varchar(40)

ALTER TABLE Demo

ALTER COLUMN FirstName Char(100)

sp\_rename 'Demo.FirstName', 'EmpFirstName', 'COLUMN';



Questions:

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

LastName VARCHAR(100),

DateofBirth VARCHAR(100)

)

INSERT INTO Demo Values(2,'Gaddam','Venki','2019-15-31')

INSERT INTO Demo Values(3,'Pavani','Ramu','2019-12-31')

SELECT \*,ISDATE(DateofBirth) IsValid FROM Demo

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

LastName VARCHAR(100),

DateofBirth VARCHAR(100)

)

INSERT INTO Demo Values(2,'Gaddam','Venki','2019-15-31')

INSERT INTO Demo Values(3,'Pavani','Ramu','2019-12-31')

ALTER TABLE Demo

Alter column FirstName VARCHAR(3)

Insert data from one table to another Table:

CREATE TABLE Child

(

EmpId INT,

FirstName VARCHAR(100),

LastName VARCHAR(100),

Salary INT

)

CREATE TABLE Parent

(

Id INT,

EmpFirstName VARCHAR(100),

EmpLastName VARCHAR(100),

Salary INT

)

INSERT INTO Parent VALUES(1,'Gaddam','Venki',3400)

INSERT INTO Parent VALUES(2,'V','Ramu',360)

INSERT INTO Parent VALUES(3,'S','Pavai',874)

SELECT \* FROM Parent

SELECT \* FROM Child

INSERT INTO Child

SELECT \* FROM Parent

INSERT INTO Child

(

EmpId,

FirstName,

LastName

)

SELECT

Id,

EmpFirstName,

EmpLastName

FROM Parent

ISNULL:

The ISNULL() function returns a specified value if the expression is NULL.

ISNULL(expression, value)

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

LastName VARCHAR(100),

Salary INT

)

INSERT INTO Demo Values(1,'Venki',NULL,458)

INSERT INTO Demo Values(2,'Ramu',NULL,360)

INSERT INTO Demo Values(3,'Subbu','Mahesh',5100)

INSERT INTO Demo Values(4,'Pavani',NULL,1000)

INSERT INTO Demo Values(4,NULL,'Rakesh',1000)

SELECT \*,ISNULL(LastName,'Missing Name') AS IsNullVale FROM Demo

SELECT EmpFirstName,

ISNULL(EmpFirstName,'FirstName Missing') AS FirtName,

LastName,

ISNULL(LastName,'LastName Missing') AS LastName

FROM Demo

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.

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

SELECT Color,

CASE WHEN Color IS NULL THEN 'No Color'

ELSE 'Color is there'

END AS ColorName

FROM Product

SELECT

Color ,

CASE WHEN Color = 'Red' THEN 'Red Color'

WHEN Color = 'Yellow' THEN 'Yellow Color'

WHEN Color = 'Blue' THEN 'Blue Color'

ELSE 'No Color'

END AS ColorName

FROM Product

CREATE TABLE dbo.Employee

(

EmployeeID INT IDENTITY PRIMARY KEY,

EmployeeName VARCHAR(100) NOT NULL,

Gender VARCHAR(1) NOT NULL,

StateCode VARCHAR(20) NOT NULL,

Salary money NOT NULL,

)

SET IDENTITY\_INSERT [dbo].[Employee] ON

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (201, N'Jerome', N'M', N'FL', 83000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (202, N'Ray', N'M', N'AL', 88000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (203, N'Stella', N'F', N'AL', 76000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (204, N'Gilbert', N'M', N'Ar', 42000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (205, N'Edward', N'M', N'FL', 93000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (206, N'Ernest', N'F', N'Al', 64000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (207, N'Jorge', N'F', N'IN', 75000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (208, N'Nicholas', N'F', N'Ge', 71000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (209, N'Lawrence', N'M', N'IN', 95000.0000)

INSERT [dbo].[Employee] ([EmployeeID], [EmployeeName], [Gender], [StateCode], [Salary]) VALUES (210, N'Salvador', N'M', N'Co', 75000.0000)

SET IDENTITY\_INSERT [dbo].[Employee] OFF

SELECT EmployeeName,Gender,Salary

FROM Employee

ORDER BY

CASE WHEN Gender = 'F' THEN Salary END DESC,

CASE WHEN Gender='M' THEN Salary END ASC

SELECT

CASE

WHEN Salary >=80000 AND Salary <=100000 THEN 'Director'

WHEN Salary >=50000 AND Salary <80000 THEN 'Senior Consultant'

ELSE 'Director'

END AS Designation,

MIN(salary) as MinimumSalary,

MAX(Salary) as MaximumSalary

FROM Employee

GROUP BY

CASE

WHEN Salary >=80000 AND Salary <=100000 THEN 'Director'

WHEN Salary >=50000 AND Salary <80000 THEN 'Senior Consultant'

ELSE 'Director'

END

SELECT

\* FROM Employee

WHERE (CASE WHEN Gender = 'M' THEN EmployeeID

END ) > 204

IIF:

The IIF() function returns a value if a condition is TRUE, or another value if a condition is FALSE.

IIF(condition, value\_if\_true, value\_if\_false)

SELECT ProductID,IIF(productId > 10,'Yes','No') AS Result FROM Product

SELECT Color,IIF(Color = 'Black','Yes','No') AS Result FROM Product

SELECT Color,

IIF(Color = 'Black','Black Color',

IIF(Color = 'Red','Red Color','No Color'))

FROM Product

Decimal Data Type:

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

Birthdate datetime,

Price Decimal(5,2)

)

INSERT INTO Demo Values(1,'Venki','2019-02-12',44.65)

INSERT INTO Demo Values(1,'Venki','2019-02-12',4340.1)

SELECT \* FROM Demo

Cast Function:

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

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

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

Birthdate datetime,

Price Decimal(5,2)

)

INSERT INTO Demo Values(1,'Venki','2019-02-12',44.65)

INSERT INTO Demo Values(1,'Venki','2019-02-12',4340.1)

SELECT \*,

CAST(price AS int) AS PriceInt,

CAST(Birthdate as date) BirthDate,

CAST(EmpId AS varchar(100)) AS EmpFirstName,

CAST(FirstName as int)

FROM Demo

Concatenation:

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

CONCAT(string1, string2, ...., string\_n)

SELECT

ProductID + ' ' + MakeFlag AS Result,

ProductNumber + ' \*\* ' + Name AS TotalResult,

CONCAT(productId,makeflag) AS Value1,

CONCAT(Productid,Concat(' ',MakeFlag)) AS MainResult

FROM Product

COALESCE :

The COALESCE() function returns the first non-null value in a list.

COALESCE(val1, val2, ...., val\_n)

SELECT COALESCE (NULL,'A','B')

SELECT COALESCE (NULL,100,20,30,40)

SELECT COALESCE (NULL,NULL,20,NULL,NULL)

SELECT COALESCE (NULL,NULL,NULL,NULL,NULL,'Prashanth')

SUBSTRING:

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

SUBSTRING(string, start, length)

**string** Required. The string to extract from

**start** Required. The start position. The first position in string is 1

**length** Required. The number of characters to extract. Must be a positive number

SELECT

Name,

ProductNumber,

SUBSTRING(Name,1,4) AS PartString,

SUBSTRING(ProductNumber,1,2) AS ProductNumber

FROM Product

VARCHAR & NVARCHAR & CHAR & NCHAR difference:

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

LastName CHAR(100),

MiddleName NVARCHAR(100),

Address NCHAR(100)

)

INSERT INTO Demo VALUES(101,'Ramu','Suresh','G','Hyderabad')

INSERT INTO Demo VALUES(102,'Mahesh','Subbu','Raju','Bangalore')

INSERT INTO Demo VALUES(103,'Koti','Ravidra','Pavani','Chennai')

SELECT \* ,

DATALENGTH(FirstName) AS varcharvalue,

DATALENGTH(LastName) AS CharValue,

DATALENGTH(MiddleName) AS NvarcharValue,

DATALENGTH(Address) AS NCharValue

FROM Demo

SET OPERATORS:

/\*

UNION

UNION ALL

INTERSECT

EXCEPT

\*/

**UNION**: is used to combine the results of two or more SELECT statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.

## UNION ALL: This operation is similar to Union. But it also shows the duplicate rows

**Intersect**: operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same.

## Except: The Minus operation combines results of two SELECT statements and return only those in the final result, which belongs to the first set of the result.

CREATE TABLE First

(

EmpId INT,

FirstName varchar(100),

Salary INT

)

INSERT INTO First Values(101,'Venki',2300)

INSERT INTO First Values(102,'Ramu',6500)

INSERT INTO First Values(103,'Srinu',2001)

CREATE TABLE Secondtbl

(

Id INT,

EmpFirstName varchar(100),

EmpSalary INT

)

INSERT INTO Secondtbl Values(101,'Venki',2300)

INSERT INTO Secondtbl Values(104,'Pavani',4353)

INSERT INTO Secondtbl Values(103,'Srinu',2001)

SELECT \* FROM First

UNION

SELECT \* FROM Secondtbl

SELECT \* FROM First

UNION ALL

SELECT \* FROM Secondtbl

SELECT \* FROM First

INTERSECT

SELECT \* FROM Secondtbl

SELECT \* FROM First

SELECT \* FROM Secondtbl

SELECT \* FROM First

EXCEPT

SELECT \* FROM Secondtbl

CONVERT:

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

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

SELECT ModifiedDate,ProductId,

CONVERT(date,modifieddate,108) AS Date1,

CONVERT(varchar,modifieddate,103),

CONVERT(varchar,modifieddate,109),

CONVERT(varchar,productid) AS ChangeType,

CONVERT(int,ProductNumber) AS ModifiedName

ABS:

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

ABS(number)

SELECT ABS(34) AS Value

SELECT ABS(-34) AS Value

SELECT ABS(-34.0987) AS Value

SELECT ABS(34.0987) AS Value

CHARINDEX:

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

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

CHARINDEX(substring, string, start)

SELECT CHARINDEX('a','Gaddam Venki',5)

SELECT CHARINDEX('a','Hyderabad',7)

SELECT Name,ProductNumber,

CASE WHEN CHARINDEX('DC',ProductNumber,1) > 0 THEN 'Exists' ELSE 'Not Exisst'

END AS DerivedColumn,

CASE WHEN SUBSTRING(ProductNumber,1,2) = 'DC' THEN 'Value there'

ELSE 'Value Not there' END AS DerivedCol

FROM Product

CREATE TABLE Demo

(

EmpId INT,

FirstName VARCHAR(100),

LastName VARCHAR(100),

Gmail VARCHAR(100),

Salary INT

)

INSERT INTO Demo VALUES(1,'Venki','G','gaddam.venki@gmail.com',21)

INSERT INTO Demo VALUES(2,'Srinu','G','Srinu.venki@gmail.com',433)

INSERT INTO Demo VALUES(3,'Rami','S','Rmu.venki@gmail.com',34)

INSERT INTO Demo VALUES(4,'Pavani','D','Pavani.venki@gmail.com',123)

SELECT \* FROM Demo

SELECT \*,

SUBSTRING(Gmail,1,CHARINDEX('@',Gmail)-1) AS GmailName,

SUBSTRING(Gmail,CHARINDEX('@',Gmail)+1,LEN(Gmail)) AS DomainName

FROM Demo

PATINDEX:

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

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

PATINDEX(%pattern%, string)

SELECT PATINDEX('%a%','Gaddam Venki')

SELECT PATINDEX('d%','Hyderabad')

SELECT Name,

ProductNumber,

CASE WHEN CHARINDEX('DC',ProductNumber,1) > 0 THEN 'Exists'

ELSE 'Not EXISTS'

END AS CharIndexValue,

CASE WHEN SUBSTRING(ProductNumber,1,2) = 'DC' THEN 'Exists'

ELSE 'NOT EXISTS'

END AS SubstrinvValue,

CASE WHEN PATINDEX('D%',ProductNumber) > 0 THEN 'Exists'

ELSE 'NOT EXISTS'

END AS PatIndexValue

FROM Product

STUFF:

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

STUFF (source\_string, start, length, add\_string)

SELECT STUFF('Gaddam Venkateswarlu',2,3,'Inserting')

SELECT STUFF('Tiger woods',4,2,'Gold Player')

SELECT

ProductNumber,

Name,

STUFF(ProductNumber,2,3,'DC') AS StuffFunction

FROM Product

Variables in SQL Server:

## Types of Variable: Local, Global

MS SQL has two types of variables:

1. Local variable
2. Global variable.

### Local variable:

* A user declares the local variable.
* By default, a local variable starts with **@.**
* Every local variable scope has the restriction to the **current batch or procedure** within any given session.

### Global variable:

* The system maintains the global variable**.**A user cannot declare them.
* The global variable starts with **@@**
* It stores **session related information**.

