SQL Server

SQL

What is SQL Server?

It is a software, developed by Microsoft, which is implemented from the specification of RDBMS.

It is also an ORDBMS.

It is platform dependent.

It is both GUI and command based software.

It supports SQL (SEQUEL) language which is an IBM product, non-procedural, common database and case insensitive language.

MS SQL Server is a relational database management system (RDBMS) developed by Microsoft. This product is built for the basic function of storing retrieving data as required by other applications. It can be run either on the same computer or on another across a network.

Create Alter and Delete Database

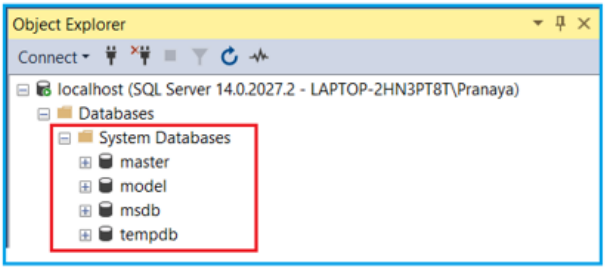
Different Types of Database in SQL Server.

In SQL Server we are going to interact with 2 types of databases such as

1. System databases
2. User databases

System Databases in SQL Server:

The databases which are created and managed by the SQL Server itself called System databases. SQL Server has four system databases as shown in the below image.



Let us discuss the role and responsibilities of each of the above databases.

Master database: This database is used to store all system-level information such as system id, culture, server id no, server version, server culture, etc
Model database: The model database will act as a template for creating new databases under a server environment.
Msdb (Microsoft database): Microsoft database will store jobs and alerts information i.e. backup file information.
Tempdb database: It is a temporary database location that is allocated by the server when the user connected to the SQL Server for storing temporary table information.

Note: Once you disconnected from the SQL Server, then the temporary database location will be destroyed automatically. The above system databases are maintained and managed by the system by default.

User Databases in SQL Server:

The databases which are created and managed by the user are called User Databases.

the user databases can be created, altered and dropped in two ways

1. Graphically using SQL Server Management Studio (SSMS) or
2. Using a Query

Creating SQL Server Database Graphically:

1. Right Click on the Databases folder in the Object Explorer
2. Select New Database
3. In the New Database dialog box, enter the Database name and click the OK button as shown in the below image.

How to create SQL Server Database using Query?

The syntax for creating a database in SQL Server: Create database <Database Name>
Example: Create database TestDB

Select the above query and Click on either Execute option or F5 key for execution. Whether we create a database graphically using the designer window or using a query, **the following 2 files get generated.**

.MDF file: Master Data File (Contains actual data). This file will store all Tables data and will be saved with an extension of .mdf (master data file)

.LDF file: Transaction Log file (Used to recover the database). This file will store transaction Query information (insert, update, delete. Create, etc) and saved with an extension of .ldf (log data file)

Note: The above two files are used for transferring the required database from one system to another system or from one location to another location. The Root Location of .mdf and .ldf files: C:\Program Files\Microsoft SQL Server\MSSQL10.SQLEXPRESS\MSSQL\DATA

How to Rename a database in SQL Server?

Once you create a database, then you can modify the name of the database using the Alter command as shown below.
Alter database DatabaseName Modify Name = NewDatabaseName

Alternatively, you can also use the following system-defined stored procedure to change the name.
Execute sp_renameDB 'OldDatabaseName','NewDatabaseName'

Example

```
create database DBTesting


--Alter Database name

Alter database TestDb1 Modify Name = NewName
```

How to Delete or Drop a database in SQL Server?

In order to delete or drop a database in SQL Server, you need to use **DROP command.**

Drop Database DatabaseThatYouWantToDrop

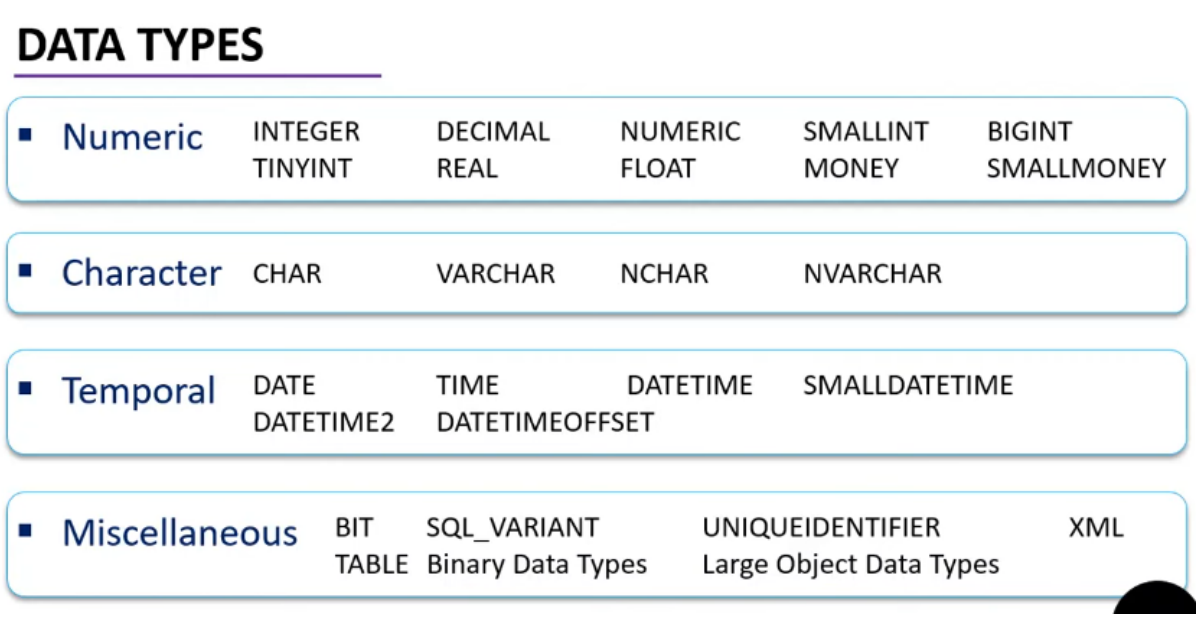
Whenever you drop a database in SQL Server, internally it deletes the LDF and MDF files. You cannot drop a database if it is currently in use and at that time you will get an error stating – Cannot drop database "DatabaseName" because it is currently in use. So, if other users are connected to your database, then **first you need to put the database in single-user mode and then drop the database.** In order to put the database in single-user mode, you need to use the following command.

Alter Database DatabaseName Set SINGLE_USER With Rollback Immediate

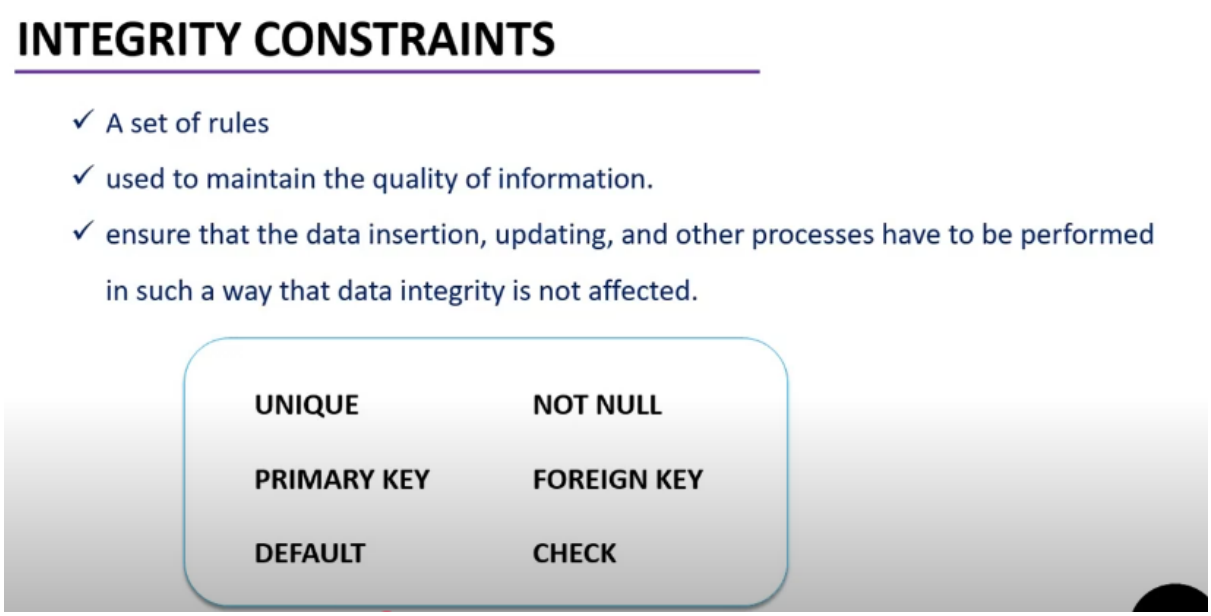
What is SQL?

1. It is a non-procedural language that is used to communicate with any database such as Oracle, SQL Server, etc.
2. This Language was developed by the German Scientist Mr. E.F.Codd in 1968
3. ANSI (American National Standard Institute) approved this concept and in 1972 SQL was released into the market
4. SQL is also called **Sequel** it stands for Structured English Query Language,
5. The sequel will provide a common language interface facility it means that a sequel is a language that can communicate with any type of database such as SQL Server, Oracle, MySQL, Sybase, BD2, etc.
6. SQL is not a case-sensitive language it means that all the commands of SQL are not case sensitive
7. Every command of SQL should end with a semicolon (;) (It is optional for SQL Server)

SQL Data Types in SQL Server



Integrity Constraints



Primary Key and Foreign Key in SQL Server

What is Primary Key in SQL Server?

The Primary Key in SQL Server is **the combination of Unique and Not Null Constraint**. That means it will **not allow either NULL or Duplicate values into a column or columns on which the primary key constraint is applied**. Using the primary key value we should uniquely identify a record.

A table should contain only 1 Primary Key which can be either on a single or multiple columns i.e. the composite primary key. A table should have a primary key to uniquely identify each record. The Primary Key constraint can be applied to any data type like integer, character, decimal, money, etc.

Understanding the Primary Key Constraint in SQL Server:

We cannot apply more than one primary key in a table. Let us prove this.

But We can create one primary key in a table.

Adding Duplicate value in the Primary Key column:

it will give us the following error. That means the Primary Key constraint will not accept duplicate values in it.

Inserting NULL in Primary Key Column in SQL Server:

it gives us the following error stating cannot insert null value which proves that Primary Key will not accept NULL.

Example:

```
CREATE TABLE Branches
(
    Bcode INT PRIMARY KEY,
    Bname VARCHAR(40),
    Bloc  CHAR(40)
)
```

```
-- Insert data into table

INSERT Branches VALUES (1021, 'SBI', 'SRNAGAR') -- It executed successfully.

-- Insert duplicate value in primary key

INSERT Branches VALUES (1021,'SBI','SRNAGAR') -- Not Allowed

-- Insert null value in primary key

INSERT Branches VALUES (NULL, 'SBI', 'SRNAGAR')
```

Composite Primary Key

What is the Composite Primary key in SQL Server?

It is also possible in SQL Server to create the Primary Key constraint on more than one columns and when we do so, it is called a Composite Primary Key. The maximum number of columns is including in the composite primary key is 16 columns. It is only possible to impose the Composite Primary Key at the table level, it is not possible at the column level.

Note: In a composite primary key, each column can accept duplicate values but the duplicate combination should not be duplicated.

Let us understand this with an example.

Example:

```
--Composite Primary Key

CREATE TABLE BranchDetails
(
    City  VARCHAR(40),
    Bcode INT,
    Bloc  VARCHAR(30),
    PRIMARY KEY(City, Bcode)
)
```

```
--Insert statement to insert a record into the BranchDetails table.

Insert into BranchDetails (City, Bcode, Bloc) values('Mumbai', 10, 'Goregaon')

Insert into BranchDetails (City, Bcode, Bloc) values('Mumbai', 10, 'Malad')

--Execute the following two statements.

Insert into BranchDetails (City, Bcode, Bloc) values('Mumbai', 20, 'Malad')
Insert into BranchDetails (City, Bcode, Bloc) values('Hyderabad', 20, 'SR Nagar')
```

Foreign key

What is a Foreign Key Constraint in SQL Server?

One of the most important concepts in a database is, creating the relationship between the database tables. This relationship provides a mechanism for linking the data stores in multiple tables and retrieving them in an efficient manner.

In order to create a link between two tables, we must specify a Foreign Key in one table that references a column in another table. That means Foreign Key constraint is used for binding two tables with each other and then verify the existence of one table data in other tables.

A foreign key in one TABLE points to a primary key or unique key in another table. The foreign key constraints are used to enforce referential integrity.

Creating PRIMARY KEY and FOREIGN KEY relation on two tables.

Create a table with the name as DEPT by using PRIMARY KEY constraint (Parent table)

Creating another table with the name as Employee by using FOREIGN KEY constraint (Child table)

When we impose Foreign Key constraint and establish the relation between tables the following 3 rules come into the picture

Rule1: Cannot insert a value into the foreign key column provided that value is not existing in the reference key column of the parent (master) table.

Example: INSERT into Employee VALUES (105,'EE', 42000, 50) — Not Allowed

Example:

```
--Foreign Key
--First create a table

CREATE TABLE Dept
(
    Dno   INT PRIMARY KEY,
    Dname VARCHAR(30),
    Dloc  CHAR(40)
)
```

```
--Creating another table with the name as Employee by using FOREIGN KEY c

CREATE TABLE Employee
(
    Eid    INT PRIMARY KEY,
    Ename  VARCHAR(30),
    Salary MONEY,
    Dno    INT FOREIGN KEY REFERENCES Dept(Dno)
)
```

Rule2: Cannot update the reference key value of a parent table provided that the value has a corresponding child record in the child table without addressing what to do with the child records.

Example: UPDATE DEPT SET DNO = 100 WHERE DNO = 10 — Not Allowed

Rule3: Cannot delete a record from the parent table provided that records reference key value has child record in the child table without addressing what to do with the child record.

Example: DELETE FROM DEPT WHERE DNO = 20 — Not Allowed

```
--Insert values into the Employee Table to understand the power of Foreig

INSERT into Employee VALUES (101,'AA', 25000, 10) -- Allowed
INSERT into Employee VALUES (102,'BB', 32000, 20) -- Allowed
INSERT into Employee VALUES (103,'CC', 52000, 40) -- Not Allowed
INSERT into Employee VALUES (104,'DD', 62000, 30) -- Allowed
INSERT into Employee VALUES (105,'EE', 42000, 50) -- Not Allowed
```

```sql
--Rule 1
INSERT into Employee VALUES (105,'EE', 42000, 50) -- Not Allowed
--Rule 2
UPDATE DEPT SET DNO = 100 WHERE DNO = 10 -- Not Allowed
--Rule 3
DELETE FROM DEPT WHERE DNO = 20 -- Not Allowed
```

SQL Statements – INTRO

DDL – Data Definition Language

DML- Data Manipulation Language

SQL STATEMENTS

✓ **DDL STATEMENTS**

used to create, modify the object's structure (database, tables, views, triggers)

CREATE	ALTER
DROP	TRUNCATE
RENAME	

✓ **DML STATEMENTS**

used to insert data, modify the existing data, removing and retrieving data from the tables

INSERT	UPDATE
DELETE	SELECT

✓ **TCL STATEMENTS**

used to save data, undo/redo transactions performed on the database.

COMMIT	ROLLBACK	SAVE TRANSACTION

✓ **DCL STATEMENTS**

used to create roles, permissions, referential integrity and as well it is used to control access to database by securing it.

GRANT	REVOKE

CREATE Statement

DDL STATEMENTS

✓ **CREATE**

used to create object's structure (database, tables, views, triggers)

```
CREATE DATABASE database_name;
CREATE DATABASE employeeDB;
```

```
CREATE TABLE table_name
(column_1     datatype(size)     constraints,
 column_2     datatype(size)     constraints
 .      .     .      .      .     .      .
column_n      datatype(size)     constraints);
```

```
CREATE TABLE employee_info (
 empId          INTEGER          PRIMARY KEY,
 empName        VARCHAR(20)      NOT NULL,
 empSalary      DECIMAL(10,2)    NOT NULL,
 job            VARCHAR(20),
 phone          VARCHAR(20),
 deptID         INTEGER          NOT NULL
);
```

INSERT STATEMENT

✓ **INSERT**

used to insert data or information into the table.

```
INSERT INTO table_name VALUES(col1_value, col2_value, .....);

INSERT INTO employee_info
VALUES (01, 'Adam', 20000, 'Developer', 9879879879, 10);
```

```
INSERT INTO table_name (col1, col2)
VALUES (col1_value, col2_value);

INSERT INTO employee_info (empId, empName, empSalary,
deptId) VALUES (02, 'Smith', 35000, 10);
```

SELECT STATEMENT

DML STATEMENTS

✓ **SELECT**

used to retrieve data or information from the table.

```
SELECT */ [column_name] [function_name] FROM table_name
[WHERE condition]
[GROUP BY condition]
[HAVING condition]
[ORDER BY column_name];
```

```
SELECT * FROM employee_info;

SELECT empName, empSalary FROM employee_info;
```

UPDATE STATEMENT

DML STATEMENTS

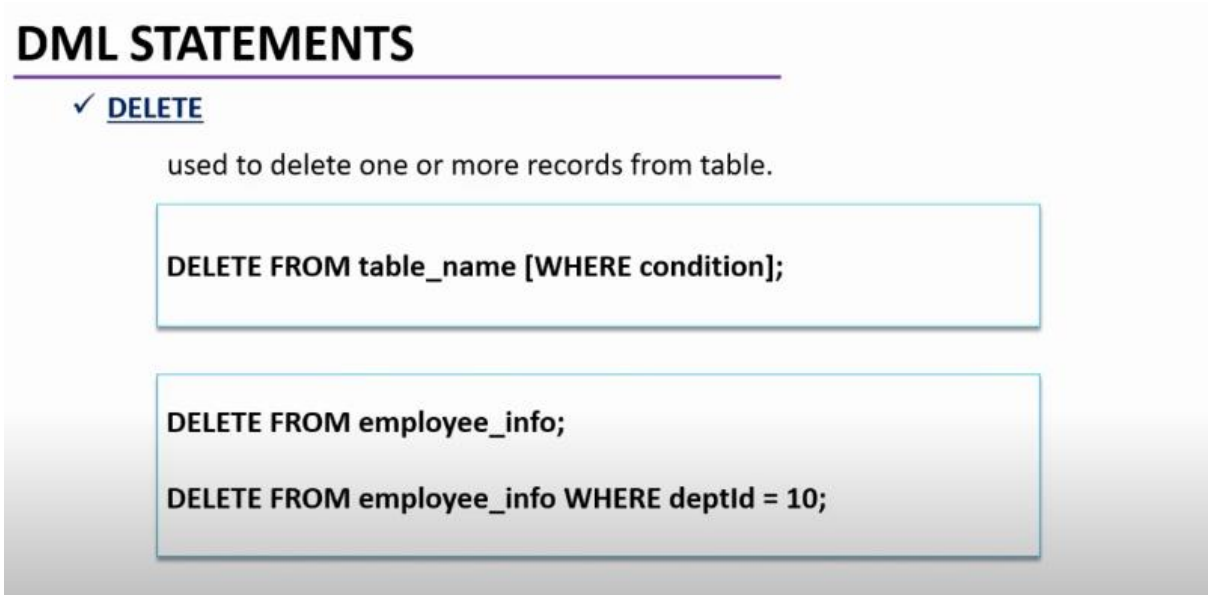
✓ **UPDATE**

used to update/modify existing table data/information .

```
UPDATE table_name
SET column_name = column_value, . . .
[WHERE condition];
```

```
UPDATE employee_info SET empSalary = empSalary + 1000;

UPDATE employee_info SET job = 'Tester'
WHERE empName = 'Smith';
```

DELETE STATEMENT



Rename table and Column:

```
--rename table name
SP_RENAME 'Student', 'StudentDetails'

--truncate table
TRUNCATE TABLE StudentDetails
```

Delete	Truncate
It is a DML command.	It is a DDL command
By using the delete command we can delete a specific record from the table.	But it is not possible with truncate command.
Delete supports WHERE clause.	Truncate does not support the WHERE clause
It is a temporary deletion	It is a permanent deletion
Delete supports rollback transactions for restoring the deleted data.	Truncate doesn't support rollback transaction so that we cannot restore the deleted information
Delete command will not reset identity property.	But it will reset the identity property

```
--truncate table
TRUNCATE TABLE StudentDetails
```

```
--drop table
DROP TABLE StudentDetails
```

Identity Column in SQL Server

What is Identity in SQL Server?

The Identity in SQL Server **is a property that can be applied to a column of a table whose value is automatically created by the server.** So, whenever you marked a column as identity, then that column will be filled in an auto-increment way by SQL Server. That means as **a user we cannot insert a value manually into an identity column**.

Syntax: IDENTITY [(seed,increment)] **Arguments:**

1. Seed: Starting value of a column. The default value is 1.
2. Increment: It specifies the incremental value that is added to the identity column value of the previous row. The default value is 1.

We can set the identity property to a column either when the table is created or after table creation. The following shows an Identity property when the table is created:

Example:

```
--Identity Column

Create Table Person
(
    PersonId int identity(1, 1),
    Name nvarchar(20)
)

select * from person
```

```
--Identity Column after table created


ALTER TABLE Person
DROP COLUMN PersonId;
GO
ALTER TABLE Person
ADD PersonId INT IDENTITY(1,1);

--insert records in the table
Insert into Person values ('Bob')
Insert into Person values ('James')
```

How to explicitly supply Values for Identity Column in SQL Server?

To explicitly supply a value for the identity column

1. First, turn on identity insert – SET Identity_Insert Person ON
2. Secondly, you need to specify the identity column name in the insert query as shown below.

As long as the Identity_Insert is turned on for a table, we **need to explicitly provide the value for that column.** If we don't provide the value, we get an error as shown in the above example.

So once we filled the gaps in the identity column, and if we wish the SQL server to calculate the value, turn off Identity_Insert as shown below.

Example:

```
--To specify the manual value to identity column

SET Identity_Insert Person ON
--Secondly, you need to specify the identity column name in the insert query as shown belo

Insert into Person(Name,PersonId) values( 'Sara',2)
```

```
-- set identity insert off

SET Identity_Insert Person OFF
```

How to Reset the Identity Column Value in SQL Server?

If you have deleted all the rows in a table, and you want to reset the identity column value, then you need to use the DBCC CHECKIDENT command. This command will reset the identity column value.

Syntax: DBCC CHECKIDENT(TableName, RESEED, 0)

Then insert some values and select the data from the table

Example:

```
-- reset identity column value
Delete from Person -- Delete all the records from the Person table

DBCC CHECKIDENT(Person, RESEED, 0) -- Use DBCC command to reset the identity column value
```

Sequence Object in SQL Server

What is a Sequence Object in SQL Server?

A sequence is an object in SQL Server that is used to generate a number sequence. This can be useful when we need to create a unique number to act as a primary key.

The Sequence Object is one of the new features introduced in SQL Server 2012. A sequence is a user-defined object and as its name suggests it generates a sequence of numeric values according to the properties with which it is created. It is similar to the Identity column, but there are many differences between them. But the most important point to keep in mind is that the Sequence Object in SQL Server is not limited to a column or table but is scoped to an entire database.

Properties of Sequence Object:

1. **DataType**: Built-in integer type (tinyint, smallint, int, bigint, decimal, etc…) or user-defined integer type. The default is bigint.
2. **START WITH**: The Start With Value is nothing but the first value that is going to be returned by the sequence object
3. **INCREMENT BY**: The Increment by value is nothing but the value to increment or decrement by the sequence object for each row. If you specify a negative value then the value is going to be decrement.
4. **MINVALUE**: It specifies the value for the sequence object
5. **NO MINVALUE**: It specifies that there is no minimum value specified for the given sequence object.
6. **MAXVALUE**: Maximum value for the sequence object
7. **NO MAXVALUE**: It means that there is no maximum value specified for the sequence.

1. **CYCLE**: It specifies that reset the sequence object when the Sequence Object reached the maximum or minimum value.
2. **NO CYCLE**: When you specify the No Cycle option, then it will throw an error when the Sequence Object reached its maximum or minimum value.
3. **CACHE**: Cache sequence values for performance. The default value is CACHE.
4. **NO CACHE**: As the name says, if you specify the NO CACHE option then it will not cache the sequence numbers.