-- Local VARIABLE in SQL Server

DECLARE @Number INT

SET @Number = 10

SELECT @Number AS FirstValue

DECLARE @Second VARCHAR(100)

SET @Second = 'Venki'

SELECT @Second AS Name

-- Global Variables in SQL Server

SELECT @@LANGUAGE

SELECT @@VERSION

In these variables we can store only one single value (Scalar value).Local variables can be available within the batch of execution but in case of Global variables we can use across of any session and any scope.

DECLARE @Name VARCHAR(100)

SELECT @Name = ProductNumber FROM Product WHERE ProductID = 1

SELECT @Name

GO

DECLARE @Name VARCHAR(100)

SET @Name = (SELECT ProductNumber FROM Product WHERE ProductID = 1)

SELECT @Name

DECLARE @Name VARCHAR(100)

SELECT @Name = ProductNumber FROM Product WHERE ProductID > 1

SELECT @Name

GO

DECLARE @Name VARCHAR(100)

SET @Name = (Select ProductNumber FROM Product WHERE ProductID > 1)

SELECT @Name

GO

IF:

IF boolean\_expression

**BEGIN**

{ statement\_block }

**END**

* If the condition evaluates to **True,** then T-SQL statements followed by **IF** keyword will be executed.
* If the condition evaluates to **False,** then T-SQL statements followed by **ELSE** keyword will be executed.
* Once, either IF T-SQL statements or ELSE T-SQL statement is executed then other unconditional T-SQL statements continues execution.

DECLARE @City VARCHAR(100)

SET @City = 'Bangalore'

DECLARE @Country VARCHAR(100)

SET @Country = 'India'

IF @City = 'Bangalore' AND @Country = 'India'

SELECT 'City is Bangalore'

ELSE

SELECT 'City is Hyderabed'

DECLARE @City VARCHAR(100)

SET @City = 'Hyderabad '

DECLARE @Country VARCHAR(100)

SET @Country = 'India'

IF @City = 'Bangalore'

SELECT 'City is Bangalore'

ELSE IF @City = 'Chennai'

SELECT 'City is Chennai'

ELSE

SELECT 'City is Hyderabed'

WHILE:

SQL WHILE loop provides us with the advantage to execute the SQL statement(s) repeatedly until the specified condition result turn out to be false.

WHILE condition

BEGIN

   {...statements...}

END

DECLARE @Id INT

SET @Id = 1

WHILE (@Id <= 10)

BEGIN

PRINT @Id

SET @Id = @Id + 1

END

CREATE TABLE Demo

(

ID INT,

Name VARCHAR(100)

)

DECLARE @Id INT

SET @Id = 1

WHILE (@Id <= 10)

BEGIN

INSERT INTO Demo VALUES(@Id,'Demo' + CAST(@Id as varchar) )

SET @Id = @Id + 1

END

SELECT \* FROM Demo

Cascading Options:

CREATE TABLE Dept

(

DeptId INT PRIMARY KEY,

DeptName VARCHAR(100)

)

INSERT INTO Dept values(1,'IT')

INSERT INTO Dept values(2,'Hardware')

INSERT INTO Dept values(3,'Software')

CREATE TABLE Employee

(

EmpId INT PRIMARY KEY,

FirstName varchar(100),

Gender Varchar(100),

Salary INT,

DeptId INT NULL DEFAULT 100000

)

INSERT INTO Employee values(1,'Venki','M',1200,1)

INSERT INTO Employee values(2,'Venki','M',1200,2)

INSERT INTO Employee values(4,'Venki','M',1200,3)

ALTER TABLE Employee

ADD CONSTRAINT FK\_Dept\_Eid FOREIGN KEY(DeptId) REFERENCES Dept(DeptId)

Go

UPDATE Dept

SET DeptId = 12

WHERE DeptId = 1

DELETE FROM Dept WHERE DeptId = 1

UPDATE Employee

SET DeptId = 233

WHERE DeptId = 1

SELECT \* FROM Dept

SELECT \* FROM Employee

ALTER TABLE Employee

DROP CONSTRAINT FK\_Dept\_Eid;

ALTER TABLE Employee

ADD CONSTRAINT FK\_Dept\_Eid FOREIGN KEY(DeptId) REFERENCES Dept(DeptId)

ON DELETE CASCADE ON UPDATE CASCADE

UPDATE Dept

SET DeptId = 121

WHERE DeptId = 1

SELECT \* FROM Dept

DELETE FROM Dept WHERE DeptId = 121

ALTER TABLE Employee

ADD CONSTRAINT FK\_Dept\_Eid FOREIGN KEY(DeptId) REFERENCES Dept(DeptId)

ON DELETE SET NULL ON UPDATE SET NULL

UPDATE Dept

SET DeptId = 121

WHERE DeptId = 2

SELECT \* FROM Dept

SELECT \* FROM Employee

DELETE FROM Dept WHERE DeptId = 1

ALTER TABLE Employee

ADD CONSTRAINT FK\_Dept\_Eid FOREIGN KEY(DeptId) REFERENCES Dept(DeptId)

ON DELETE SET DEFAULT ON UPDATE SET DEFAULT

Go

**Temp Tables in SQL Server:**

A temporary table in SQL Server, as the name suggests, is a database table that exists temporarily on the database server. A temporary table stores a subset of data from a normal table for a certain period of time.

Temporary tables are particularly useful when you have a large number of records in a table and you repeatedly need to interact with a small subset of those records. In such cases instead of filtering the data again and again to fetch the subset, you can filter the data once and store it in a temporary table. You can then execute your queries on that temporary table. Temporary tables are stored inside “tempdb” which is a system database. Let’s take a look at how you can use a temporary data in a simple scenario.

There are two types of Temp Tables.

1. Local Temp Tables

2. Global Temp Tables.

Local Temp Tables:

CREATE TABLE #Demo

(

EmpId INT,

EmpName VARCHAR(100),

Salary INT

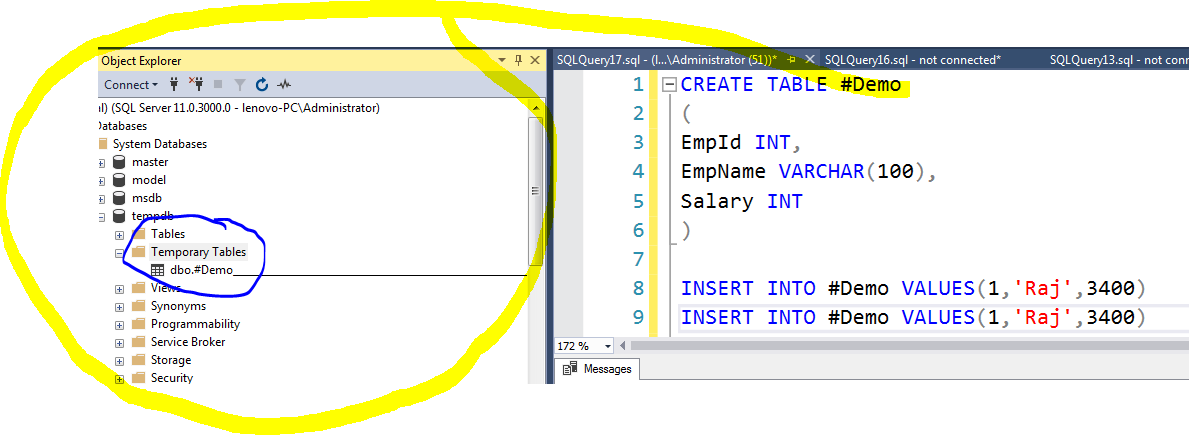
)

INSERT INTO #Demo VALUES(1,'Raj',3400)

INSERT INTO #Demo VALUES(1,'Raj',3400)

INSERT INTO #Demo VALUES(1,'Raj',3400)

GO



**Global Temp Tables:**

CREATE TABLE ##Demo

(

EmpId INT,

EmpName VARCHAR(100),

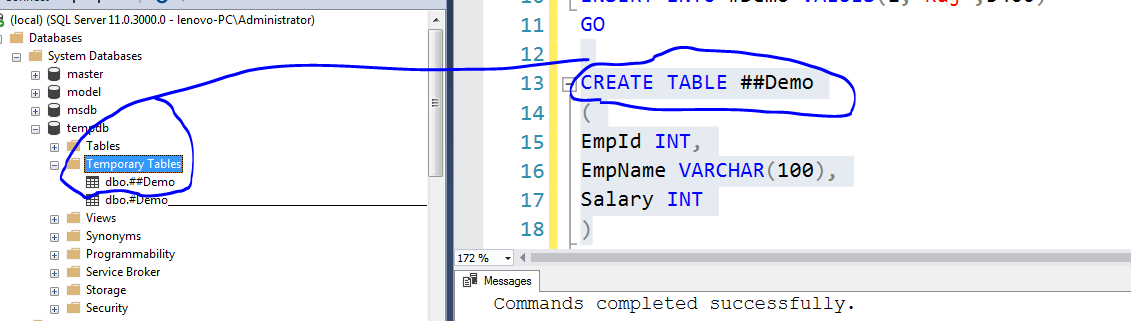
Salary INT

)

INSERT INTO ##Demo VALUES(1,'Raj',3400)

INSERT INTO ##Demo VALUES(1,'Raj',3400)

INSERT INTO ##Demo VALUES(1,'Raj',3400)



**Table Variables in SQL Server:**

The table variable is a special data type that can be used to store temporary data similar to a temporary table. The syntax for the table variable looks similar to defining a new table using the CREATE TABLE statement:

DECLARE @Products TABLE

(ProductID INT,

ProductName VARCHAR(30)

);

In the above query, we specified a table variable using the DECLARE and TABLE keyword. The table variable name must start with the @ symbol. We also define table columns, data types, constraint similar to a regular table.

The table variable scope is within the batch. We can define a table variable inside a stored procedure and function as well. In this case, the table variable scope is within the stored procedure and function. We cannot use it outside the scope of the batch, stored procedure or function.

DECLARE @Demo TABLE

(

ID INT,

Name VARCHAR(100),

Salary INT

)

INSERT INTO @Demo VALUES(101,'Venki',4300)

INSERT INTO @Demo VALUES(102,'Srinu',4908)

INSERT INTO @Demo VALUES(103,'Ramu',1230)

SELECT \* FROM @Demo

**Joins in SQL Server:**

Joins are used to relate one or more tables in SQL Server. Joins are a part of a SQL Statement that retrieves rows from a table or tables according to specified conditions.

There are 5 types of Joins.

1. Inner Join

2. Left Join

3. Right Join

4. Full Outer Join

5. Cross Join

SELECT EmployeeID,ContactID,Title,BirthDate,Gender,HireDate FROM tblEmployee

SELECT EmployeeId,DepartmentId,StartDate,Enddate FROM tblEmployeeHistory

SELECT Emp.EmployeeID,ContactID,Title,BirthDate,Gender,HireDate,

EmpHist.DepartmentID,

EmpHist.EmployeeID

FROM tblEmployee AS Emp

INNER JOIN

tblEmployeeHistory EmpHist

ON Emp.EmployeeId = EmpHist.EmployeeId

GO

SELECT Emp.EmployeeID,ContactID,Title,BirthDate,Gender,HireDate,

EmpHist.DepartmentID,

EmpHist.EmployeeID

FROM tblEmployee AS Emp

LEFT JOIN

tblEmployeeHistory EmpHist

ON Emp.EmployeeId = EmpHist.EmployeeId

GO

SELECT Emp.EmployeeID,ContactID,Title,BirthDate,Gender,HireDate,

EmpHist.DepartmentID,

EmpHist.EmployeeID

FROM tblEmployee AS Emp

RIGHT JOIN

tblEmployeeHistory EmpHist

ON Emp.EmployeeId = EmpHist.EmployeeId

GO

SELECT Emp.EmployeeID,ContactID,Title,BirthDate,Gender,HireDate,

EmpHist.DepartmentID,

EmpHist.EmployeeID

FROM tblEmployee AS Emp

FULL OUTER JOIN

tblEmployeeHistory EmpHist

ON Emp.EmployeeId = EmpHist.EmployeeId

GO

SELECT Emp.EmployeeID,ContactID,Title,BirthDate,Gender,HireDate,

EmpHist.DepartmentID,

EmpHist.EmployeeID

FROM tblEmployee AS Emp

CROSS JOIN

tblEmployeeHistory EmpHist

**Rank Functions in SQL Server:**

We perform calculations on data using various aggregated functions such as Max, Min, and AVG. We get a single output row using these functions. SQL Server provides SQL RANK functions to specify rank for individual fields as per the categorizations. It returns an aggregated value for each participating row. SQL RANK functions also know as Window Functions.

We have the following rank functions.