Note: If you have not specified either Cycle or No Cycle then the default is No Cycle in SQL Server.

Creating an Incrementing Sequence Object in SQL Server:

The following code creates a Sequence object in SQL Server that starts with 1 and increments by 1

How to Generate the Next Sequence Value in SQL Server?

Once we created the sequence object, now let see how to generate the sequence object value. To generate the sequence value in SQL Server, we need to use the NEXT VALUE FOR clause.

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

Every time you execute the above select statement the sequence value will be automatically incremented by 1

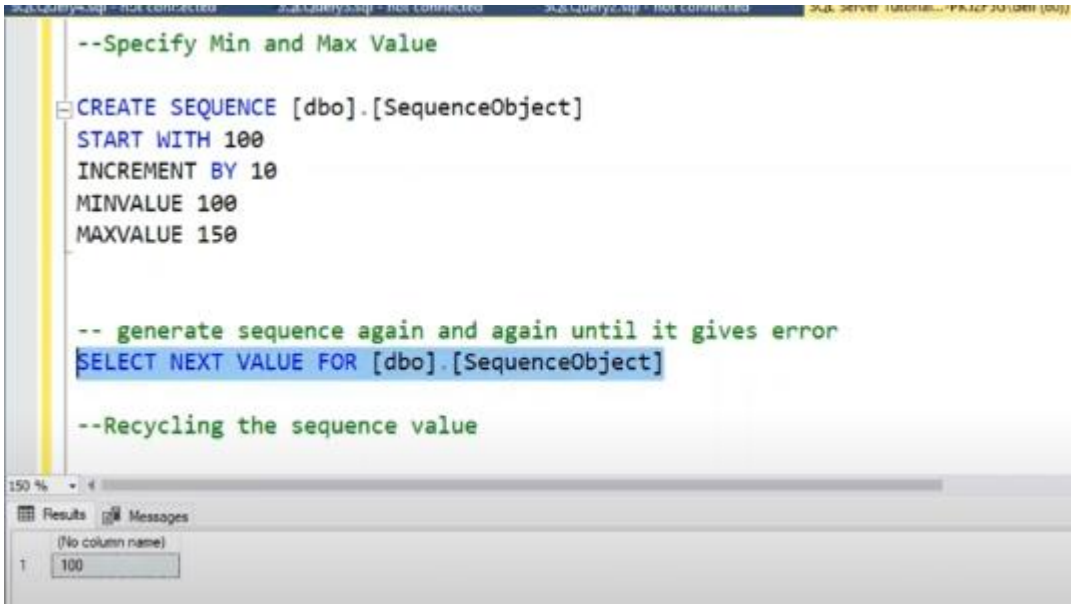
Alter the object to reset the sequence value:

```
ALTER SEQUENCE [SequenceObject] RESTART WITH 1
```

To ensure the value now going to starts from 1, select the next sequence value as shown below.

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

Example



```
--Specify Min and Max Value

CREATE SEQUENCE [dbo].[SequenceObject]
START WITH 100
INCREMENT BY 10
MINVALUE 100
MAXVALUE 150


-- generate sequence again and again until it gives error
SELECT NEXT VALUE FOR [dbo].[SequenceObject]

--Recycling the sequence value
```

150 %

Results | Messages

	(No column name)
1	100

```
--Recycling the sequence value

ALTER SEQUENCE [dbo].[SequenceObject]
INCREMENT BY 10
MINVALUE 100
MAXVALUE 150
CYCLE

--improve the performance of sequence by cache

CREATE SEQUENCE [dbo].[SequenceObject1]
START WITH 1
INCREMENT BY 1
CACHE 10

-- drop sequence

Drop Sequence SequenceObject
```

ORDER BY

ORDER BY CLAUSE

- ✓ used to sort / arrange records in either ascending or descending order.
- ✓ always used with SELECT statement.

```
SELECT column_name(s) FROM table_name
ORDER BY column_name [DESC];
```

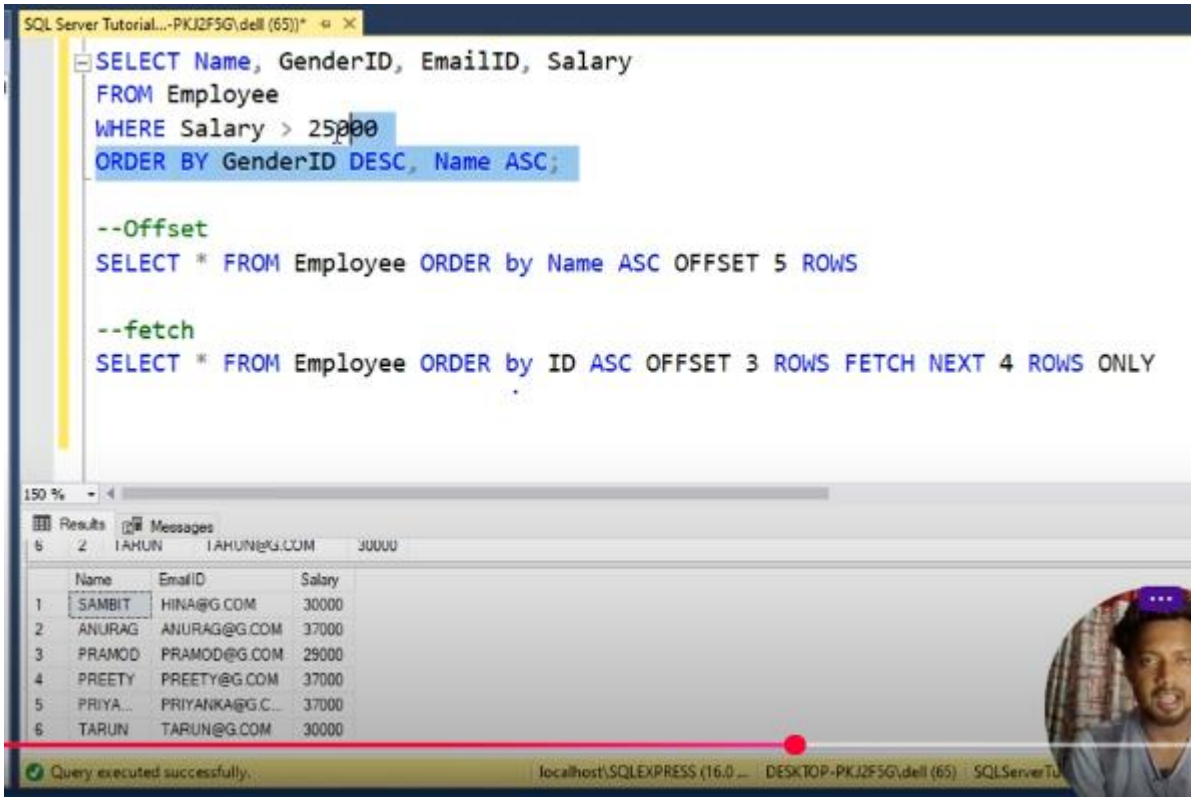
```
SELECT * FROM employee_info ORDER BY empSalary;

SELECT * FROM employee_info
ORDER BY empSalary DESC;
```

WHERE clause

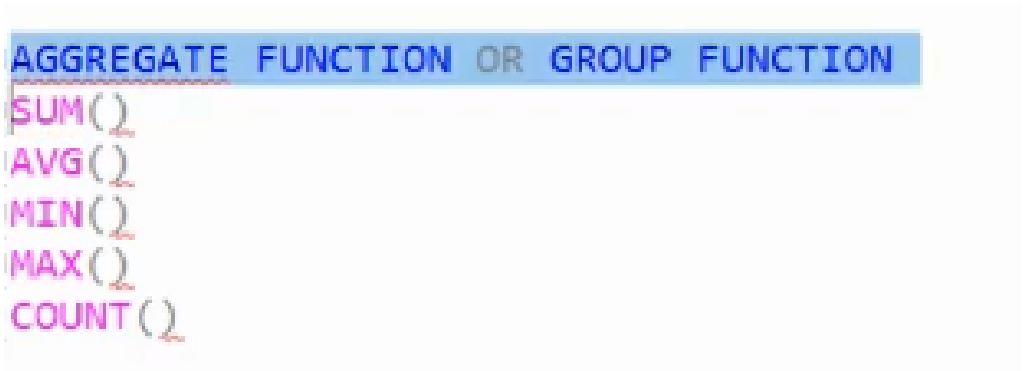
WHERE CLAUSE

- ✓ Using WHERE clause, you are able to restrict the query to rows that meet a condition.
- ✓ You can use any operators in WHERE clause.

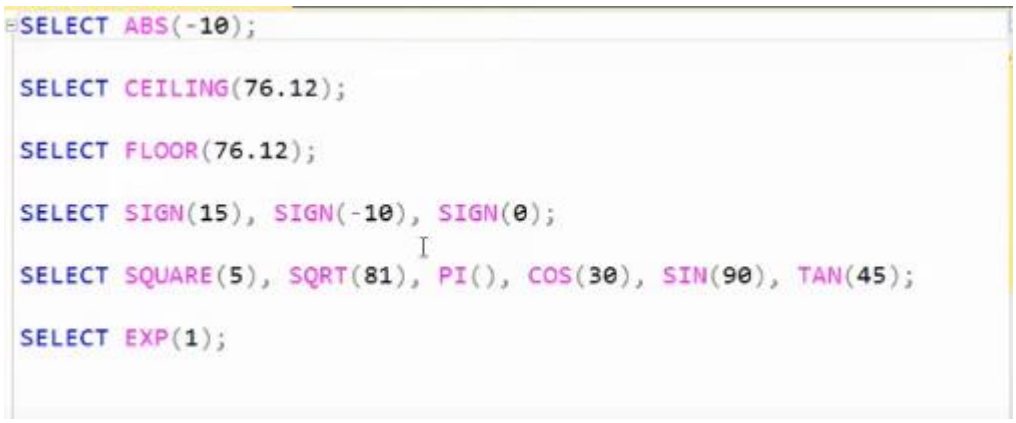
```
SELECT * FROM table_name WHERE condition;

UPDATE table_name SET column_name = value WHERE condition;

DELETE FROM table_name WHERE condition;
```



AGGREGATE FUNCTION



NUMERIC FUNCTION



STRING FUNCTION

```
SELECT empName, LEN(empName) FROM employee_info;

SELECT UPPER('this is a statement');

SELECT empName, LOWER(empName) FROM employee_info;

SELECT LTRIM('    anystring'), RTRIM('anystring     ');

SELECT SUBSTRING('MICROSOFT',6,9);

SELECT REPLACE('MICROSOFT','MICRO','MAJOR');

SELECT REPLICATE('DUMMY ',5)
```

Boolean Operators

```
SELECT * FROM tbl_user_login
    WHERE email = 'steve@captain.com'
    AND password = 'steve456';

SELECT * FROM tbl_user_login
    WHERE username = 'spid'
    OR email = 'peter@parker.com';

SELECT * FROM tbl_user_login
    WHERE NOT username = 'wonder';

SELECT * FROM tbl_user_login
    WHERE username != 'wonder';
```

DATE & TIME FUNCTION:

```
SELECT GETDATE() AS TODAY_DATE;
SELECT SYSDATETIME() AS TODAY_DATE;
SELECT CURRENT_TIMESTAMP AS TODAY_DATE;

SELECT DATENAME(MONTH, CURRENT_TIMESTAMP) AS 'MONTH';
SELECT DATENAME(YEAR, CURRENT_TIMESTAMP) AS 'YEAR';
SELECT DATENAME(HOUR, CURRENT_TIMESTAMP) AS 'HOUR';

SELECT DATEDIFF(YEAR, 'JANUARY 6 1995', CURRENT_TIMESTAMP) AS "AgeInYears";
SELECT DATEDIFF(MONTH, 'JANUARY 6 1995', CURRENT_TIMESTAMP) AS "AgeInMonths";
SELECT DATEDIFF(YEAR, 'JANUARY 10 1997', 'December 31 2019') AS "AgeInYears";

SELECT DATEADD(MONTH, 10, CURRENT_TIMESTAMP) AS "10MonthsFromNow";
```

GROUP BY Clause

- defines one or more columns as a group such that all rows within any group have the same values for those columns.
- always used with SELECT statement.

```
SELECT column_name(s), aggregate_function( )
FROM table_name GROUP BY column_name;
```

```
SELECT deptId FROM employee_info GROUP BY deptID;

SELECT deptId, sum(empSalary) FROM employee_info
GROUP BY deptId;
```

HAVING Clause

- The HAVING clause defines the condition that is then applied to groups of rows.
- always used with SELECT statement inside GROUP BY clause.

```
SELECT column_name(s), aggregate_function( )
FROM table_name
GROUP BY column_name HAVING condition;
```

```
SELECT deptId, sum(empSalary)
FROM employee_info
GROUP BY deptId HAVING deptId = 20;
```

TOP Clause

- The TOP clause specifies the first n rows of the query result that are to be retrieved.
- This clause should always be used with the ORDER BY clause

```
SELECT TOP(n) column_name FROM table_name
ORDER BY column_name [DESC];
```

```
SELECT TOP(3) empSalary FROM employeeInfo
ORDER BY empSalary DESC;
```

```sql
-- use of percent keyword

SELECT TOP 10 Percent
ID, Name, EmailID, CITY
FROM Employee
WHERE Gender = 'MALE'
ORDER BY ID;
```

Over and Partition By Clause

OVER Clause in SQL Server:

The OVER clause in SQL Server is used with PARTITION BY to break up the data into partitions. Following is the syntax of the OVER clause.

```
<function> OVER (
                [PARTITION BY clause]
                [ORDER BY clause]
                [ROWS or RANGE clause])
```

Example:

We need to generate a report to display the total number of employees department-wise. Along with this we also need to display the Total Salary, Average Salary, Minimum Salary, and Maximum Salary department wise. That means we need to generate a report like below.

Department	NoOfEmployees	TotalSalary	AvgSalary	MinSalary	MaxSalary
HR	4	152000	38000	15000	67000
IT	4	188000	47000	15000	96000
Payroll	4	172000	43000	15000	67000

We can easily achieve the above data simply by using the GROUP BY clause in SQL Server.

Example

```
--group by query
SELECT  Department,
 COUNT(*) AS NoOfEmployees,
 SUM(Salary) AS TotalSalary,
 AVG(Salary) AS AvgSalary,
 MIN(Salary) AS MinSalary,
 MAX(Salary) AS MaxSalary
FROM Employees
GROUP BY Department
```

Example:

Now the business requirement changes, now we also need to show the non-aggregated values (Name and Salary) in the report along with the aggregated values as shown in the below image.

Name	Salary	Department	DepartmentTotals	TotalSalary	AvgSalary	MinSalary	MaxSalary
Rasol	15000	HR	4	152000	38000	15000	67000
Stokes	15000	HR	4	152000	38000	15000	67000
Taylor	67000	HR	4	152000	38000	15000	67000
Marshal	55000	HR	4	152000	38000	15000	67000
David	96000	IT	4	188000	47000	15000	96000
Pam	42000	IT	4	188000	47000	15000	96000
James	15000	IT	4	188000	47000	15000	96000
Smith	35000	IT	4	188000	47000	15000	96000
Rakesh	35000	Payroll	4	172000	43000	15000	67000
Preety	67000	Payroll	4	172000	43000	15000	67000
Priyanka	55000	Payroll	4	172000	43000	15000	67000
Anurag	15000	Payroll	4	172000	43000	15000	67000

```
--group by query with adding some more fields

SELECT  Name, Salary, Department,
 COUNT(*) AS NoOfEmployees,
 SUM(Salary) AS TotalSalary,
 AVG(Salary) AS AvgSalary,
 MIN(Salary) AS MinSalary,
 MAX(Salary) AS MaxSalary
FROM Employees
GROUP BY Department
```

```
Messages
Msg 8120, Level 16, State 1, Line 174
Column 'Employees.Name' is invalid in the select list because it is not contained in ei

Completion time: 2023-05-28T20:55:46.4273758+05:30
```

How can we achieve the desired output?

We can get the desired output in two ways.

Solution1:

One of the ways to get the desired output is by including all the aggregations in a subquery and then JOINING that subquery with the main query.

Once you execute the above query then you will get the desired output. But look at the number of T-SQL statements that we wrote.

```
--Desired output using subquery

SELECT Name, Salary, Employees.Department,
  Departments.DepartmentTotals,
  Departments.TotalSalary,
  Departments.AvgSalary,
  Departments.MinSalary,
  Departments.MaxSalary
FROM  Employees
INNER JOIN
( SELECT Department, COUNT(*) AS DepartmentTotals,
  SUM(Salary) AS TotalSalary,
  AVG(Salary) AS AvgSalary,
   MIN(Salary) AS MinSalary,
   MAX(Salary) AS MaxSalary
  FROM Employees
  GROUP BY Department) AS Departments
ON Departments.Department = Employees.Department
```

Solution2:

The second way that is the most preferable way to get the desired output is by using the OVER clause combined with the PARTITION BY clause

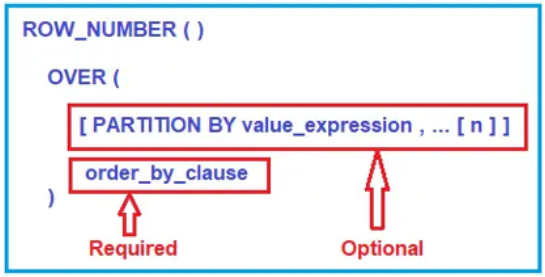
```sql
--using over clause

SELECT  Name,
 Salary,
 Department,
 COUNT(Department) OVER(PARTITION BY Department) AS DepartmentTotals,
 SUM(Salary) OVER(PARTITION BY Department) AS TotalSalary,
 AVG(Salary) OVER(PARTITION BY Department) AS AvgSalary,
 MIN(Salary) OVER(PARTITION BY Department) AS MinSalary,
 MAX(Salary) OVER(PARTITION BY Department) AS MaxSalary
FROM Employees
```

Row Number Function

ROW_NUMBER Function in SQL Server:

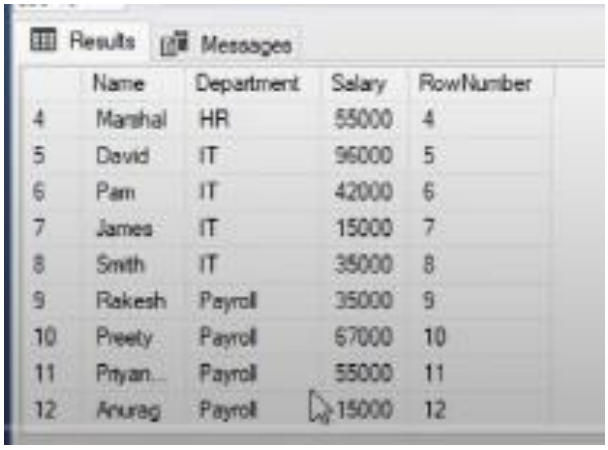
The Row Number function was introduced in SQL Server 2005. The ROW_NUMBER function is basically used when you want to return a sequential number starting from 1.

The ROW_NUMBER() is a built-in function in SQL Server that assigns a sequential integer number to each row within a partition of a result set. The row number always starts with 1 for the first row in each partition and then increases by 1 for the next row onwards in each partition. The syntax to use the ROW_NUMBER function is given below.



ROW_NUMBER Function without PARTITION BY:

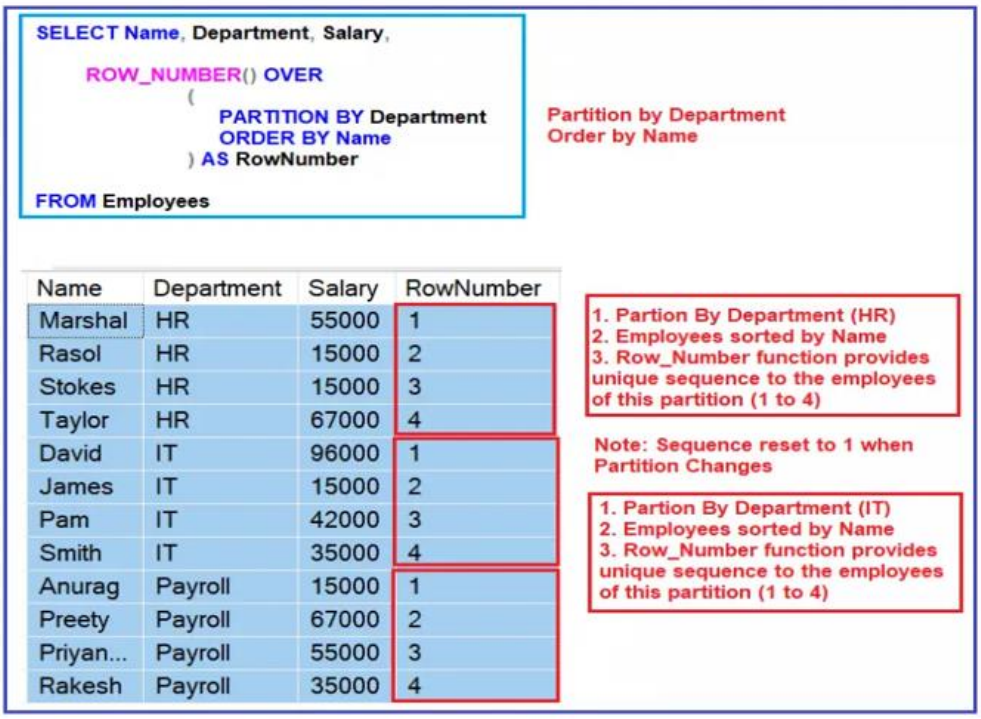
```sql
--Row_Number Class

--Row_Number Function without partition by

SELECT Name, Department, Salary,
ROW_NUMBER() OVER (ORDER BY Department) AS RowNumber
FROM Employees
```



Results | Messages

	Name	Department	Salary	RowNumber
4	Marshal	HR	55000	4
5	David	IT	96000	5
6	Pam	IT	42000	6
7	James	IT	15000	7
8	Smith	IT	35000	8
9	Rakesh	Payroll	35000	9
10	Preety	Payroll	67000	10
11	Priyan...	Payroll	55000	11
12	Anurag	Payroll	15000	12

Row_Number Function with PARTITION BY Clause:



```
SELECT Name, Department, Salary,
    ROW_NUMBER() OVER
        (
            PARTITION BY Department
            ORDER BY Name
        ) AS RowNumber
FROM Employees
```

Partition by Department
Order by Name

Name	Department	Salary	RowNumber
Marshal	HR	55000	1
Rasol	HR	15000	2
Stokes	HR	15000	3
Taylor	HR	67000	4
David	IT	96000	1
James	IT	15000	2
Pam	IT	42000	3
Smith	IT	35000	4
Anurag	Payroll	15000	1
Preety	Payroll	67000	2
Priyan...	Payroll	55000	3
Rakesh	Payroll	35000	4

1. Partion By Department (HR)
2. Employees sorted by Name
3. Row_Number function provides unique sequence to the employees of this partition (1 to 4)

Note: Sequence reset to 1 when Partition Changes

1. Partion By Department (IT)
2. Employees sorted by Name
3. Row_Number function provides unique sequence to the employees of this partition (1 to 4)

Example:

ID	Name	Department	Salary
1	James	IT	15000
1	James	IT	15000
2	Rasol	HR	15000
2	Rasol	HR	15000
2	Rasol	HR	15000
3	Stokes	HR	15000
3	Stokes	HR	15000
3	Stokes	HR	15000
3	Stokes	HR	15000

ID	Name	Department	Salary
1	James	IT	15000
2	Rasol	HR	15000
3	Stokes	HR	15000

```
--Delete duplicate records using Row_Number function

WITH DeleteDuplicateCTE AS
(
    SELECT *, ROW_NUMBER() OVER(PARTITION BY ID ORDER BY ID) AS RowNumber
    FROM Employees
)
DELETE FROM DeleteDuplicateCTE WHERE RowNumber > 1
```

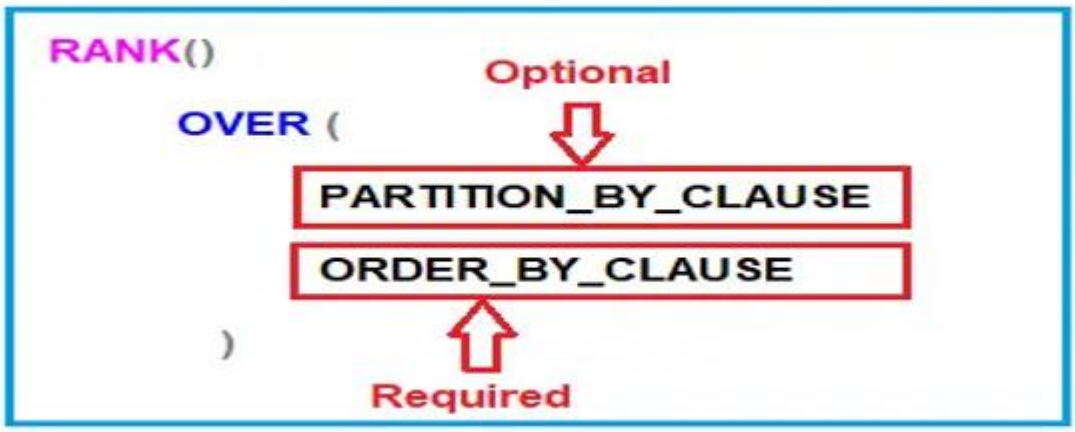
150 %

Results | Messages

	ID	Name	Department	Salary
1	1	James	IT	15000
2	2	Rasol	HR	15000
3	3	Stokes	HR	15000

RANK and DENSE_RANK Function in SQL Server

RANK:



We are going to use the following Employees table to understand the RANK and DENSE_RANK function.