ROW\_NUMBER()

RANK()

DENSE\_RANK()

NTILE()

CREATE TABLE ExamResult

(StudentName VARCHAR(70),

Subject VARCHAR(20),

Marks INT

);

INSERT INTO ExamResult

VALUES

('Lily',

'Maths',

65

);

INSERT INTO ExamResult

VALUES

('Lily',

'Science',

80

);

INSERT INTO ExamResult

VALUES

('Lily',

'english',

70

);

INSERT INTO ExamResult

VALUES

('Isabella',

'Maths',

50

);

INSERT INTO ExamResult

VALUES

('Isabella',

'Science',

70

);

INSERT INTO ExamResult

VALUES

('Isabella',

'english',

90

);

INSERT INTO ExamResult

VALUES

('Olivia',

'Maths',

55

);

INSERT INTO ExamResult

VALUES

('Olivia',

'Science',

60

);

INSERT INTO ExamResult

VALUES

('Olivia',

'english',

89

);

SELECT Studentname, Subject, Marks,

ROW\_NUMBER() OVER(ORDER BY Marks) RowNumber

FROM ExamResult;

SELECT Studentname, Subject, Marks,

ROW\_NUMBER() OVER(ORDER BY Marks DESC) RowNumber

FROM ExamResult;

SELECT Studentname,Subject, Marks,

ROW\_NUMBER () OVER(PARTITION BY Studentname ORDER BY Marks DESC) Rank

FROM ExamResult

SELECT Studentname,Subject,Marks,

RANK() OVER(ORDER BY Marks DESC) Rank

FROM ExamResult

SELECT Studentname,Subject,Marks,

RANK() OVER(PARTITION BY StudentName ORDER BY Marks DESC) Rank

FROM ExamResult

SELECT Studentname,Subject, Marks,

DENSE\_RANK() OVER(ORDER BY Marks DESC) Rank

FROM ExamResult

SELECT Studentname,Subject, Marks,

DENSE\_RANK() OVER(PARTITION BY StudentName ORDER BY Marks DESC) Rank

FROM ExamResult

SELECT \*,NTILE(2) OVER(ORDER BY Marks DESC) Rank

FROM ExamResult

SELECT \*,NTILE(2) OVER(PARTITION BY StudentName ORDER BY Marks DESC) Rank

FROM ExamResult

**Sub Queries in SQL Server:**

In SQL Server, a subquery is a query within a query. You can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

SELECT \* FROM tblEmployee Emp WHERE EmployeeID IN Outer Query

(

SELECT EmployeeID FROM tblEmployeeHistory Inner Query

)

GO

Here Inner query is independent so we can execute individually without dependent on outer query.

SELECT \* FROM tblEmployee Emp WHERE EmployeeID NOT IN

(

SELECT EmployeeID FROM tblEmployeeHistory

)

GO

SELECT \* FROM tblEmployee Emp WHERE EmployeeID IN

(

SELECT EmployeeID FROM tblEmployeeHistory EmpHist

WHERE EmpHist.EmployeeID = Emp.EmployeeID

)

GO

Here Inner query is dependent on outer query even if you try to run the inner query alone you will get an error. It means that we cannot execute the inner query without outer query called as correlated sub query.

SELECT \* FROM tblEmployee Emp WHERE EmployeeID NOT IN

(

SELECT EmployeeID FROM tblEmployeeHistory EmpHist

WHERE EmpHist.EmployeeID = Emp.EmployeeID

)

GO

SELECT Emp.\* FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON EmpHist.EmployeeID = Emp.EmployeeID

GO

SELECT \* FROM tblEmployee Emp

LEFT JOIN tblEmployeeHistory EmpHist

ON EmpHist.EmployeeID = Emp.EmployeeID

WHERE EmpHist.EmployeeID IS NULL

GO

SELECT \* FROM tblEmployee Emp WHERE EXISTS

(

SELECT \* FROM tblEmployeeHistory EmpHist

WHERE Emp.EmployeeID = EmpHist.EmployeeID

)

GO

SELECT \* FROM tblEmployee Emp WHERE NOT EXISTS

(

SELECT \* FROM tblEmployeeHistory EmpHist

WHERE Emp.EmployeeID = EmpHist.EmployeeID

)

GO

DECLARE @Id INT

SET @Id = 3122

IF EXISTS (SELECT \* FROM tblEmployee WHERE EmployeeID = @Id)

BEGIN

PRINT 'The Condition is True'

END

ELSE

BEGIN

PRINT 'The Condition is False'

END

GO

DECLARE @Id INT

SET @Id = 1

IF NOT EXISTS (SELECT \* FROM tblEmployee WHERE EmployeeID = @Id)

BEGIN

PRINT 'The Condition is true'

END

ELSE

BEGIN

PRINT 'The Condition is False'

END

GO

SELECT

EmployeeID,

BirthDate,

Gender,

Title,

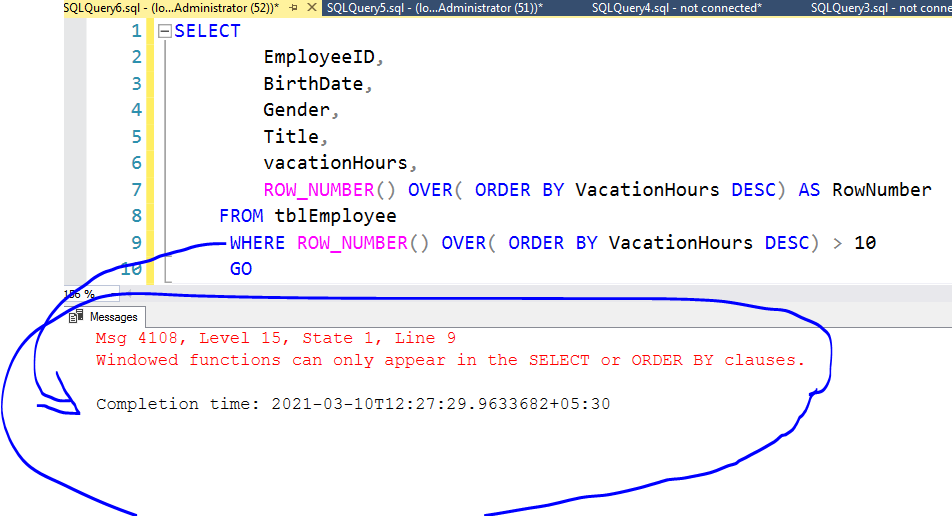
vacationHours,

ROW\_NUMBER() OVER( ORDER BY VacationHours DESC) AS RowNumber

FROM tblEmployee

WHERE ROW\_NUMBER() OVER( ORDER BY VacationHours DESC) > 10

GO



SELECT \* FROM

(

SELECT

EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

ROW\_NUMBER() OVER( ORDER BY VacationHours DESC) AS RowNumber

FROM tblEmployee

) Temp

WHERE RowNumber > 10 AND Title = 'Production Technician - WC50'

GO

SELECT Temp.\*,EmpHist.DepartmentID,EmpHist.StartDate FROM

(

SELECT

EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

ROW\_NUMBER() OVER( ORDER BY VacationHours DESC) AS RowNumber

FROM tblEmployee

) Temp

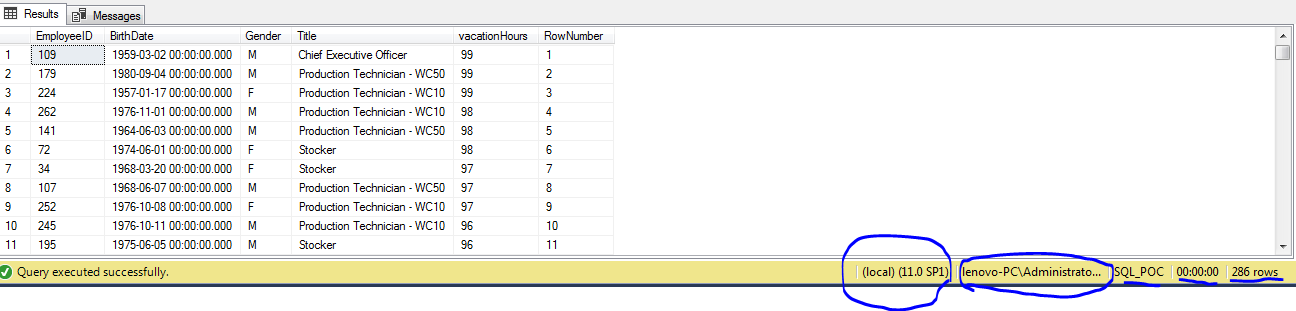
INNER JOIN tblEmployeeHistory EmpHist

ON Temp.EmployeeID = EmpHist.EmployeeID

WHERE RowNumber > 10 AND Title = 'Production Technician - WC50'

GO

**Different options to understand:**



**Common Table Expressions (CTE)**

Were introduced into standard SQL in order to simplify various classes of SQL Queries for which a derived table was just unsuitable. CTE was introduced in SQL Server 2005, the common table expression (CTE) is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. You can also use a CTE in a CREATE a view, as part of the view’s SELECT query. In addition, as of SQL Server 2008, you can add a CTE to the new MERGE statement.

Syntax:

WITH expression\_name [ ( column\_name [,...n] ) ]

AS

( CTE\_query\_definition )

;WITH DemoCTE(EmployeeID,BirthDate,Gender,Title,vacationHours)

AS

(

SELECT EmployeeID,

BirthDate,

Gender,

Title,

vacationHours

FROM tblEmployee

)

SELECT \* FROM DemoCTE

GO

Here we can change the column names also.

;WITH DemoCTE(EmpId,DateofBirth,Gender,Title,vacationHours)

AS

(

SELECT EmployeeID,

BirthDate,

Gender,

Title,

vacationHours

FROM tblEmployee

)

SELECT \* FROM DemoCTE

GO

Here didn’t mention column Names in the CTE Definition

;WITH DemoCTE

AS

(

SELECT EmployeeID,

BirthDate,

Gender,

Title,

vacationHours

FROM tblEmployee

)

SELECT \* FROM DemoCTE

GO

;WITH DemoCTE(EmpId,DateofBirth,Gender,Title,vacationHours,RowNum)

AS

(

SELECT EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

ROW\_NUMBER() OVER(ORDER BY vacationHours) AS RowNum

FROM tblEmployee

)

SELECT DemoCTE.\*,EmpHist.DepartmentID FROM DemoCTE

INNER JOIN tblEmployeeHistory EmpHist

ON DemoCTE.EmpId = EmpHist.EmployeeID

WHERE RowNum > 10 AND

Title = 'Production Technician - WC20'

GO

SELECT Temo.EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

EmpHist.DepartmentID

FROM

(

SELECT

EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

ROW\_NUMBER() OVER( ORDER BY VacationHours DESC) AS RowNumber

FROM tblEmployee

) Temo

INNER JOIN tblEmployeeHistory EmpHist

ON Temo.EmployeeID = EmpHist.EmployeeID

WHERE RowNumber > 10 AND Title = 'Production Technician - WC20'

ORDER BY RowNumber ASC

Today Class pending Doubt:

In today’s class while discussing CTE we have encountered an issue regarding the different results from the below two queries one is Derived Table and another one is CTE(Common Table Expression). There is one keyword we have add as extra in one of the below two queries and rest of all lines are identical. So Please have a close look below code and try to find which keyword I have added extra because of that results are different. If you catch the difference let me know through email.

;WITH DemoCTE(EmpId,DateofBirth,Gender,Title,vacationHours,RowNum)

AS

(

SELECT EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

ROW\_NUMBER() OVER(ORDER BY vacationHours) AS RowNum

FROM tblEmployee

)

SELECT DemoCTE.\*,EmpHist.DepartmentID FROM DemoCTE

INNER JOIN tblEmployeeHistory EmpHist

ON DemoCTE.EmpId = EmpHist.EmployeeID

WHERE RowNum > 10 AND

Title = 'Production Technician - WC20'

GO

SELECT Temo.EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

EmpHist.DepartmentID

FROM

(

SELECT

EmployeeID,

BirthDate,

Gender,

Title,

vacationHours,

ROW\_NUMBER() OVER( ORDER BY VacationHours DESC) AS RowNumber

FROM tblEmployee

) Temo

INNER JOIN tblEmployeeHistory EmpHist

ON Temo.EmployeeID = EmpHist.EmployeeID

WHERE RowNumber > 10 AND

Title = 'Production Technician - WC20'

**Merge Statement in SQL Server:**

Beginning with SQL Server 2008, you can use MERGE command to perform these operations in a single statement. This new command is similar to the UPSERT (fusion of the words UPDATE and INSERT.) command of Oracle where it inserts rows that don't exist and updates the rows that do exist. With the introduction of the MERGE command, developers can more effectively handle common data warehousing scenarios, like checking whether a row exists, and then executing an INSERT or UPDATE or DELETE.

The MERGE statement basically merges data from a source result set to a target table based on a condition that you specify and if the data from the source already exists in the target or not. The new SQL command combines the sequence of conditional INSERT, UPDATE and DELETE commands in a single atomic statement, depending on the existence of a record. The new MERGE command looks like as below:

MERGE <target\_table> [AS TARGET]

USING <table\_source> [AS SOURCE]

ON <search\_condition>

[WHEN MATCHED

THEN <merge\_matched> ]

[WHEN NOT MATCHED [BY TARGET]

THEN <merge\_not\_matched> ]

[WHEN NOT MATCHED BY SOURCE

THEN <merge\_matched> ];

CREATE TABLE Source

(

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

Rate MONEY

)

GO

INSERT INTO Source

VALUES

(1, 'Tea', 10.00),

(2, 'Coffee', 20.00),

(3, 'Muffin', 30.00),

(4, 'Biscuit', 40.00)

GO

CREATE TABLE Target

(

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

Rate MONEY

)

GO

INSERT INTO Target

VALUES

(1, 'Tea', 10.00),

(2, 'Coffee', 25.00),

(3, 'Muffin', 35.00),

(5, 'Pizza', 60.00)

GO

SELECT \* FROM Source

SELECT \* FROM Target

GO

INSERT INTO Target

SELECT ProductID,

ProductName,

Rate

FROM Source

WHERE ProductID = 4

DELETE FROM Target

WHERE ProductID = 5

UPDATE T

SET T.Rate = S.Rate

FROM Source S

INNER JOIN Target T

ON S.ProductID = T.ProductID

GO

INSERT INTO Target

SELECT S.\* FROM Source S

LEFT JOIN Target T

ON S.ProductID = T.ProductID

WHERE T.ProductID IS NULL

UPDATE T

SET T.ProductName = S.ProductName,

T.Rate = S.Rate

FROM Target T

INNER JOIN Source S

ON T.ProductId = S.ProductId

WHERE

(

T.ProductName <> S.ProductName OR

T.Rate <> S.Rate OR

T.ProductID <> S.ProductID

)

GO

DELETE FROM Target

WHERE ProductID NOT IN

(

SELECT S.ProductID FROM Source S

WHERE S.ProductID = Target.ProductID

)

SELECT \* FROM Source

SELECT \* FROM Target

GO

;MERGE TARGET AS T

USING SOURCE AS S

ON (S.ProductId = T.ProductId)

WHEN MATCHED THEN

UPDATE SET T.Rate = S.Rate

WHEN NOT MATCHED BY TARGET THEN

INSERT (ProductId,Productname,Rate) VALUES(S.ProductId,S.ProductName,S.Rate)

WHEN NOT MATCHED BY SOURCE THEN

DELETE;

;MERGE TARGET AS T

USING SOURCE AS S

ON (S.ProductId = T.ProductId)

WHEN MATCHED AND

(S.Rate <> T.Rate OR S.ProductName <> T.ProductName OR S.ProductId <> T.ProductId) THEN

UPDATE SET T.Rate = S.Rate

WHEN NOT MATCHED BY TARGET THEN

INSERT (ProductId,Productname,Rate) VALUES(S.ProductId,S.ProductName,S.Rate)

WHEN NOT MATCHED BY SOURCE THEN

DELETE;

GO

;MERGE TARGET AS T

USING SOURCE AS S

ON (S.ProductId = T.ProductId)

WHEN MATCHED AND

(S.Rate <> T.Rate OR S.ProductName <> T.ProductName OR S.ProductId <> T.ProductId) THEN

UPDATE SET T.Rate = S.Rate,

T.ProductName = S.ProductName

WHEN NOT MATCHED BY TARGET THEN

INSERT (ProductId,Productname,Rate) VALUES(S.ProductId,S.ProductName,S.Rate)

WHEN NOT MATCHED BY SOURCE THEN

DELETE;

GO

**User Defined Functions in SQL Server:**

Like programming languages SQL Server also provides User Defined Functions (UDFs). From SQL Server 2000 the UDF feature was added. UDF is a programming construct that accepts parameters, does actions and returns the result of that action. The result either is a scalar value or result set. UDFs can be used in scripts, Stored Procedures, triggers and other UDFs within a database.  
  
**Benefits of UDF**

1. UDFs support modular programming. Once you create a UDF and store it in a database then you can call it any number of times. You can modify the UDF independent of the source code.
2. UDFs reduce the compilation cost of T-SQL code by caching plans and reusing them for repeated execution.
3. They can reduce network traffic. If you want to filter data based on some complex constraints then that can be expressed as a UDF. Then you can use this UDF in a WHERE clause to filter data.

Types of user defined functions

1. Scalar user defined functions

2. Multi statement Table Valued Function

3. Inline Statement Table Valued Functions

1. Scalar user defined functions

A Scalar UDF accepts zero or more parameters and return a single value. The return type of a scalar function is any data type except text, ntext, image, cursor and timestamp. Scalar functions can be use in a WHERE clause of the SQL Query.

CREATE FUNCTION function-name (Parameters)

RETURNS return-type

AS

BEGIN

Statement 1

Statement 2

.

.

Statement n

RETURN return-value

END

SELECT

EmployeeID,

Title,

Gender,

MaritalStatus,

HireDate,

CASE WHEN Gender = 'M' THEN 'Male'

WHEN Gender = 'F' THEN 'Female'

ELSE 'Unknown'

END AS Genderinfo,

CASE WHEN MaritalStatus = 'M' THEN 'Maried'

WHEN MaritalStatus = 'S' THEN 'Single'

ELSE 'Dont know'

END AS Marriageinfo

FROM tblEmployee

GO

CREATE FUNCTION udfGetInfo(@Gender INT)

RETURNS VARCHAR(100)

AS

BEGIN

DECLARE @GenderInfo VARCHAR(100)

SET @GenderInfo = (SELECT

CASE WHEN Gender = 'M' THEN 'Male'

WHEN Gender = 'F' THEN 'Female'

ELSE 'Unknown'

END AS Genderinfo

FROM tblEmployee

WHERE EmployeeID = @Gender

)

RETURN @GenderInfo

END

GO

SELECT dbo.udfGetGenderInfo('M')

SELECT dbo.udfGetGenderInfo(Gender) AS GenderInfo

FROM tblEmployee

GO

CREATE FUNCTION udfGetDataInfo(@vacationHours INT)

RETURNS VARCHAR(100)

AS

BEGIN

DECLARE @Data VARCHAR(100)

SELECT @Data =

CASE WHEN VacationHours >= 0 AND VacationHours <=10 THEN 'Below Average'

WHEN VacationHours >= 11 AND VacationHours <=20 THEN 'Average'

WHEN VacationHours >= 21 AND VacationHours <=30 THEN 'Above Average'

WHEN VacationHours >= 31 AND VacationHours <=40 THEN 'Good'

ELSE 'Very Good'

END

FROM tblEmployee WHERE VacationHours = @vacationHours

RETURN @Data

END

CREATE FUNCTION udfGetAge(@BirthDate INT)

RETURNS INT

AS

BEGIN

DECLARE @CurrentMonth INT

DECLARE @Age INT

DECLARE @BirthMonth INT

SET @CurrentMonth = MONTH(GetDate())

SET @BirthMonth = (SELECT DATEDIFF(MM,BirthDate,GETDATE())

FROM tblEmployee WHERE EmployeeID = @ BirthDate)

SELECT @Age =

CASE WHEN @BirthMonth - @CurrentMonth <= 12 THEN 0

ELSE DATEDIFF(YY,BirthDate,GETDATE())

END

FROM tblEmployee

WHERE EmployeeID = @ BirthDate

RETURN @Age

END

GO

SELECT

EmployeeID,

Title,

Gender,

MaritalStatus,

BirthDate,

CASE WHEN Gender = 'M' THEN 'Male'

WHEN Gender = 'F' THEN 'Female'

ELSE 'Unknown'

END AS Genderinfo,

CASE WHEN MaritalStatus = 'M' THEN 'Maried'

WHEN MaritalStatus = 'S' THEN 'Single'

ELSE 'Dont know'

END AS Marriageinfo,

dbo.udfGetInfo(Gender) AS GenderStatus,

dbo.udfGetAge(BirthDate) AS EmployeeAge,

VacationHours,

[dbo].[udfGetData](VacationHours) AS VacationHours

FROM tblEmployee

GO

**2. MultiStatement Table Valued Function:**

The main idea of this approach is to reduce code repetition. The main advantages of this idea are:

* Improve maintainability
* Improve the code readability

If we want to gain these benefits in SQL Server, the user-defined functions come at the first point in the list in order to implement this idea.

**Syntax:**

CREATE FUNCTION MultiStatement\_TableValued\_Function\_Name

(

@param1 DataType,

@param2 DataType,

@paramN DataType

)

RETURNS

@OutputTable TABLE

(

@Column1 DataTypeForColumn1 ,

@Column2 DataTypeForColumn2

)

AS

BEGIN

--FunctionBody

RETURN

END

Go

CREATE FUNCTION udfMultistatementFunction

(

@ManagerId INT

)

RETURNS @Employee TABLE

(

EmployeeId INT,

Title VARCHAR(100),

BirthDate DATETIME,

MaritalStatus VARCHAR(100),

Gender VARCHAR(100),

VacationHours INT,

VacationInfo VARCHAR(100)

)

AS

BEGIN

INSERT INTO @Employee

SELECT

EmployeeID,

Title,

BirthDate,

MaritalStatus,

Gender,

VacationHours,

CASE WHEN VacationHours >= 0 AND VacationHours <=10 THEN 'Below Average'

WHEN VacationHours >= 11 AND VacationHours <=20 THEN 'Average'

WHEN VacationHours >= 21 AND VacationHours <=30 THEN 'Above Average'

WHEN VacationHours >= 31 AND VacationHours <=40 THEN 'Good'

ELSE 'Very Good'

END AS VacationInfo

FROM tblEmployee

WHERE ManagerID = @ManagerId

UPDATE @Employee

SET Gender = CASE WHEN Gender = 'M' THEN 'Male'

WHEN Gender = 'F' THEN 'Female'

ELSE 'Unknown'

END

RETURN

END

GO

SELECT \* FROM dbo.udfMultistatementFunction(3)

**3. Inline Table Valued Function:**

**Syntax:**

CREATE FUNCTION [dbo].[udfGetProductList]

(

@Param1,@Param2

)

RETURNS TABLE

AS

RETURN

(

Block of SQL Statements

)

CREATE FUNCTION [dbo].[udfInlineFunction]

(

@ManagerId INT

)

RETURNS TABLE

AS

RETURN

(

SELECT

EmployeeID,

Title,

BirthDate,

MaritalStatus,

Gender,

VacationHours,

CASE WHEN VacationHours >= 0 AND VacationHours <=10 THEN 'Below Average'

WHEN VacationHours >= 11 AND VacationHours <=20 THEN 'Average'

WHEN VacationHours >= 21 AND VacationHours <=30 THEN 'Above Average'

WHEN VacationHours >= 31 AND VacationHours <=40 THEN 'Good'

ELSE 'Very Good'

END AS VacationInfo

FROM tblEmployee

WHERE ManagerID = @ManagerId

)

SELECT \* FROM [dbo].[udfInlineFunction](3)

Even it is possible to write some expression or case statement while return clause also in the Scalar User Defined function. So for your reference please find the below example.

CREATE FUNCTION [dbo].[udfGetData]

(

@VacationHours INT

)

RETURNS VARCHAR(100)

AS

BEGIN

DECLARE @MainStrig VARCHAR(100)

SELECT @MainStrig =

CASE

WHEN @VacationHours >= 0 AND @VacationHours <= 10 THEN 'Below Average'

WHEN @VacationHours > 11 AND @VacationHours <= 20 THEN 'Average'

WHEN @VacationHours > 21 AND @VacationHours <= 30 THEN 'Above Average'

WHEN @VacationHours > 31 AND @VacationHours <= 40 THEN 'Better'

WHEN @VacationHours > 41 AND @VacationHours <= 50 THEN 'Good'

ELSE 'Very Good'

END

FROM tblEmployee;

RETURN

CASE WHEN @MainStrig = 'Below Average' THEN 'First'

WHEN @MainStrig = 'Average' THEN 'Second'

ELSE 'None of the Above'

END

END

**Stored Procedures in SQL Server:**

SQL Server stored procedure is a batch of statements grouped as a logical unit and stored in the database. The stored procedure accepts the parameters and executes the T-SQL statements in the procedure, returns the result set if any.

**Benefits of using a stored procedure**

**It can be easily modified**: We can easily modify the code inside the stored procedure without the need to restart or deploying the application. For example, If the T-SQL queries are written in the application and if we need to change the logic, we must change the code in the application and re-deploy it. SQL Server Stored procedures eliminate such challenges by storing the code in the database. so, when we want to change the logic inside the procedure we can just do it by simple ALTER PROCEDURE statement.