Id	Name	Department	Salary
1	James	IT	80000
2	Taylor	IT	80000
3	Pamela	HR	50000
4	Sara	HR	40000
5	David	IT	35000
6	Smith	HR	65000
7	Ben	HR	65000
8	Stokes	IT	45000
9	Taylor	IT	70000
10	John	IT	68000

RANK Function without PARTITION

```
--Rank function without partition by

SELECT Name, Department, Salary,
RANK() OVER (ORDER BY Salary DESC) AS [Rank]
FROM Employees
```

table SQLServerTutorial.dbo.Employees

```
--Rank Function with Partition by

SELECT Name, Department, Salary,
```

150 %

Results | Messages

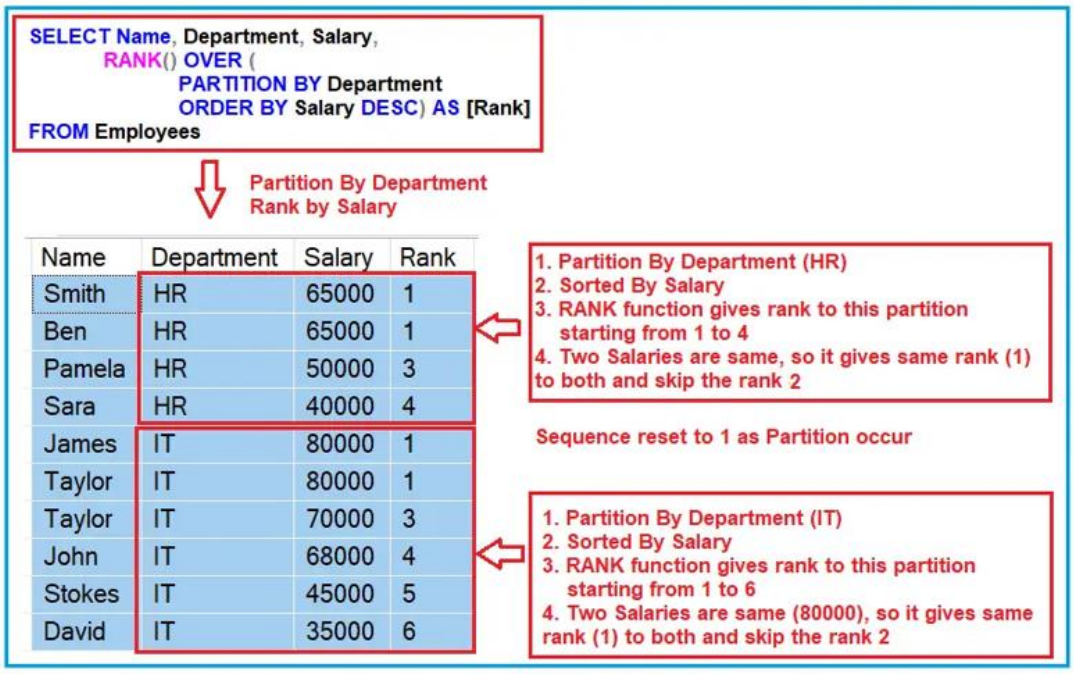
	Name	Department	Salary	Rank
1	James	IT	80000	1
2	Taylor	IT	80000	1
3	Taylor	IT	70000	3
4	John	IT	68000	4
5	Smith	HR	65000	5
6	Ben	HR	65000	5
7	Pamela	HR	50000	7
8	Stokes	IT	45000	8
9	Sara	HR	40000	9
10	David	IT	35000	10

Applying Order BY Clause in Salary Column

Name	Department	Salary	Rank	
James	IT	80000	1	Salary Same so Rank Same
Taylor	IT	80000	1	
Taylor	IT	70000	3	
John	IT	68000	4	
Smith	HR	65000	5	Salary Same so Rank Same
Ben	HR	65000	5	
Pamela	HR	50000	7	
Stokes	IT	45000	8	
Sara	HR	40000	9	
David	IT	35000	10	

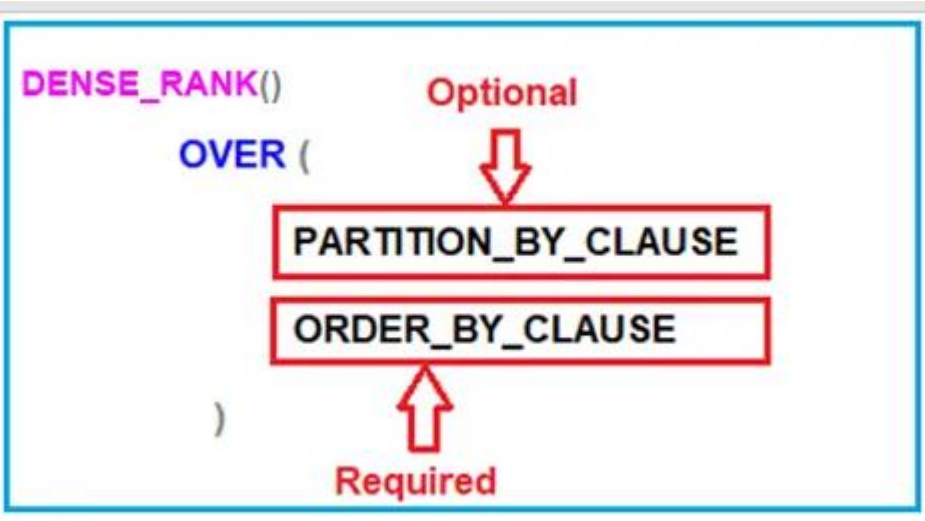
The Rank function in SQL Server skips the ranking(s) when there is a tie. As you can see in the above output, Ranks 2 and 6 are skipped as there are 2 rows at rank 1 as well as 2 rows at rank 5. The third row gets rank 3 and the 7th row gets rank 7.

RANK Function with PARTITION BY clause in SQL Server:



So, in short, The RANK function Returns an increasing unique number for each row starting from 1 and for each partition. When there are duplicates, the same rank is assigned to all the duplicate rows, but the next row after the duplicate rows will have the rank it would have been assigned if there had been no duplicates. So the RANK function skips rankings if there are duplicates.

DENSE_RANK Function in SQL Server:



DENSE_RANK Function without PARTITION BY clause in SQL Server:

```sql
--Dense_Rank without partition by

SELECT Name, Department, Salary,
           DENSE_RANK() OVER (ORDER BY Salary DESC) AS [Rank]
FROM Employees

--Dense_Rank with partition by
```

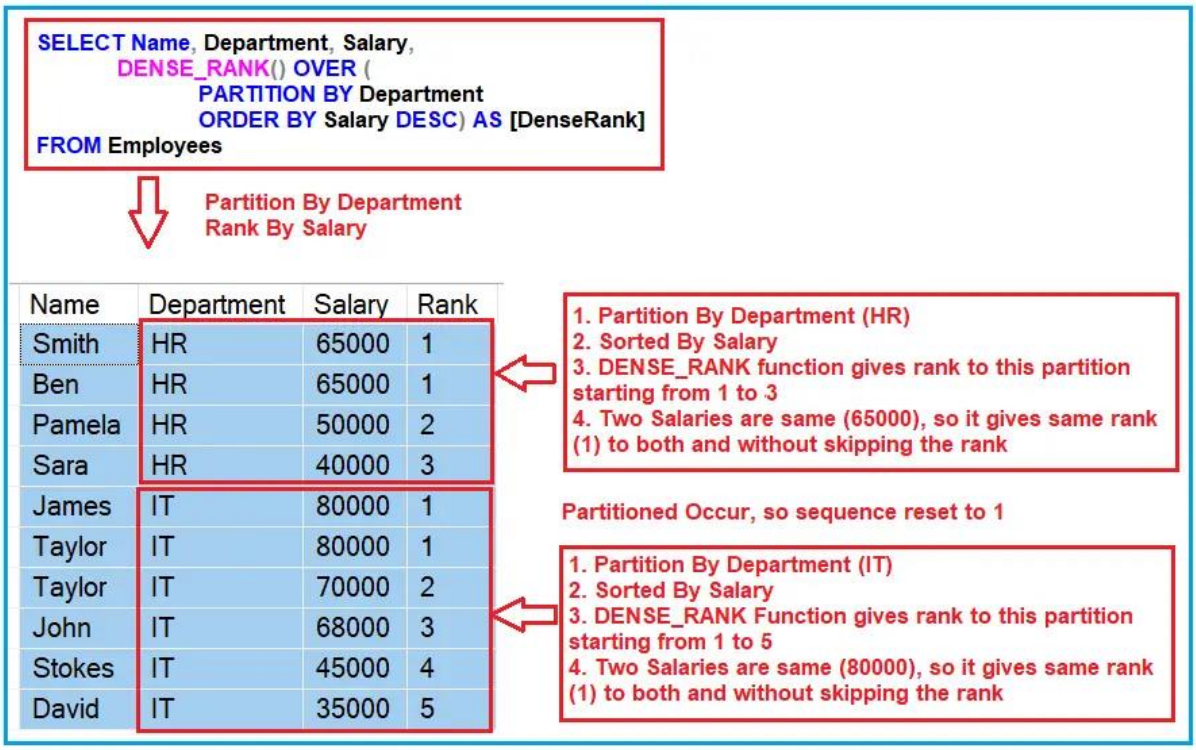
150 %

Results | Messages

	Name	Department	Salary	Rank
1	Smith	HR	65000	1
2	Ben	HR	65000	1
3	Pamela	HR	50000	3
4	Sara	HR	40000	4
5	James	IT	80000	1
6	Taylor	IT	80000	1
7	Taylor	IT	70000	3
8	John	IT	68000	4
9	Stokes	IT	45000	5
10	David	IT	35000	6

Name	Department	Salary	Rank	
James	IT	80000	1	Salary same so same rank
Taylor	IT	80000	1	
Taylor	IT	70000	2	
John	IT	68000	3	
Smith	HR	65000	4	Salary same so same rank
Ben	HR	65000	4	
Pamela	HR	50000	5	
Stokes	IT	45000	6	
Sara	HR	40000	7	
David	IT	35000	8	

DENSE_RANK Function with PARTITION BY clause in SQL Server:



The Real-time examples of RANK and DENSE_RANK Functions in SQL Server:

Find the nth highest salary. Both the RANK and DENSE_RANK functions can be used to find nth highest salary.

Fetch the 2nd Highest Salary using the RANK function:

```
-- Fetch the 2nd Hight Salary
WITH EmployeeCTE  AS
(
    SELECT Salary, DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank_Salry
    FROM Employees
)
SELECT TOP 1 Salary FROM EmployeeCTE WHERE DenseRank_Salry = 2
```

Since, in our Employees table, we have 2 employees with the FIRST highest salary (80000), the Rank() function will not return any data for the SECOND highest Salary. Please execute the below SQL Script and see the output.