**Reduced network traffic:** When we use stored procedures instead of writing T-SQL queries at the application level, only the procedure name is passed over the network instead of the whole T-SQL code.

**Reusable:**Stored procedures can be executed by multiple users or multiple client applications without the need of writing the code again.

**Security:**Stored procedures reduce the threat by eliminating direct access to the tables. we can also encrypt the stored procedures while creating them so that source code inside the stored procedure is not visible.

**Performance:**The SQL Server stored procedure when executed for the first time creates a plan and stores it in the buffer pool so that the plan can be reused when it executes next time.

**Syntax:**

CREATE PROCEDURE ProcedureName

(

@Param1 Datatype,

@Param2 Datatype

)

AS

BEGIN

Block of SQL Statements

END

CREATE PROCEDURE GetDataInfo

(

@ManagerId INT,

@Gender VARCHAR(100)

)

AS

BEGIN

CREATE TABLE #Test

(

EmployeeId INT,

ManagerId INT,

Title VARCHAR(100),

BirthDate DATETIME,

VacationHours INT,

Gender VARCHAR(100),

MartialStatus VARCHAR(100),

DepartmentID INT,

ModifiedDate DATETIME

)

INSERT INTO #Test

SELECT Emp.EmployeeID,

ManagerID,

Title,

BirthDate,

VacationHours,

Gender,

MaritalStatus,

EmpHist.DepartmentID,

EmpHist.ModifiedDate

FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

WHERE ManagerID = @ManagerId AND Gender = @Gender

UPDATE #Test

SET MartialStatus = 'Marriage'

WHERE EmployeeId = 80

SELECT \*,

CASE WHEN Gender = 'F' THEN 'Female'

ELSE 'Male'

END AS GenderInfo

FROM #Test WHERE VacationHours > 15

END

EXECUTE GetDataInfo @Gender='F',@ManagerId=16

**How to drop existing Table in SQL Server:**

IF OBJECT\_ID('tempdb..#Test') IS NOT NULL

DROP TABLE #Test

IF OBJECT\_ID('SQL\_POC..Test') IS NOT NULL

DROP TABLE Test

**How to pass default values to parameters in SQL Server**

ALTER PROCEDURE [dbo].[GetDataInfo]

(

@ManagerId INT = 3 ,

@Gender VARCHAR(100) = 'M'

)

AS

BEGIN

IF OBJECT\_ID('tempdb..#Test') IS NOT NULL

DROP TABLE #Test

CREATE TABLE #Test

(

EmployeeId INT,

ManagerId INT,

Title VARCHAR(100),

BirthDate DATETIME,

VacationHours INT,

Gender VARCHAR(100),

MartialStatus VARCHAR(100),

DepartmentID INT,

ModifiedDate DATETIME

)

INSERT INTO #Test

SELECT Emp.EmployeeID,

ManagerID,

Title,

BirthDate,

VacationHours,

Gender,

MaritalStatus,

EmpHist.DepartmentID,

EmpHist.ModifiedDate

FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

WHERE ManagerID = @ManagerId AND Gender = @Gender

UPDATE #Test

SET MartialStatus = 'Marriage'

WHERE EmployeeId = 80

We can add join another if want to fetch columns from other table.

SELECT Dept.Name,Dept.GroupName,#Test.\* FROM #Test

INNER JOIN tblDepartment Dept

ON #Test.DepartmentID = Dept.DepartmentID

END

**Error Handling in SQL Server:**

Error handling in SQL Server gives us control over the Transact-SQL code. For example, when things go wrong, we get a chance to do something about it and possibly make it right again. SQL Server error handling can be as simple as just logging that something happened, or it could be us trying to fix an error. It can even be translating the error in SQL language because we all know how technical SQL Server error messages could get making no sense and hard to understand. Luckily, we have a chance to translate those messages into something more meaningful to pass on to the users, developers, etc.

In this article, we’ll take a closer look at the [TRY… CATCH](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/try-catch-transact-sql?view=sql-server-2017) statement: the syntax, how it looks, how it works and what can be done when an error occurs. Furthermore, the method will be explained in a SQL Server case using a group of T-SQL statements/blocks, which is basically SQL Server way of handling errors. This is a very simple yet structured way of doing it and once you get the hang of it, it can be quite helpful in many cases.

Syntax:

BEGIN TRY

--code to try

END TRY

BEGIN CATCH

--code to run if an error occurs

--is generated in try

END CATCH

ERROR\_NUMBER – Returns the internal number of the error

ERROR\_LINE – Returns the line number at which an error happened on

ERROR\_MESSAGE – Returns the most essential information and that is the message text of the error

Begin try

Select 1/0

end try

Begin Catch

Select ERROR\_NUMBER()

Select ERROR\_LINE()

SELECT ERROR\_MESSAGE()

End Catch

BEGIN TRY

SELECT 1 + ' Error Message'

END TRY

BEGIN CATCH

SELECT ERROR\_NUMBER()

SELECT ERROR\_LINE()

SELECT ERROR\_MESSAGE()

END CATCH

GO

BEGIN TRY

SELECT CAST(1 as varchar(100)) + ' Error Message'

END TRY

BEGIN CATCH

SELECT ERROR\_NUMBER()

SELECT ERROR\_LINE()

SELECT ERROR\_MESSAGE()

END CATCH

GO

CREATE TABLE DemoTest

(

Id INT Primary key,

Name varchar(100)

)

CREATE TABLE ErrorLog

(

Id INT IDENTITY(1,1),

ErrorMessage VARCHAR(1000),

ErrorLine INT,

ErrorNumber INT,

TimeStamp DATETIME

)

GO

BEGIN TRY

SELECT 1/0

END TRY

BEGIN CATCH

DECLARE @ErrorMessage VARCHAR(100)

SET @ErrorMessage = ERROR\_MESSAGE()

DECLARE @ErrorLine INT = ERROR\_LINE()

DECLARE @ErrorNumber INT = ERROR\_NUMBER()

INSERT INTO ErrorLog VALUES

(

@ErrorMessage,

@ErrorLine,

@ErrorNumber,

GETDATE()

)

END CATCH

**TRANSACTIONS IN SQL SERVER:**

In general, a **Transaction** is a single unit of work consists of multiple related tasks that should succeed or fail as one atomic unit. To make the concept of the transaction more familiar and why it should go all or none, imagine one of the most critical transaction examples in our daily life, which is withdrawing money from the ATM.

After a hard-working month, you are happy that your salary is released. As it is late to visit the bank after the working hours, your destination now is the ATM. All of us are familiar with the steps of withdrawing money from the ATM, but it deserves listing the common ones below:

BEGIN TRANSACTION

-- Block of SQL Statements

COMMIT TRANSACTION

ROLLBACK TRANSACTION

BEGIN TRANSACTION

INSERT INTO Main VALUES (1,'Venki')

INSERT INTO Main VALUES (2,'Ramu')

COMMIT TRANSACTION

ROLLBACK TRANSACTION

BEGIN TRY

BEGIN TRANSACTION

INSERT INTO Main VALUES(7,'Ramu')

INSERT INTO Main VALUES(1,'Venki')

INSERT INTO Main VALUES(8,'Ramu')

COMMIT TRANSACTION

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

END CATCH

SELECT \* FROM Main

BEGIN TRY

BEGIN TRANSACTION

INSERT INTO Main VALUES(7,'Ramu')

INSERT INTO Main VALUES(1,'Venki')

INSERT INTO Main VALUES(8,'Ramu')

COMMIT TRANSACTION

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

DECLARE @ErrorMsg VARCHAR(1000) = ERROR\_MESSAGE()

DECLARE @ErrorNumber INT = ERROR\_NUMBER()

DECLARE @ErrorLine INT = ERROR\_LINE()

INSERT INTO ErrorLog VALUES

(

@ErrorMsg,

@ErrorLine,

@ErrorNumber,

GETDATE()

)

END CATCH

Select \* FROM ErrorLog

CREATE PROCEDURE TestProcedure

(

@ID INT,

@Name VARCHAR(100)

)

AS

BEGIN

BEGIN TRY

BEGIN TRANSACTION

INSERT INTO Main Values (@ID,@Name)

COMMIT TRANSACTION

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

DECLARE @ErrorMsg VARCHAR(100) = ERROR\_MESSAGE()

DECLARE @ErrorLine INT = ERROR\_LINE()

DECLARE @ErrorNumber INT = ERROR\_NUMBER()

INSERT INTO ErrorLog VALUES

(@ErrorMsg,@ErrorLine,@ErrorNumber,GETDATE())

END CATCH

END

GO

EXECUTE TestProcedure 12,'Ramu'

DECLARE @EmpId INT

SET @EmpId = 6

SELECT \* FROM dbo.udfMultistatementFunction(@EmpId) Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeId = EmpHist.EmployeeID

EXECUTE [dbo].[TrnsProc] 1,'tEST'

EXEC [dbo].[TrnsProc] 1,'tEST'

CREATE PROC Testing

AS

BEGIN

SELECT 1

END

APPLY OPERATOR IN SQL SERVER:

SQL Server 2005 introduced the APPLY operator, which is like a [join clause](https://www.mssqltips.com/sqlservertip/1667/sql-server-join-example/) and it allows joining between two table expressions i.e. joining a left/outer table expression with a right/inner table expression. The difference between the join and APPLY operator becomes evident when you have a table-valued expression on the right side and you want this table-valued expression to be evaluated for each row from the left table expression. In this tip I am going to demonstrate the APPLY operator, how it differs from [regular JOINs](https://www.mssqltips.com/sqlservertip/1667/sql-server-join-example/) and some uses.

SELECT

Emp.EmployeeID,

EmpHist.DepartmentID,

Emp.BirthDate,

Emp.Gender FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

GO

SELECT

Emp.EmployeeID,

Temp.DepartmentID,

Emp.BirthDate,

Emp.Gender FROM tblEmployee Emp

CROSS APPLY

(

SELECT \* FROM tblEmployeeHistory EmpHist

WHERE Emp.EmployeeID = EmpHist.EmployeeID

) Temp

GO

SELECT

Emp.EmployeeID,

Emp.BirthDate,

Emp.Gender,

EmpHist.DepartmentID

FROM tblEmployee Emp

LEFT JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

GO

SELECT

Emp.EmployeeID,

Temp.DepartmentID,

Emp.BirthDate,

Emp.Gender FROM tblEmployee Emp

OUTER APPLY

(

SELECT \* FROM tblEmployeeHistory EmpHist

WHERE Emp.EmployeeID = EmpHist.EmployeeID

) Temp

GO

ALTER FUNCTION dbo.Multistatemnt

(

@DepartmentId INT

)

RETURNS @Employee TABLE

(

EmployeeId INT,

BirthDate DATETIME,

MartialStatus VARCHAR(100),

Title VARCHAR(100),

DepartmentId INT

)

AS

BEGIN

INSERT INTO @Employee

SELECT Emp.EmployeeID,Emp.BirthDate,

Emp.MaritalStatus,Emp.Title,EmpHist.DepartmentID

FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

WHERE EmpHist.DepartmentID = 1

ORDER BY DepartmentID

RETURN

END

SELECT Temp.\*,Dept.Name FROM tblDepartment Dept

CROSS APPLY

(

SELECT \* FROM dbo.Multistatemnt(dept.DepartmentID)

) Temp

SELECT Temp.\*,Dept.Name FROM tblDepartment Dept

OUTER APPLY

(

SELECT \* FROM dbo.Multistatemnt(dept.DepartmentID)

) Temp

Lead:

**LAG** and **LEAD** The **LAG** function has the ability to fetch data from a previous row, while **LEAD** fetches data from a subsequent row. Both functions are very similar to each other and you can just replace one by the other by changing the sort order.