Between and Like Operators

BETWEEN Operator in SQL Server:

The BETWEEN operator in SQL Server is used to get the values within a range. Generally, we use the BETWEEN operator in the WHERE clause to get values within a specified range. For example, if you want to fetch all the employees between ID 3 and 7 from the Employee table, then you need to use the BETWEEN operator

Example.

Points to Remember while working with Between Operator in SQL Server:

1. Between Operator returns true if the operand is within a range.
2. Between Operator will return records including the starting and ending values.
3. This operator support only the AND operator.
4. The BETWEEN Operator takes the values from small to big range in the query.

NOT BETWEEN Operator in SQL Server:

If you use the NOT keyword along with the BETWEEN operator then it will return data where the column values not in between the range values. For example, the following SQL Query will return employees whose employee id not between 3 and 7. The point that you need to remember is it also not return the employee whose id is 3 and 7.

Example:

```
--Between Operator

SELECT * FROM Employee WHERE ID BETWEEN 3 AND 7

--Not Between Operator

SELECT * FROM Employee WHERE ID NOT BETWEEN 3 AND 7
```

IN Operator in SQL Server:

The IN Operator in SQL Server is used to search for specified values that match any value in the set of multiple values it accepts. Generally, we will use this IN operator in WHERE clause to compare column or variable values with a set of multiple values. For example, if you want to fetch all the employees whose department is either IT or HR, then you could write the SQL Query using the IN Operator.

NOT IN Operator in SQL Server:

This is just opposite to the IN Operator. The IN operator takes a set of values and then returns the records whose column values matched with the values it has. But if you use the NOT keyword along with the IN operator, then it will return data where column value not in the set of values. For example, the following SQL query will return all the employees where the Department not in IT and HR.

```
--IN Operator

SELECT * FROM Employee WHERE Department IN ('IT', 'HR')

--NOT IN Operator

SELECT * FROM Employee WHERE Department NOT IN ('IT', 'HR')
```

LIKE Operator in SQL Server:

This is one of the most frequently used operators in SQL Server. The LIKE operator in SQL Server is used to search for character string with the specified pattern using wildcards in the column. Generally, we will use this LIKE operator in the WHERE clause. Here, we will try to understand the LIKE operator using different types of wildcard characters.

Using '%' wildcard

```
--Like Operator
--Using % wildcard

SELECT * FROM Employee WHERE Name LIKE 'P%'

-- Example 2

SELECT * FROM Employee WHERE Name LIKE '%a'
```

```sql
--Example 3

SELECT * FROM Employee WHERE Name LIKE '%am%'
```

Understanding the WildCard Characters:

You can use the following wildcard characters with the LIKE operator in SQL Server.

1. % symbol represents any no of characters in the expression.
2. _ will represent a single character in the expression.
3. The [] symbol indicates a set of characters in the expression.
4. [^] will represent any single character, not within the specified range

```sql
--Understanding the WildCard Characters

--WAQ to display employee details whose name contains 3 characters.
SELECT * FROM Employee WHERE Name LIKE '___'

--WAQ to display employee details whose name contains 'A' character.
SELECT  * FROM Employee WHERE Name LIKE '%A%'
```

```sql
--WAQ to display employee details whose name starts with 'P' character and ends with 'A'
SELECT * FROM Employee WHERE Name LIKE 'P%A'

--WAQ to display employee details whose name starts with J, H, K, U characters.
SELECT * FROM Employee WHERE Name LIKE '[J, H, K, U]%'

--WAQ to display employee details whose names start with A to Z characters.
SELECT * FROM Employee WHERE Name LIKE '[A-Z]%'
```

All and Any Operator in SQL Server

```sql
SELECT * FROM PermanentEmployee WHERE Age > ALL (SELECT AGE FROM ContractEmployee)

-- Create Temp Table and insert some test data

CREATE TABLE #TEMP_TABLE (ID INT)

INSERT INTO #TEMP_TABLE VALUES(1)
INSERT INTO #TEMP_TABLE VALUES(2)
INSERT INTO #TEMP_TABLE VALUES(3)
```

```sql
IF 3 > ALL (SELECT ID FROM #TEMP_TABLE)
      PRINT 'Returned True'
ELSE
      PRINT 'Returned False'
```

O/P: Returned True

Exists Operator

```
-- Exist with select statement

SELECT *
FROM EmployeeDetails
WHERE EXISTS (SELECT *
              FROM EmployeeContactDetails
              WHERE EmployeeDetails.ID = EmployeeContactDetails.EmployeeID)

--No Exist with Select statement

SELECT *
FROM EmployeeDetails
WHERE NOT EXISTS (SELECT *
              FROM EmployeeContactDetails
              WHERE EmployeeDetails.ID = EmployeeContactDetails.EmployeeID);
```

Intersect Operator in SQL Server

INTERSECT operator in SQL Server?

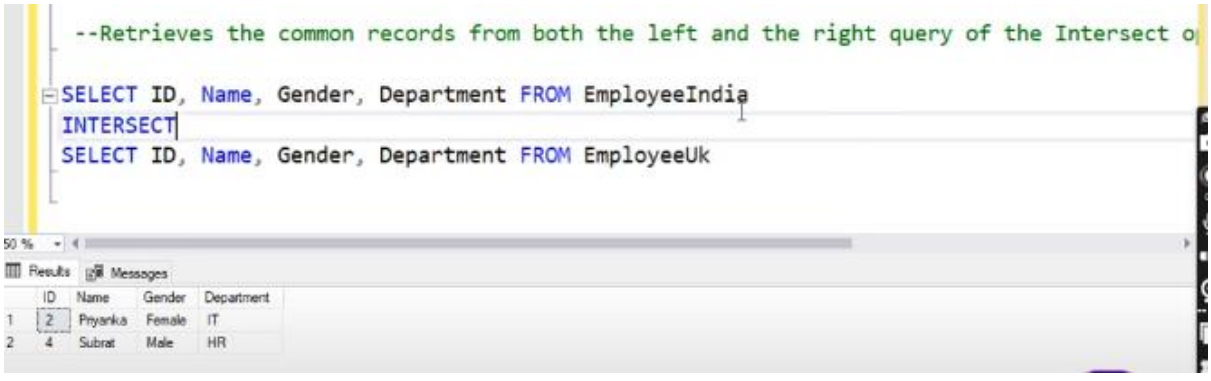
The INTERSECT operator in SQL Server is used to retrieve the common records of both the left and the right query of the Intersect operator.

Understanding the INTERSECT operator with examples.

We are going to use the following "**EmployeeIndia**" and "**EmployeeUK**" tables to understand INTERSECT SET operator.

Difference between INTERSECT and INNER JOIN in SQL Server?

The INTERSECT Operator filters duplicate rows and return only the DISTINCT rows that are common between the LEFT and Right Query, whereas INNER JOIN does not filter the duplicates.



Except Operator in SQL Server: **Return data employeeiIndia data which are not present in employeeUK**

```
-- Class - 22
--Except Example

SELECT ID, Name, Gender, Department FROM EmployeeIndia
EXCEPT
SELECT ID, Name, Gender, Department FROM EmployeeUK
```

Union and Union All Operator

UNION and UNION ALL operators in SQL Server

The UNION and UNION ALL operators in SQL Server are used to combine the result-set of two or more SELECT statements into a single result set. We are going to use the following "**EmployeeIndia**" and "**EmployeeUK**" tables to understand these two operators.

Points to Remember while working with Set Operations:

1. The result sets of all queries **must have the same number of columns.**
2. In every result set the **data type of each column must be compatible (well-matched) to the data type of its corresponding column in other result sets.**

Differences between UNION and UNION ALL Operator in SQL Server

From the output, it is very clear that UNION removes duplicate rows whereas UNION ALL does not remove the duplicate rows. When we use a UNION operator to remove the duplicate rows from the result set, **the SQL server has to do a distinct operation which is time-consuming.** For this reason, **UNION ALL is much faster than UNION**.

```sql
--Union Example
SELECT ID, Name, Gender, Department FROM EmployeeIndia
UNION
Select ID, Name, Gender, Department FROM EmployeeUK
```

150 %

Results | Messages

	ID	Name	Gender	Department
1	1	James	Male	IT
2	1	Pranaya	Male	IT
3	2	Priyanka	Female	IT
4	3	Preety	Female	HR
5	3	Sara	Female	HR
6	4	Subrat	Male	HR
7	5	Anurag	Male	IT
8	5	Pam	Female	HR

```sql
--Union All Example

SELECT ID, Name, Gender, Department FROM EmployeeIndia
UNION ALL
SELECT ID, Name, Gender, Department FROM EmployeeUK

--sorting should be on last select statement column
```

150 %

Results | Messages

	ID	Name	Gender	Department
1	1	Pranaya	Male	IT
2	2	Priyanka	Female	IT
3	3	Preety	Female	HR
4	4	Subrat	Male	HR
5	5	Anurag	Male	IT
6	1	James	Male	IT
7	2	Priyanka	Female	IT
8	3	Sara	Female	HR
9	4	Subrat	Male	HR
10	5	Pam	Female	HR

Coping data from one database to another in SQL Server

✓ **To create a copy of the table from different database**

```
SELECT column_name / * INTO table_name
FROM database_name.table_name;
```

```
SELECT * INTO employee_details
FROM employee_db.employee_info
```

ALTER STATEMENT (Part I - Add new columns in an existing table)

- Modifies a table definition by adding, altering, or dropping columns and constraints.
- It also reassigns and rebuilds partitions, or disables and enables constraints and triggers.

✓ **Adding a new column**

– adds a column without constraint that allows null values.

```
ALTER TABLE table_name ADD column_name datatype(size) [null];
```

```
ALTER TABLE emp_info ADD salary decimal;

ALTER TABLE emp_info ADD phone varchar(10) null;
```

✓ **Adding a column with a constraint**

– adds a new column with constraint (UNIQUE, DEFAULT etc.).

```
ALTER TABLE table_name ADD column_name datatype(size) constraint [null];

ALTER TABLE table_name ADD column_name datatype(size) null
        CONSTRAINT constraint_reference_name constraint;
```

```
ALTER TABLE emp_info ADD salary decimal not null;

ALTER TABLE emp_info ADD projectID integer null
        CONSTRAINT pID_unique_key UNIQUE;
```

✓ **Adding several columns with constraints**

- adds more than one column with constraints defined with the new column.

```
ALTER TABLE table_name
        ADD column_name datatype(size) constraint,
                        ...     ...     ...             ;
```

```
ALTER TABLE emp_info
        ADD salary decimal DEFAULT 15000,
            projectID integer null CONSTRAINT pID_unique_key UNIQUE;
```

ALTER STATEMENT (Part II - Drop column and constraint from table)

✓ **Dropping a column or columns**

- remove a column or multiple columns.

```
ALTER TABLE table_name DROP COLUMN column_name(s);
```

```
ALTER TABLE emp_info DROP COLUMN salary;

ALTER TABLE emp_info DROP COLUMN salary, age;
```

✓ **Dropping constraints and columns**

- removes a constraint
- removes constraints and columns.

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;

ALTER TABLE table_name DROP CONSTRAINT constraint_name,
                                COLUMN column_name;
```

```
ALTER TABLE emp_info DROP CONSTRAINT pID_unique_key;

ALTER TABLE emp_info DROP CONSTRAINT pID_unique_key,
                                COLUMN salary;
```

ALTER STATEMENT (Part III - Altering an existing table)

✓ **Changing the data type of a column**

- changes a column of a table from one data type to another.

```
ALTER TABLE table_name ALTER COLUMN column_name datatype(size);
```

```
ALTER TABLE emp_info ALTER COLUMN salary decimal(8,2);

ALTER TABLE emp_info ALTER COLUMN emp_name varchar(50);
```

✓ **Changing the size of a column**

- change (increase or decrease) the size of a column.