LEAD ( scalar\_expression [ ,offset ] , [ default ] )

OVER ( [ partition\_by\_clause ] order\_by\_clause )

LAG (scalar\_expression [,offset] [,default])

OVER ( [ partition\_by\_clause ] order\_by\_clause )

CREATE TABLE Demo

(

ProductId INT,

CustomerId INT,

PurchaseDate DATETIME,

SalesAmount DECIMAL(10,6)

)

INSERT INTO Demo VALUES(1,1,'20180110',34.00)

INSERT INTO Demo VALUES(2,1,'20180210',23.98)

INSERT INTO Demo VALUES(3,2,'20180310',31.12)

INSERT INTO Demo VALUES(4,2,'20180410',67.80)

INSERT INTO Demo VALUES(5,2,'20180510',34.09)

INSERT INTO Demo VALUES(6,3,'20180610',56.10)

INSERT INTO Demo VALUES(7,3,'20180710',76.50)

INSERT INTO Demo VALUES(8,4,'20180810',98.09)

SELECT \* FROM Demo

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAG(SalesAmount) OVER(ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAG(SalesAmount,1,0) OVER(ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAG(SalesAmount,2,0) OVER(ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAG(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LEAD(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LEAD(SalesAmount) OVER(ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

First Value & Last Value in SQL Server:

Returns the first value in an ordered set of values

Syntax:

FIRST\_VALUE ( [scalar\_expression ] ) [ IGNORE NULLS | RESPECT NULLS ]

OVER ( [ partition\_by\_clause ] order\_by\_clause [ rows\_range\_clause ] )

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

FIRST\_VALUE(SalesAmount) OVER(ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

FIRST\_VALUE(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

FIRST\_VALUE(SalesAmount) OVER(ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

FIRST\_VALUE(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId ) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

FIRST\_VALUE(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS PrevSalesAmount

FROM LeadDemo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAST\_VALUE(SalesAmount) OVER(ORDER BY ProductId

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS LastAmount

FROM LeadDemo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAST\_VALUE(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS LastAmount

FROM LeadDemo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

SUM(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS LastAmount

FROM LeadDemo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAST\_VALUE(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

) AS PrevSalesAmount

FROM Demo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

MAX(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY SalesAmount

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS LastAmount

FROM LeadDemo

GO

SELECT

ProductId,

CustomerId,

PurchaseDate,

SalesAmount,

LAST\_VALUE(SalesAmount) OVER(PARTITION BY CustomerId ORDER BY ProductId

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

) AS PrevSalesAmount

FROM Demo

GO

SELECT

\*,

AVG(SalesAmount)

OVER(PARTITION BY CustomerId ORDER BY ProductId

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

) AS LastValue

FROM LeadDemo

SELECT

\*,

MIN(SalesAmount)

OVER(PARTITION BY CustomerId ORDER BY ProductId

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS LastValue

FROM LeadDemo

Views in SQL Server:

A VIEW in SQL Server is like a virtual table that contains data from one or multiple tables. It does not hold any data and does not exist physically in the database. Similar to a SQL table, the view name should be unique in a database. It contains a set of predefined SQL queries to fetch data from the database. It can contain database tables from single or multiple databases as well.

CREATE VIEW Name AS

Select column1, Column2...Column N From tables

Where conditions;

CREATE VIEW vDemo

AS

SELECT

EmployeeID,

BirthDate,

Gender,

MaritalStatus,

Title

FROM tblEmployee

GO

ALTER VIEW vDemo

AS

SELECT

Emp.EmployeeID,

BirthDate,

Gender,

MaritalStatus,

Title,

EmpHist.DepartmentID

FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

GO

SELECT \* FROM vDemo

SELECT EmployeeId,BirthDate,Gender,Title FROM vDemo

ALTER VIEW vDemo

AS

SELECT

Emp.EmployeeID,

BirthDate,

Gender,

MaritalStatus,

Title,

EmpHist.DepartmentID

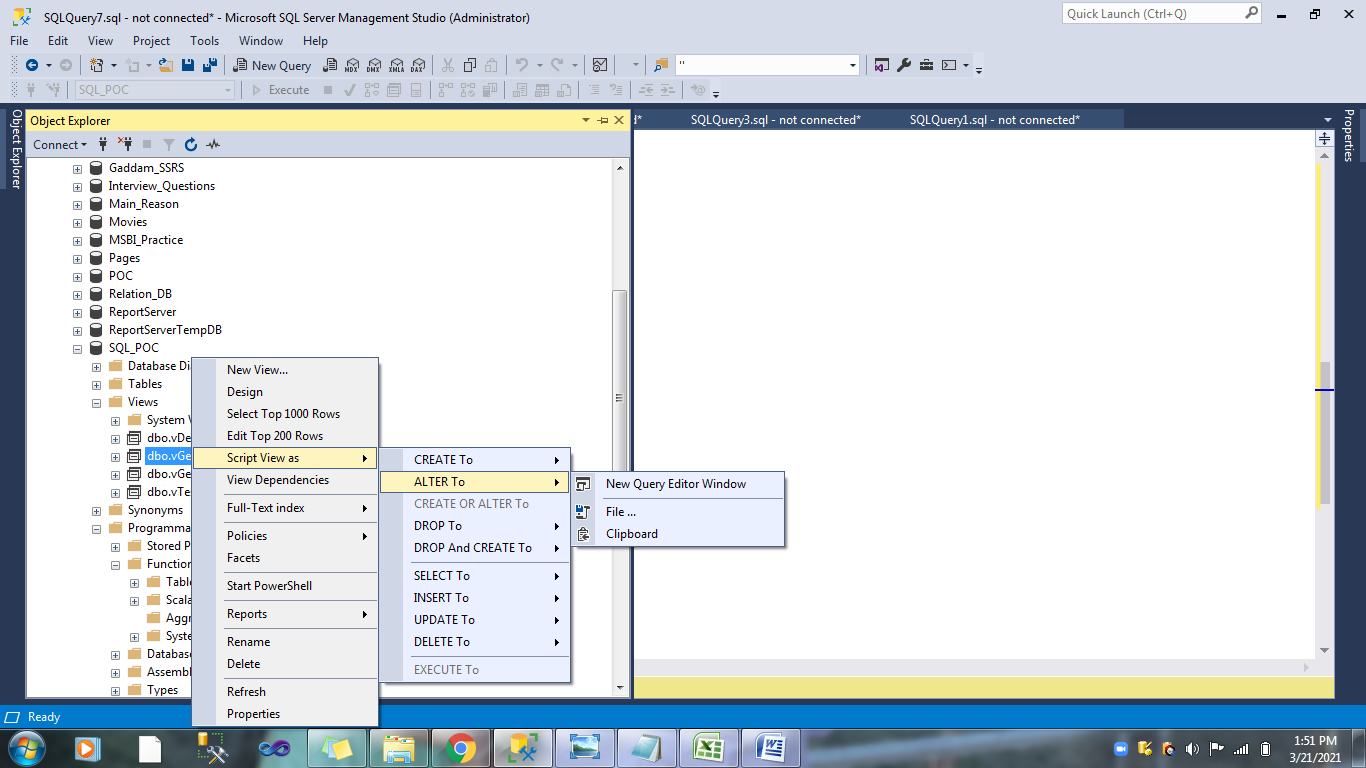
FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

WHERE Title = 'Production Technician - WC30'

GO



If you want to add another Table as join we can to the View Statement.

SELECT EmployeeId,BirthDate,Gender,Title,Dept.Name FROM vDemo

INNER JOIN tblDepartment Dept

ON vDemo.DepartmentId = Dept.DepartmentID

Sp\_HelpText We Can Understand what is behind the code for Creating the View.

And we can verify the code by using GUI also.

CREATE TABLE DemoTest

(

ID INT,

Name VARCHAR(100)

)

INSERT INTO DemoTest Values(1,'Venki')

INSERT INTO DemoTest Values(2,'Ramu')

ALTER TABLE DemoTest

ADD City Varchar(100)

CREATE VIEW vTest

AS

Select \* FROM DemoTest

SELECT \* FROM vTest

Sp\_Refreshview 'vTest'

ALTER VIEW vTest

WITH SCHEMABINDING

AS

Select id,Name FROM dbo.DemoTest

Here we have to specify Individual Column Names and also specify the SchemanName before the Table Name.

ALTER TABLE DemoTest

ALTER COLUMN Name Varchar(50)

When we have to try to Change the Column length of the existing column in the view you will get an error.

ALTER VIEW vTest

WITH SCHEMABINDING,Encryption

AS

Select id,Name FROM dbo.DemoTest

Pivot and UN Pivot in SQL Server:

Learn how to use Pivot in SQL and Unpivot in SQL. Pivot and Unpivot in SQL are two relational operators that are used to convert a table expression into another. Pivot in SQL is used when we want to transfer data from row level to column level and Unpivot in SQL is used when we want to convert data from column level to row level. PIVOT and UNPIVOT relational operators are used to generate a multidimensional reporting.

In this article, we will discuss both Pivot and Unpivot operators in a SQL Server database and how to create interactive tables that quickly combines and compares large amount of data.

CREATE TABLE Grades(

[Student] VARCHAR(50),

[Subject] VARCHAR(50),

[Marks] INT

)

GO

INSERT INTO Grades VALUES

('Jacob','Mathematics',100),

('Jacob','Science',95),

('Jacob','Geography',90),

('Amilee','Mathematics',90),

('Amilee','Science',90),

('Amilee','Geography',100)

GO

SELECT \* FROM Grades

SELECT Student,Mathematics AS Maths,Science,Geography FROM

(

SELECT Student,Subject,Marks FROM Grades

) Temp

PIVOT

(

SUM(Marks)

FOR Subject IN

(

[Mathematics],

[Science],

[Geography]

)

) AS TempPivot

CREATE TABLE Employee

(

Name [nvarchar](max),

[Year] [int] ,

Sales [int]

)

INSERT INTO Employee

SELECT 'Pankaj',2010,72500 UNION ALL

SELECT 'Rahul',2010,60500 UNION ALL

SELECT 'Sandeep',2010,52000 UNION ALL

SELECT 'Pankaj',2011,45000 UNION ALL

SELECT 'Sandeep',2011,82500 UNION ALL

SELECT 'Rahul',2011,35600 UNION ALL

SELECT 'Pankaj',2012,32500 UNION ALL

SELECT 'Pankaj',2010,20500 UNION ALL

SELECT 'Rahul',2011,200500 UNION ALL

SELECT 'Sandeep',2010,32000

SELECT \* FROM Employee

SELECT \* FROM

(

SELECT Name,[Year],Sales FROM Employee

) AS Temp

PIVOT

(

SUM(Sales)

FOR [Year] IN

(

[2010],

[2011],

[2012]

)

) AS TempPivot

GO

CREATE TABLE Students

(

Id INT PRIMARY KEY IDENTITY,

StudentName VARCHAR (50),

Course VARCHAR (50),

Score INT

)

GO

INSERT INTO Students VALUES ('Sally', 'English', 95 )

INSERT INTO Students VALUES ('Sally', 'History', 82)

INSERT INTO Students VALUES ('Edward', 'English', 45)

INSERT INTO Students VALUES ('Edward', 'History', 78)

Select \* FROM Students

SELECT \* FROM

(

SELECT StudentName,Course,Score FROM

Students

) Temp

PIVOT

(

SUM(Score)

FOR Course IN

(

English,

History

)

) AS TempPivot

GO

**UNPIVOT IN SQL Server:**

CREATE TABLE Students

(

Id INT PRIMARY KEY IDENTITY,

StudentName VARCHAR (50),

Math INT,

English INT,

History INT,

Science INT

)

GO

INSERT INTO Students VALUES ('Sally', 78, 85, 91, 76 )

INSERT INTO Students VALUES ('Edward', 87, 90, 82, 87)

Select \* FROM Students

SELECT StudentName, Course, SourceData

FROM Students

UNPIVOT

(

SourceData

FOR Course in (Math, English, History, Science)

) AS SchoolUnpivot

**Dynamic SQL Server:**

SELECT \* FROM tblEmployee

'SELECT \* FROM tblEmployee'

EXECUTE ('SELECT \* FROM tblEmployee')

EXEC ('SELECT \* FROM tblEmployee')

EXEC sp\_executesql N'SELECT \* FROM tblEmployee';

We have to give N as a prefix for the sp\_ExecuteSQL otherwise it will throw the error. By using Sp\_ExecuteSQL We can reuse the caching mechanisum but when we use Exec every time it will it will fetch data from tables not from the cache,

DECLARE @TableName VARCHAR(100)

DECLARE @SQLString NVARCHAR(MAX)

SET @TableName = 'tblDepartment'

SET @SQLString = N'SELECT \* FROM ' + @TableName

EXEC Sp\_executesql @SQLString

GO

ALTER PROCEDURE DemoTest1

(

@TableName VARCHAR(100)

)

AS

BEGIN

DECLARE @SQLString NVARCHAR(MAX)

SET @SQLString = N'SELECT \* FROM ' + @TableName

EXEC Sp\_executesql @SQLString

END

EXECUTE DemoTest1 'tblEmployeeHistory'

DECLARE @TOP INT

SET @TOP = 2

DECLARE @TableName VARCHAR(100)

SET @TableName = 'tblEmployee'

DECLARE @SQLString NVARCHAR(100)

SET @SQLString = 'SELECT TOP ' + CAST(@TOP as nvarchar(100)) + '\* FROM ' + @TableName

EXEC sp\_ExecuteSQL @SQLString

CREATE PROCEDURE TopRecords

(

@RecordCount INT,

@TableName VARCHAR(100)

)

AS

BEGIN

DECLARE @SQLString NVARCHAR(MAX)

SET @SQLString = 'SELECT TOP ' + CAST(@RecordCount AS nvarchar(100)) +

' \* FROM ' + @TableName

EXEC sp\_executesql @SQLString

END

Execute TopRecords 23,'tblDepartment'

ALTER PROCEDURE DynamicSQL

(

@EmployeeID VARCHAR(100)

)

AS

BEGIN

DECLARE @SQLString NVARCHAR(MAX)

SET @SQLString = 'SELECT \* FROM tblEmployee WHERE EmployeeId IN ('+ @EmployeeID +')

ORDER BY EmployeeId DESC'

EXEC sp\_Executesql @SQLString

END

EXECUTE DynamicSQL '1,2,3,4,5,66,8'

ALTER PROCEDURE DynamicSQL

(

@EmployeeID VARCHAR(100),

@ManagerID VARCHAR(100)

)

AS

BEGIN

DECLARE @SQLString NVARCHAR(MAX)

SET @SQLString = 'SELECT \* FROM tblEmployee WHERE EmployeeId IN ('+ @EmployeeID +')

AND ManagerId IN ('+ @ManagerID + ')

ORDER BY EmployeeId DESC'

PRINT @SQLString

EXEC sp\_Executesql @SQLString

END

EXECUTE DynamicSQL '1,2,3,4,9,11,13','3,16,12'

Dynamic SQL Class Continution

ALTER PROCEDURE DynamicDemo

(

@ManagerId INT = NULL,

@Gender VARCHAR(100) = NULL,

@MaritalStatus VARCHAR(100) = NULL,

@Title VARCHAR(100) = NULL

)

AS

BEGIN

DECLARE @SQLString NVARCHAR(1000)

SET @SQLString = 'SELECT \* FROM tblEmployee WHERE 1 = 1'

IF @ManagerId IS NOT NULL

SET @SQLString = @SQLString + ' AND ManagerId = ''' + CAST(@ManagerId AS varchar(100)) + ''''

IF @Gender IS NOT NULL

SET @SQLString = @SQLString + ' AND Gender = ''' + @Gender + ''''

IF @MaritalStatus IS NOT NULL

SET @SQLString = @SQLString + ' AND MaritalStatus = ''' + @MaritalStatus + ''''

IF @Title IS NOT NULL

SET @SQLString = @SQLString + ' AND Title = ''' + @Title + ''''

PRINT (@SQLString)

EXECUTE sp\_executesql @SQLString

END

IF OBJECT\_ID('tempdb..#Temp') IS NOT NULL

DROP TABLE #Temp

SELECT

DISTINCT

Year

INTO #Temp

FROM Employee

DECLARE @YearList VARCHAR(100) = ''

DECLARE @SQLString NVARCHAR(MAX)

SELECT @YearList = @YearList + QUOTENAME([Year]) + ',' FROM #Temp

SELECT @YearList

SELECT @YearList = LEFT(@YearList,LEN(@YearList)-1)

SET @SQLString = '

SELECT \* FROM

(

SELECT DISTINCT

Name,Year,Sales

FROM Employee

) Temp

PIVOT

(

SUM(Sales)

FOR Year IN

('

+ @YearList +

')

) AS PivotTable'

PRINT (@SQLString)

EXECUTE sp\_executesql @SQLString

ALTER PROCEDURE Testing111

(

@Name VARCHAR(100)

)

AS

BEGIN

IF OBJECT\_ID('tempdb..#TempData1') IS NOT NULL

DROP TABLE #TempData1

DECLARE @SQLString NVARCHAR(MAX)

SELECT

Emp.EmployeeID,Emp.ManagerID,Emp.BirthDate,

Emp.Gender,EmpHist.DepartmentID

INTO #TempData1

FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

SET @SQLString = ' SELECT T.DepartmentID,Dept.Name,

Dept.GroupName

FROM #TempData1 T

INNER JOIN tblDepartment Dept

ON T.DepartmentID = Dept.DepartmentID

WHERE Dept.Name LIKE ''' + @Name + '%'''

--WHERE T.DepartmentId = ' + CAST(@DepartmentId AS varchar(100)) +

--'AND Dept.Name = ''' + @Name + '''' + ' ORDER BY Name '

-- WHERE Dept.Name LIKE ''%' + @Name + ''''

WHERE Dept.Name LIKE ''%' + @Name + '%'''

PRINT(@SQLString)

EXECUTE sp\_executesql @SQLString

END

**How to Insert the Data from Stored procedure to Temp Table/Physical Table:**

CREATE PROCEDURE GetProceCount

AS

BEGIN

SELECT

Emp.EmployeeID,

Emp.Title,

Emp.Gender,

Emp.MaritalStatus,

EmpHist.DepartmentID

FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

END

CREATE TABLE #Test

(

EmpId INT,

EmpName VARCHAR(100),

Gender VARCHAR(100),

MartialStatus VARCHAR(100),

DepartmentId INT

)

INSERT INTO #Test

EXECUTE GetProceCount

TRUNCATE VS DELETE:

Basically if we want to delete entire data from tables we can use Truncate but in case if you want to delete specific records or based on some conditions we can go for Delete Statement.

Also Truncate will reset Identity values where as delete statement cannot reset the Identity values.

CREATE TABLE DemoDelete

(

ID INT IDENTITY(1,1),

Name VARCHAR(100),

Salary INT

)

INSERT INTO DemoDelete VALUES ('Somu',2300)

INSERT INTO DemoDelete VALUES ('EE',45)

INSERT INTO DemoDelete VALUES ('Ranj',23)

INSERT INTO DemoDelete VALUES ('Ramu',988)

SELECT \* FROM DemoDelete

DELETE FROM DemoDelete WHERE Id = 5

TRUNCATE TABLE DemoDelete WHERE Id = 6

**How to call one Procedure another Procedure:**

CREATE PROCEDURE GetSampleData

AS

BEGIN

SELECT Emp.EmployeeID,Emp.Title,

Emp.Gender,Emp.BirthDate,EmpHist.DepartmentID

FROM tblEmployee Emp

INNER JOIN tblEmployeeHistory EmpHist

ON Emp.EmployeeID = EmpHist.EmployeeID

END

GO

ALTER PROCEDURE GetDepartmentData

AS

BEGIN

CREATE TABLE #TempData

(

EmpId INT,

Title VARCHAR(100),

Gender VARCHAR(100),

BirthDate DATETIME,

DepartmentId INT

)

INSERT INTO #TempData

EXECUTE GetSampleData

SELECT T.\*,Dept.Name,Dept.GroupName FROM #TempData T

INNER JOIN tblDepartment Dept

ON T.DepartmentId = Dept.DepartmentID

END

EXECUTE GetDepartmentData

**Index in SQL Server:**

CREATE TABLE DummyData

(

ID INT,

Name VARCHAR(100)

)

INSERT INTO DummyData Values(1,'Venki')

INSERT INTO DummyData Values(3,'Ramu')

INSERT INTO DummyData Values(2,'Srinu')

INSERT INTO DummyData Values(5,'Pavan')

INSERT INTO DummyData Values(4,'Shek')

SELECT \* FROM DummyData

By Default data will be inserting as it is means that it will not do the ordering by default, if you want to do that we have to specify Order by keyword. Using Index we can do that without specify order by to

SELECT \* FROM DummyData

One of the most important routes to high performance in a SQL Server database is the index. Indexes speed up the querying process by providing swift access to rows in the data tables, similarly to the way a book’s index helps you find information quickly within that book.

We have different index in SQL Server

1. Clustered Index

2. NonClustered Index

We can create only one Clustered index per table where as we can create 999 Non Clustered indexes per tables.

CREATE UNIQUE CLUSTERED INDEX CLS\_DummyData\_Id

ON DummyData(Id)

CREATE UNIQUE NONCLUSTERED INDEX NCLS\_DummyData\_Ida

ON DummyData(Name)

CREATE UNIQUE NONCLUSTERED INDEX NCLS\_DummyData\_ID\_Name\_Order

ON DummyData(Id,Name ASC)

DROP Table DummyData

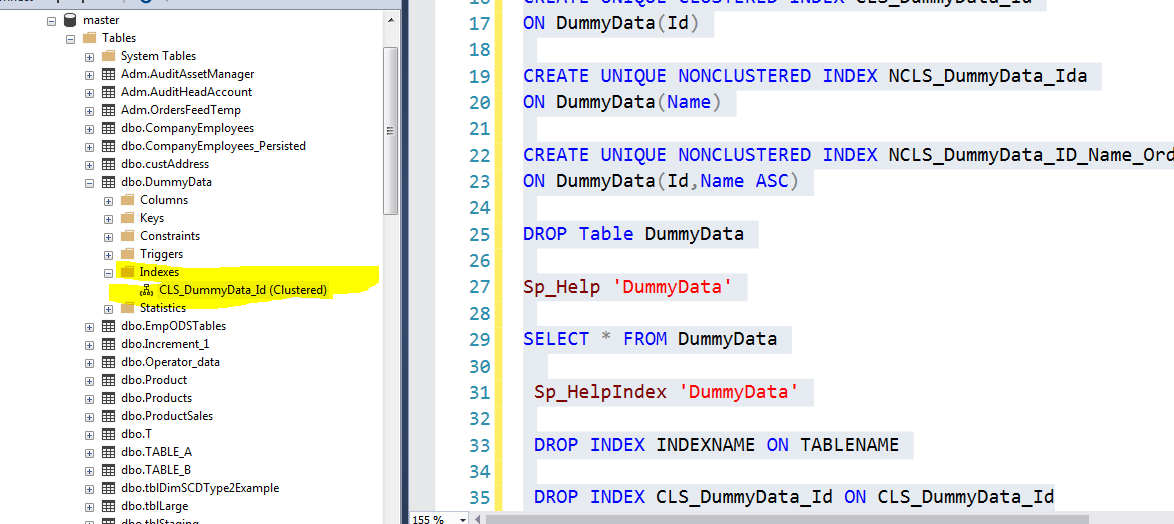
Sp\_Help 'DummyData'

SELECT \* FROM DummyData

Sp\_HelpIndex 'DummyData'

DROP INDEX INDEXNAME ON TABLENAME

DROP INDEX CLS\_DummyData\_Id ON CLS\_DummyData\_Id



## What is a Clustered index?

Cluster index is a type of index which sorts the data rows in the table on their key values. In the Database, there is only one clustered index per table.

A clustered index defines the order in which data is stored in the table which can be sorted in only one way. So, there can be an only a single clustered index for every table. In an RDBMS, usually, the primary key allows you to create a clustered index based on that specific column.

## What is Non-clustered index?

A Non-clustered index stores the data at one location and indices at another location. The index contains pointers to the location of that data. A single table can have many non-clustered indexes as an index in the non-clustered index is stored in different places.

For example, a book can have more than one index, one at the beginning which displays the contents of a book unit wise while the second index shows the index of terms in alphabetical order.

A non-clustering index is defined in the non-ordering field of the table. This type of indexing method helps you to improve the performance of queries that use keys which are not assigned as a primary key. A non-clustered index allows you to add a unique key for a table.

## KEY DIFFERENCE

* Cluster index is a type of index that sorts the data rows in the table on their key values whereas the Non-clustered index stores the data at one location and indices at another location.
* Clustered index stores data pages in the leaf nodes of the index while Non-clustered index method never stores data pages in the leaf nodes of the index.
* Cluster index doesn’t require additional disk space whereas the Non-clustered index requires additional disk space.
* Cluster index offers faster data accessing, on the other hand, Non-clustered index is slower.

## Characteristic of Clustered Index

* Default and sorted data storage
* Use just one or more than one columns for an index
* Helps you to store Data and index together
* Fragmentation
* Operations
* Clustered index scan and index seek
* Key Lookup

## Characteristics of Non-clustered Indexes

* Store key values only
* Pointers to Heap/Clustered Index rows
* Allows Secondary data access
* Bridge to the data
* Operations of Index Scan and Index Seek
* You can create a nonclustered index for a table or view
* Every index row in the nonclustered index stores the nonclustered key value and a row locator

## Advantages of Clustered Index

The pros/benefits of the clustered index are:

* Clustered indexes are an ideal option for range or group by with max, min, count type queries
* In this type of index, a search can go straight to a specific point in data so that you can keep reading sequentially from there.
* Clustered index method uses location mechanism to locate index entry at the start of a range.
* It is an effective method for range searches when a range of search key values is requested.
* Helps you to minimize page transfers and maximize the cache hits.

## Advantages of Non-clustered index

Pros of using non-clustered index are:

* A non-clustering index helps you to retrieves data quickly from the database table.
* Helps you to avoid the overhead cost associated with the clustered index
* A table may have multiple non-clustered indexes in RDBMS. So, it can be used to create more than one index.