```
ALTER TABLE table_name ALTER COLUMN column_name datatype(size);
```

```
ALTER TABLE emp_info ALTER COLUMN salary decimal(8,2);

ALTER TABLE emp_info ALTER COLUMN emp_name varchar(50);
```

Note: If the columns contain data, the column size can only be increased.

Aliases in SQL Server

- can be used to create a temporary name for columns or tables.

Types

1. **Column Aliases**	are used to make column headings in query output easier to read. (Specially with Functions and Column Concatenation)
2. **Table Aliases**	are used to shorten your SQL to make it easier to read. (Specially in Join, and Subquery)

Syntax

```
SELECT column_name [function( )] AS alias_name FROM table_name;
SELECT column_name [function( )] alias_name FROM table_name;
```

Example

```
SELECT sal AS Salary FROM employee;
SELECT fname || lname 'Full Name' FROM employee;
```

Joins in SQL Server

- used to retrieve data from multiple tables.

Types

1. **Inner Join**
2. **Outer Join**
    i. Left Outer Join
    ii. Right Outer Join
    iii. Full Outer Join
3. **Cross Join**

Inner Join

Inner Join

- The inner join is one of the most commonly used joins in SQL Server.
- It return all rows from multiple tables where the join condition is satisfied.

Syntax

```
SELECT column_name(s) FROM table1_name INNER JOIN table2_name
ON table1_name.column_name = table2_name.column_name;
```

EMPLOYEE

EMP_ID	EMP_NAME	EMP_SALARY	EMP_DEPTID
1111	STEVE	35000	D1
1112	ADAM	28000	D2
1113	JOHN	50000	D3
1114	MIKE	45000	D2
1115	PETER	60000	D5

DEPARTMENT

DEPT_ID	DEPT_NAME	DEPT_LOCATION
D1	DEVELOPMENT	CALIFORNIA
D2	MARKETING	MUMBAI
D3	ACCOUNTS	NEW YORK
D4	MANAGEMENT	SYDNEY

EMPLOYEE_RECORD

EMP_ID	EMP_NAME	EMP_SALARY	DEPT_NAME	DEPT_LOCATION
1111	STEVE	35000	DEVELOPMENT	CALIFORNIA
1112	ADAM	28000	MARKETING	MUMBAI
1113	JOHN	50000	ACCOUNTS	NEW YORK
1114	MIKE	45000	MARKETING	MUMBAI

Example

```
SELECT emp_id, emp_name, emp_salary, dept_name, dept_location
FROM employee INNER JOIN department
ON employee.emp_deptid = department.dept_id;
```

```
SELECT e.emp_id, e.emp_name, e.emp_salary, d.dept_name,
d.dept_location FROM employee e INNER JOIN department d
ON e.emp_deptid = d.dept_id;
```

Left Outer Join

Left Outer Join

- return all rows from the left-hand table and records in the right-hand table with matching values.

Syntax

```
SELECT column_name(s)
FROM table1_name LEFT OUTER JOIN table2_name
ON table1_name.column_name = table2_name.column_name;
```

EMPLOYEE

EMP_ID	EMP_NAME	EMP_SALARY	EMP_DEPTID
1111	STEVE	35000	D1
1112	ADAM	28000	D2
1113	JOHN	50000	D3
1114	MIKE	45000	D2
1115	PETER	60000	D5

DEPARTMENT

DEPT_ID	DEPT_NAME	DEPT_LOCATION
D1	DEVELOPMENT	CALIFORNIA
D2	MARKETING	MUMBAI
D3	ACCOUNTS	NEW YORK
D4	MANAGEMENT	SYDNEY

EMPLOYEE_RECORD

EMP_ID	EMP_NAME	EMP_SALARY	DEPT_NAME	DEPT_LOCATION
1111	STEVE	35000	DEVELOPMENT	CALIFORNIA
1112	ADAM	28000	MARKETING	MUMBAI
1113	JOHN	50000	ACCOUNTS	NEW YORK
1114	MIKE	45000	MARKETING	MUMBAI
1115	PETER	60000	NULL	NULL

Example

```
SELECT emp_id, emp_name, emp_salary, dept_name, dept_location
FROM employee LEFT OUTER JOIN department
ON employee.emp_deptid = department.dept_id;
```

```
SELECT e.emp_id, e.emp_name, e.emp_salary, d.dept_name,
d.dept_location FROM employee e LEFT OUTER JOIN department d
ON e.emp_deptid = d.dept_id;
```

Right Outer Join

Right Outer Join

- return all rows from the right-hand table and records in the left-hand table with matching values.

Syntax

```
SELECT column_name(s)
FROM table1_name RIGHT OUTER JOIN table2_name
ON table1_name.column_name = table2_name.column_name;
```

EMPLOYEE

EMP_ID	EMP_NAME	EMP_SALARY	EMP_DEPTID
1111	STEVE	35000	D1
1112	ADAM	28000	D2
1113	JOHN	50000	D3
1114	MIKE	45000	D2
1115	PETER	60000	D5

DEPARTMENT

DEPT_ID	DEPT_NAME	DEPT_LOCATION
D1	DEVELOPMENT	CALIFORNIA
D2	MARKETING	MUMBAI
D3	ACCOUNTS	NEW YORK
D4	MANAGEMENT	SYDNEY

EMPLOYEE_RECORD

EMP_ID	EMP_NAME	EMP_SALARY	DEPT_NAME	DEPT_LOCATION
1111	STEVE	35000	DEVELOPMENT	CALIFORNIA
1112	ADAM	28000	MARKETING	MUMBAI
1114	MIKE	45000	MARKETING	MUMBAI
1113	JOHN	50000	ACCOUNTS	NEW YORK
NULL	NULL	NULL	MANAGEMENT	SYDNEY

Example

```
SELECT emp_id, emp_name, emp_salary, dept_name, dept_location
FROM employee RIGHT OUTER JOIN department
ON employee.emp_deptid = department.dept_id;
```

```
SELECT e.emp_id, e.emp_name, e.emp_salary, d.dept_name,
d.dept_location FROM employee e RIGHT OUTER JOIN department d
ON e.emp_deptid = d.dept_id;
```

Full Outer Join

Full Outer Join

- return all rows from both left-hand and right-hand table with matching values.

Syntax

```
SELECT column_name(s)
FROM table1_name FULL OUTER JOIN table2_name
ON table1_name.column_name = table2_name.column_name;
```

EMPLOYEE

EMP_ID	EMP_NAME	EMP_SALARY	EMP_DEPTID
1111	STEVE	35000	D1
1112	ADAM	28000	D2
1113	JOHN	50000	D3
1114	MIKE	45000	D2
1115	PETER	60000	D5

DEPARTMENT

DEPT_ID	DEPT_NAME	DEPT_LOCATION
D1	DEVELOPMENT	CALIFORNIA
D2	MARKETING	MUMBAI
D3	ACCOUNTS	NEW YORK
D4	MANAGEMENT	SYDNEY

EMPLOYEE_RECORD

EMP_ID	EMP_NAME	EMP_SALARY	DEPT_NAME	DEPT_LOCATION
1111	STEVE	35000	DEVELOPMENT	CALIFORNIA
1112	ADAM	28000	MARKETING	MUMBAI
1113	JOHN	50000	ACCOUNTS	NEW YORK
1114	MIKE	45000	MARKETING	MUMBAI
1115	PETER	60000	NULL	NULL
NULL	NULL	NULL	MANAGEMENT	SYDNEY

Example

```
SELECT emp_id, emp_name, emp_salary, dept_name, dept_location
FROM employee FULL OUTER JOIN department
ON employee.emp_deptid = department.dept_id;
```

```
SELECT e.emp_id, e.emp_name, e.emp_salary, d.dept_name,
d.dept_location FROM employee e FULL OUTER JOIN department d
ON e.emp_deptid = d.dept_id;
```

Self Join in SQL Server

Self Join in SQL Server?

The Self Join is nothing a concept where **we need to join a table by itself**. You need to use Self Join **when you have some relations between the columns of the same table.**The point that you need to remember that when you are implementing the self-join mechanism then **you need to create the alias for the table**. You can also **create any number of aliases for a single table** in SQL Server.

Let's understand Self Join with Examples

We are going to use the following Employee table to understand the SQL Server Self Join concept.

EmployeeID	Name	ManagerID
1	Pranaya	3
2	Priyanka	1
3	Preety	NULL
4	Anurag	1
5	Sambit	1

```
--Self Left Join

SELECT *
FROM Employee E
LEFT JOIN Employee M
ON E.ManagerId = M.EmployeeId
```

Self Join Examples in SQL Server:

The above Employee table contains the rows for both normal employees as well as managers of that employee. So in order to find out the managers of each employee, we need a SQL Server Self Join. So here we need to write a query that should give the following result.

Employee	Manager
Pranaya	Preety
Priyanka	Pranaya
Preety	NULL
Anurag	Pranaya
Sambit	Pranaya

Cross Join

Cross Join?

When we **combine two or more tables with each other without any condition** (where or on) then **we call this type of joins Cartesian or cross join**. In Cross Join, each record of a table is joined with each record of the other table involved in the join. In SQL Server, the Cross Join should not have either ON or where clause.

A Cross Join produces the Cartesian product of the tables involved in the join. **The Cartesian product means the number of records present in the first table is multiplied by the number of records present in the second table.**

```
--Cross Join

SELECT  Cand.CandidateId,
  Cand.FullName,
  Cand.CompanyId,
  Comp.CompanyId,
        Comp.CompanyName
FROM Candidate Cand
CROSS JOIN Company Comp
```

Subquery

Subquery

- A query within another SQL query and embedded within the WHERE clause.
- Subquery must be enclosed within parenthesis ( ).
- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the comparison operators.
- A subquery can have only one column in the SELECT statement.

Syntax

```
SELECT column_name(s) FROM table_name
WHERE column_name OPERATOR
    (SELECT column_name FROM table_name [WHERE condition]);
```

Example

```
SELECT emp_name, emp_salary FROM employee
WHERE emp_salary =
    (SELECT emp_salary FROM employee WHERE emp_name = 'MIKE');
```

```
SELECT emp_name, emp_salary FROM employee
WHERE emp_salary IN
    (SELECT emp_salary FROM employee WHERE emp_name = 'MIKE');
```

EMPLOYEE

EMP_ID	EMP_NAME	EMP_SALARY	EMP_DEPTID
1111	STEVE	35000	D1
1112	ADAM	28000	D2
1113	JOHN	50000	D3
1114	MIKE	45000	D2
1115	PETER	60000	D5

DEPARTMENT

DEPT_ID	DEPT_NAME	DEPT_LOCATION
D1	DEVELOPMENT	CALIFORNIA
D2	MARKETING	MUMBAI
D3	ACCOUNTS	NEW YORK
D4	MANAGEMENT	SYDNEY

? **Display name, salary of the employees whose salary is greater than Mike's salary.**

? **Display name, salary of the employees whose salary is greater than Adam's salary and deptno same as Adam's deptno.**

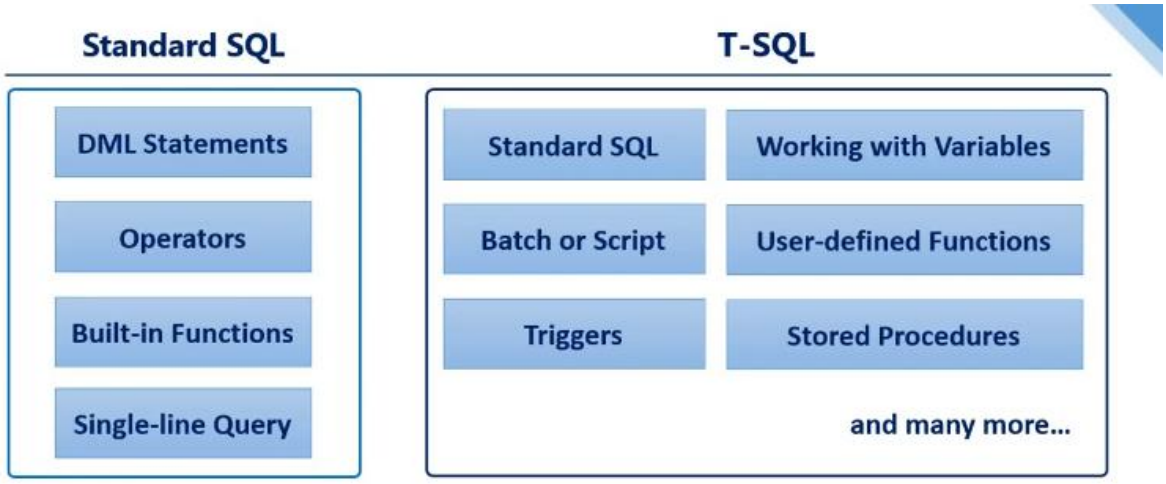
? **Display the employee information whose department is located at New York.**

```
SELECT * FROM employee;

--Display name, salary of employee whose salary is greater than Mike's salary --

SELECT emp_name, emp_salary FROM employee
WHERE emp_salary > (SELECT emp_salary FROM employee WHERE emp_name='Mike');

--Display name, salary of employee whose salary is greater than Adam's salary --
-- and deptno same as Adam's deptno.