## Disadvantages of Clustered Index

Here, are cons/drawbacks of using clustered index:

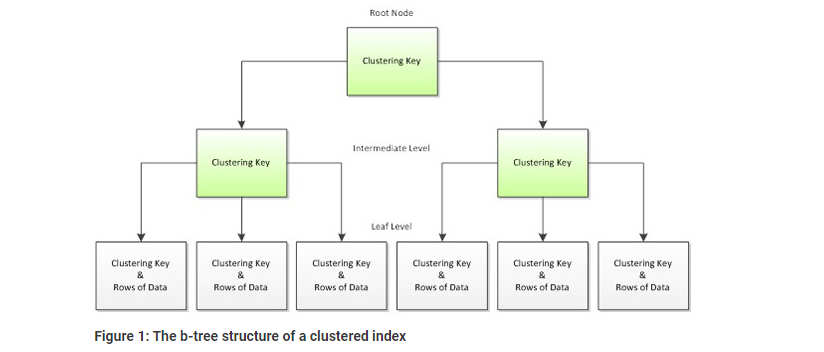
* Lots of inserts in non-sequential order
* A clustered index creates lots of constant page splits, which includes data page as well as index pages.
* Extra work for SQL for inserts, updates, and deletes.
* A clustered index takes longer time to update records when the fields in the clustered index are changed.
* The leaf nodes mostly contain data pages in the clustered index.

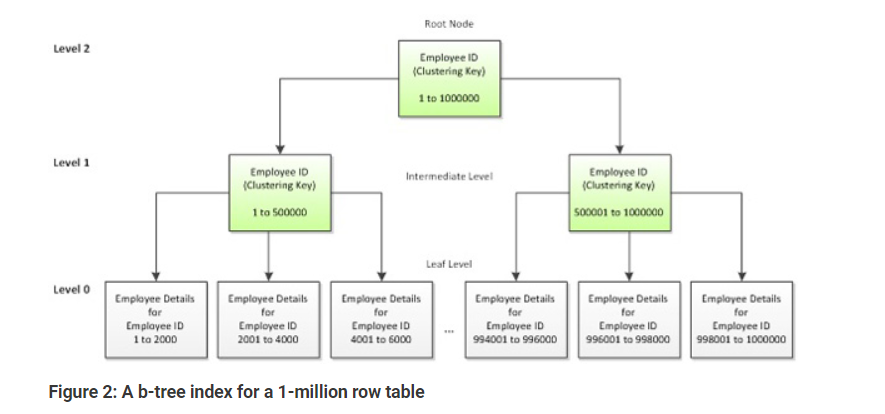
## Disadvantages of Non-clustered index

Here, are cons/drawbacks of using non-clustered index:

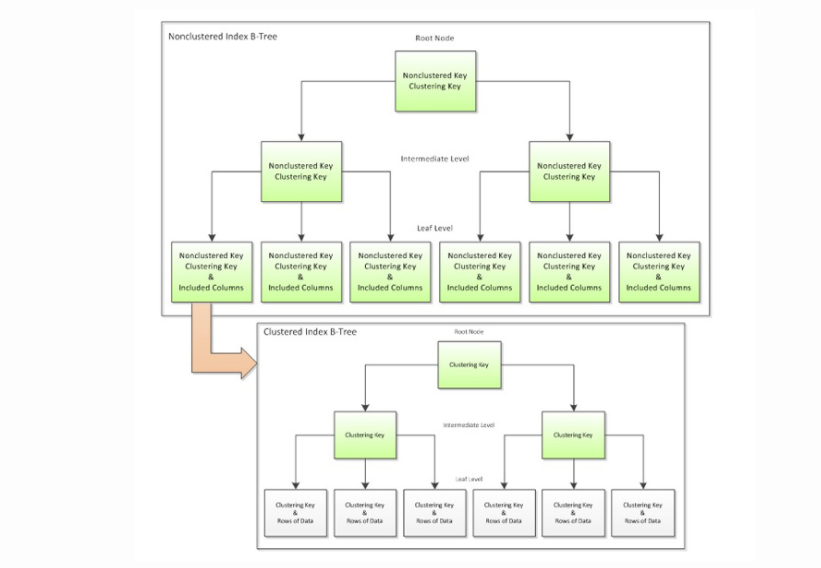
* A non-clustered index helps you to stores data in a logical order but does not allow to sort data rows physically.
* Lookup process on non-clustered index becomes costly.
* Every time the clustering key is updated, a corresponding update is required on the non-clustered index as it stores the clustering key.

**Clustered Index Structure**





**Non Clustered Index Structure**



**Magic Tables in SQL Server:**

There are Magic Tables (virtual tables) in SQL Server that hold the temporal information of recently inserted and recently deleted data in the virtual table. INSERTED and DELETED are two types of magic tables in SQL Server. An INSERTED magic table is populated with INSERT and UPDATE operations and DELETED magic table is populated with UPDATE and DELETE operations.

The INSERTED magic table stores the before version of the row, and the DELETED table stores the after version of the row for any INSERT, UPDATE, or DELETE operations.

A magic table can be utilized in INSERT, UPDATE, and DELETE activity with the table in a trigger, which is the common understanding of people. SQL Server uses magic tables outside the TRIGGER also for a lot of other purposes too. Use of Magic tables in SQL Server with the usual update statement lessens the information dependency and makes your information consistent with your transaction.

**INSERT**: The INSERTED magic table will have the newly inserted rows on the top in the table with an insert operation. It can be used to manage an audit of the table to another history table.

**DELETE**: The DELETED magic table will have the recently deleted rows on the top in the table with a delete operation. It can be used to manage a previous version of the row for the audit purpose in the history table.

**UPDATE**: Both INSERTED and DELETED virtual tables will be part of an update statement. Update statement returns deleted magic table with the previous version of the row and the inserted magic table with the new version of a row, which is going to be replaced or updated with the earlier values in the table. The important thing is that whenever users perform the update statement inside the trigger or outside the trigger, INSERTED and DELETED magic tables are being used.

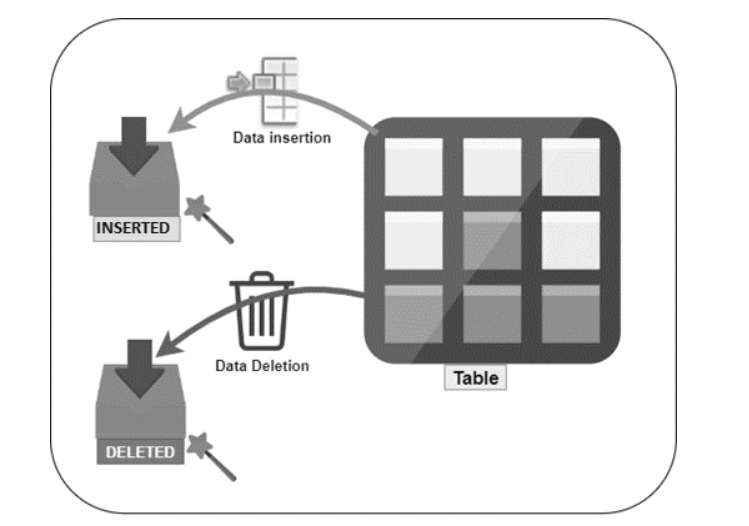
A common use of Magic tables in SQL Server is the DML (Data Manipulation Language) trigger. SQL Server DML trigger allows using these two virtual tables INSERTED and DELETED. The ideal use of the trigger is auditing and managing a before and after version of the table row on INSERT, UPDATE, or DELETE operation within the transaction statement. Even users can write data manipulation logic as well with these magic tables inside the trigger.

A magic table is stored in the temp DB. Therefore, whenever you use the magic tables in SQL Server with the query statement, tempdb will come in the picture.

Whenever the magic table is utilizing with a query statement in the transaction, tempdb will be affected by that statement. Below are the limitations of the magic table compare to the actual temp table (# table)

Users can not create any index or apply any constraint on the magic tables in SQL Server

They cannot be altered because the purpose of the magic table is to audit the information in the system



## SQL Server RAISEERROR statement overview

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.

The following illustrates the syntax of the RAISERROR statement:

RAISERROR ( { message\_id | message\_text | @local\_variable }

{ ,severity ,state }

[ ,argument [ ,...n ] ] )

[ **WITH** **option** [ ,...n ] ];

**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

SELECT 1/0

END TRY

BEGIN CATCH

SELECT ERROR\_MESSAGE()

RAISERROR('We will get Divide by zero error while executing the Program',16,1)

END CATCH

**Triggers in SQL Server:**

A SQL Server trigger is a piece of procedural code, like a stored procedure which is only executed when a given event happens. There are different types of events that can fire a trigger. Just to name you a few, the insertion of rows in a table, a change in a table structure and even a user logging into a SQL Server instance.

There are three main characteristics that make triggers different than stored procedures:

* Triggers cannot be manually executed by the user.
* There is no chance for triggers to receive parameters.
* You cannot commit or rollback a transaction inside a trigger.

The fact that it's impossible to use parameters on triggers is not a limitation to receive information from the firing event. As you will see further on, there are alternatives to obtain information about the firing event.

**Classes of SQL Server Triggers**

There are two classes of triggers in SQL Server:

DDL (Data Definition Language) triggers. This class of triggers fires upon events that change the structure (like creating, modifying or dropping a table), or in certain server related events like security changes or statistics update events.

DML (Data Modification Language) triggers. This is the most used class of triggers. In this case the firing event is a data modification statement; it could be an insert, update or delete statement either on a table or a view.

CREATE TABLE Employee

(

Empid int Primary key,

EmpName nvarchar(20),

Salary int,

DeptNo int

)

Go

INSERT INTO Employee values(1,'John',2000,3)

INSERT INTO Employee values(2,'Rao',7654,2)

INSERT INTO Employee values(3,'Malli',5454,1)

INSERT INTO Employee values(4,'Smith',7000,5)

INSERT INTO Employee values(5,'Aravinda',3400,3)

INSERT INTO Employee values(6,'Hanu',5600,2)

INSERT INTO Employee values(7,'Subbu',4300,3)

INSERT INTO Employee values(8,'Ravindra',2090,4)

INSERT INTO Employee values(9,'Srinu',2011,5)

INSERT INTO Employee values(10,'Ramu',5500,1)

SELECT \*

INTO tblDemoEmployee

FROM Employee

SELECT \* FROM tblDemoEmployee

SELECT \* FROM Employee

INSERT INTO Employee values(11,'Pavani',5500,4)

SELECT \* FROM Employee Emp

LEFT JOIN tblDemoEmployee Demo

ON Emp.Empid = Demo.Empid

WHERE Demo.Empid IS NULL

CREATE TRIGGER Emp\_Insert\_Trg

ON Employee

AFTER INSERT

AS

BEGIN

INSERT INTO tblDemoEmployee

SELECT \* FROM inserted

END

INSERT INTO Employee values(15,'Ramu',434,2)

SELECT \* FROM Employee

SELECT \* FROM tblDemoEmployee

ALTER TABLE Employee

DISABLE TRIGGER Emp\_Insert\_Trg

ALTER TABLE Employee

ENABLE TRIGGER Emp\_Insert\_Trg

DROP TRIGGER Emp\_Insert\_Trg

ALTER TRIGGER Insert\_Trg

ON Employee

AFTER INSERT,UPDATE,DELETE

AS

BEGIN

DECLARE @DateDiff INT

SET @DateDiff = DATEPART(HH,GETDATE())

IF @DateDiff BETWEEN 1 AND 23

ROLLBACK

RAISERROR('We cannot perform operation in between this timing',16,1)

END

INSERT INTO Employee values(16,'Ramu',434,2)

DELETE FROM Employee WHERE Empid = 1

UPDATE Employee

SET EmpName = 'Update doing'

WHERE Empid = 1

ALTER TABLE Employee

DISABLE TRIGGER Insert\_Trg

CREATE TRIGGER Delete\_Trg

ON Employee

AFTER DELETE

AS

BEGIN

DECLARE @EmployeeId INT

SET @EmployeeId = (SELECT Empid FROM deleted)

IF @EmployeeId = 1

BEGIN

ROLLBACK

RAISERROR('You cannot do this operaton',16,1)

END

END

Delete FROM Employee WHERE Empid = 2

SELECT \* FROM Employee

GO

CREATE TRIGGER Test\_Trigger

ON Employee

INSTEAD OF INSERT

AS

BEGIN

INSERT INTO Employee

SELECT Empid,UPPER(EmpName),Salary \* 0.1, DeptNo + 5 FROM inserted

END

Select \* FROM Employee

INSERT INTO Employee values(18,'Ramu',434,2)

SELECT \* FROM Employee WHERE Empid = 18

GO

CREATE TRIGGER UpdateTrigger

ON Employee

AFTER UPDATE

AS

BEGIN

DECLARE @OldValue VARCHAR(100)

DECLARE @NewValue VARCHAR(100)

SELECT @OldValue = (SELECT EmpName FROM deleted)

SELECT @NewValue = (SELECT EmpName FROM inserted)

SELECT @OldValue AS OldValue, @NewValue AS NewValue

END

SELECT \* FROM Employee

UPDATE Employee

SET EmpName = 'Gaddam Venki'

WHERE Empid = 1