WHERE emp_salary > (SELECT emp_salary FROM employee WHERE emp_name = 'Adam')
AND emp_deptid = (SELECT emp_deptid FROM employee WHERE emp_name = 'Adam');
```

Introduction to T-SQL

Transact-SQL

- Transact-SQL is better known as **T-SQL**.
- The purpose of T-SQL is used **to provide a set of tools for the development of a transactional database**.



Working with Variables

- In every programming language, variables are generally used to temporarily store values in memory.
- T-SQL variables are created with **DECLARE** command followed by **variable name** preceded with **@** symbol and **data type**.
- By default, the value of declared variable is **NULL**.

```
DECLARE @variable_name datatype(size);
```

```
DECLARE @name VARCHAR(50);

DECLARE @name VARCHAR(50), @age INT;
```

Assign a Value to a Variable

- Both the **SET** and the **SELECT** command can assign the value to a variable.
- **SET** can only set the value of **one variable** at a time;
- **SELECT** command retrieve data from tables and assign **multiple variables** values with a single statement.

```
SET @variable_name = value;

SELECT  @variable_name = value, @variable_name = value;
```

```
SET @salary = 30000;

SELECT @name = 'ishwar', @age = 25;
```

Incrementing Variable

- With the increment variable feature, we can perform **mathematical operations (like addition, subtraction, and multiplication)** on the variable.

```
SET @number += 10;
SET @number = @number + 10;

SET @number *= 10;
SET @number = @number * 10;

SET @number -= 10;
SET @number = @number - 10;
```

Script

Learned how to generate Scripts from database

Batch in T-SQL

Batch

- A batch of SQL statements is **a group of two or more SQL statements** or **a single SQL statement**.
- A batch of SQL statement can have :
  - ✓ Data Definition Language (DDL) Statements
  - ✓ Data Manipulation Language (DML) Statements
  - ✓ Data Control Language (DCL) Statements

Standard Types of Batches

1. **Explicit Batch**

   An ***explicit batch*** is two or more SQL statements separated by semicolons (;).

   For example,

   ```
   insert into employee (emp_name, emp_salary) values('Brad',45000);
   insert into employee (emp_name, emp_salary) values('Joe',36000);
   ```

2. **Procedure**

   If a ***procedure*** contains more than one SQL statement, it is considered to be a batch.

Go Command

GO Command

- GO is not a T-SQL statement; it is a command recognized by SQL Server utilities.
- GO can be executed by any user. It requires no permissions.
- It signals the end of a batch to the SQL Server utilities.

Syntax **GO [count]**

where, count is a positive integer. The batch will execute the specified number of times.

Example

The following example creates two batches.

```
USE COMPANY_DB;
GO

DECLARE @Name VARCHAR(50);
SELECT @Name = 'Microsoft';
GO
```

- The first batch contains only a **USE COMPANY_DB** statement to set the database.

- The remaining statements use a local variable. Therefore, all local variable declarations must be grouped in a single batch. This is done by not having a GO command until after the last statement that references the variable.

Control Flow Keywords

Control of Flow

- **Transact-SQL (T-SQL) statements are executed in sequential order** (suppose, we have created three statements then, **first statement will run, followed by second, followed by third**.).
- However, in many cases, we will want to interrupt this normal flow.
- **T-SQL has keywords to control the order of execution.**

Control-of-flow Keywords

BEGIN...END	IF...ELSE	WHILE
BREAK	CONTINUE	GOTO
RETURN	TRY...CATCH	THROW
WAITFOR		

BEGIN...END

BEGIN...END

- The **BEGIN...END** keywords are used to group multiple lines into one Statement Block.
- In addition, **BEGIN...END can be nested**. It simply means that we can place a **BEGIN...END** statement within another **BEGIN... END** statement.

Syntax

```
BEGIN
    { SQL Statements or Statement Block }
END
```

Example of Nesting BEGIN...END

```
BEGIN
    DECLARE @name VARCHAR(50), @salary INTEGER,
            @DeptID VARCHAR(10) = 'D3';
    SELECT @name = emp_name, @salary = emp_salary FROM employee
    WHERE emp_deptid = @DeptID;
    SELECT @name 'Name', @salary 'Salary';
    BEGIN
        PRINT 'Department ID : ' + @DeptID;
    END
END
```

IF...ELSE

IF...ELSE

- The **IF...ELSE** statement is a control-flow statement that allows you to execute or skip a statement block based on a specified condition.

Syntax : IF statement

```
IF condition
BEGIN
    { SQL Statements or Statement Block }
END
```

Example of IF...ELSE

```
BEGIN
    DECLARE @salary DECIMAL;
    SELECT @salary = AVG(emp_salary) FROM employee;
    SELECT @salary AS 'Avg. Salary';
    IF @salary > 25000
        BEGIN
            PRINT 'Avg. salary is greater than 25000';
        END
    ELSE
        BEGIN
            PRINT 'Avg. salary is less than 25000';
        END
END
```

WHILE Loop

WHILE

- The **WHILE** loop statement is a control-flow statement that allows you to execute a statement block repeatedly as long as a specified condition is **TRUE**.
- The execution of statements in the **WHILE** loop can be controlled from inside the loop with the **BREAK** and **CONTINUE** keywords.

Syntax

```
WHILE condition
BEGIN
    { SQL Statements or Statement Block }
END
```

Example

```
BEGIN
    WHILE ( SELECT MIN(emp_salary) FROM employee ) < 80000
    BEGIN
        UPDATE employee SET emp_salary = emp_salary + 10000;
        PRINT 'Salary updated';
        IF ( SELECT MIN(emp_salary) FROM employee) >= 80000
            PRINT 'Min. Salary is greater or equal to 80000.';
            BREAK;
    END
END
```

TRY...CATCH

With the introduction of Try/Catch blocks in SQL Server 2005, the error handling in the SQL server is now very much similar to programming languages like C#, and Java. But, before understanding the error handling using the try/catch block, let's step back and understand how error handling was done in SQL Server prior to 2005, using the system function RAISERROR and @@Error.

Exception Handling Using RAISERROR System Function in SQL Server:

The system-defined Raiserror() function returns an error message back to the calling application. The RaiseError System defined Function in SQL Server takes 3 parameters as shown below.

RAISERROR('Error Message', ErrorSeverity, ErrorState)

@@Error System Function in SQL Server:

In SQL Server 2000, in order to detect errors, we use the @@Error system function. The @@Error system function returns a NON-ZERO value if there is an error, otherwise, ZERO

indicates that the previous SQL statement was executed without any error. Let's modify the stored procedure to make use of the @@ERROR system function

TRY...CATCH

- **TRY...CATCH** implements **error handling** for T-SQL.
- It is similar to the exception handling in the object-oriented programming languages such as C++, Java, JavaScript, etc.
- A group of T-SQL statements can be enclosed in a **TRY** block.
- If the statements between the **TRY** block **complete without an error**, the statements between the **CATCH block will not execute**. However, if any statement inside the **TRY** block **causes an exception**, the **control transfers to the statements in the CATCH** block.

Syntax

```
BEGIN TRY
    { SQL Statement or Statement Block }
END TRY
BEGIN CATCH
    [ { SQL Statement or Statement Block } ]
END CATCH
```

To retrieve the information about the error :

- **ERROR_MESSAGE()** returns the complete text of the generated error message.
- **ERROR_NUMBER()** returns the number of the error.
- **ERROR_LINE()** returns the line number inside the routine that caused the error.
- **ERROR_PROCEDURE()** returns the name of the stored procedure or trigger where the error occurred.
- **ERROR_STATE()** returns the error state number.
- **ERROR_SEVERITY()** returns the severity.

```
BEGIN TRY
    SELECT 100/0 AS 'Division';
END TRY
BEGIN CATCH
    SELECT ERROR_MESSAGE() AS 'Error Message', ERROR_LINE() AS 'Error Line',
    ERROR_NUMBER() AS 'Error Number', ERROR_PROCEDURE() AS 'Procedure Name';
END CATCH
```

WAITFOR

WAITFOR

- **WAITFOR** blocks the execution of a batch, stored procedure, or transaction until either a specified time or time interval elapses, or a specified statement modifies or returns at least one row.
- **WAITFOR** has two arguments :

TIME	the period of time to wait. time_to_pass
DELAY	the time (up to a maximum of 24 hours) at which the WAITFOR statement finishes.

Syntax

```
BEGIN
    WAITFOR TIME 'time_to_execute'
    { SQL Statement or Statement Block }
END
```

where, **time_to_execute** can be specified either in a **datetime** data format, or as a **local variable**.

Syntax

```
BEGIN
    WAITFOR DELAY 'time_to_pass'
    { SQL Statement or Statement Block }
END
```

Example

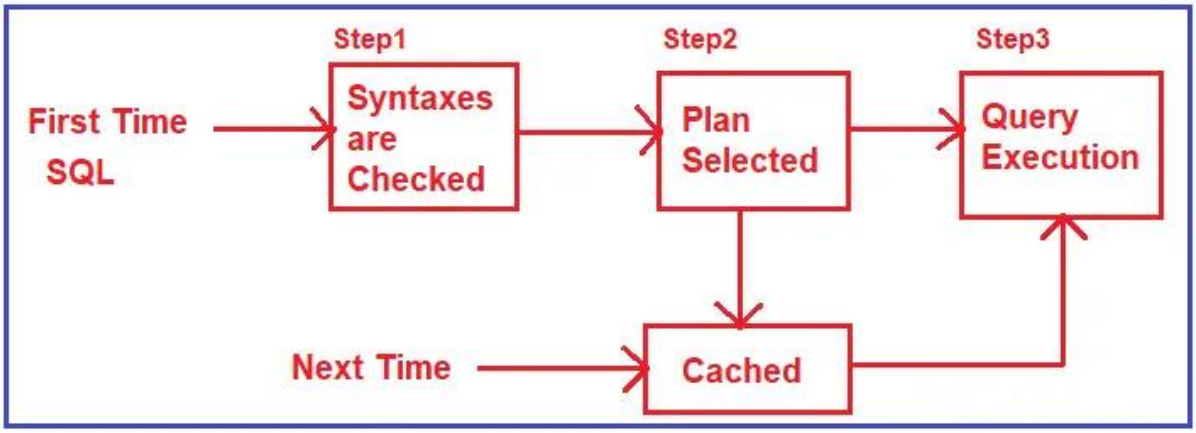
```
BEGIN
    WAITFOR TIME '18:00:00'
    SELECT * FROM employee;
END
```

```
BEGIN
    WAITFOR DELAY '00:00:10'
    SELECT * FROM employee;
END
```

Stored procedure in T-SQL

STORED PROCEDURE

- A **stored procedure** is a group of one or more T-SQL statements.
- It can be stored in the database.
- **Stored procedures**
  - ✓ accept input parameters and return multiple values
  - ✓ contain programming statements that perform operations in the database.
  - ✓ return a status value to a calling program to indicate success or failure



What is a Stored Procedure in SQL Server?

A SQL Server Stored Procedure is a database object which contains pre-compiled queries (a group of T-SQL Statements). In other words, we can say that the Stored Procedures are a block of code designed to perform a task whenever we called.

BENEFITS

- ✓ Reuse of code
- ✓ Easy to maintain
- ✓ Improve performance
- ✓ Strong security
- ✓ Reduce server/client network traffic

TYPES

- ✓ **System**
  - physically stored in the internal **Resource** database
- ✓ **User-defined**
  - It can created in a user-defined database or in all system databases except the **Resource** database.
- ✓ **Temporary**
  - a form of user-defined procedures. The temporary procedures are like a permanent procedure, except temporary procedures are stored in **tempdb.**

Create Stored Procedure in T-SQL

STORED PROCEDURE

- Two ways to create (or define) a stored procedure
  1. **Stored Procedure without Parameter (Simple Stored Procedure)**
  2. **Stored Procedure with Parameter**

Syntax (Simple Procedure)

```
CREATE PROCEDURE procedure_name
AS
BEGIN
    { SQL Statement or Statement Block }
END
```

Syntax (Stored Procedure with parameter)

```
CREATE PROCEDURE procedure_name(parameter list)
AS
BEGIN
    { SQL Statement or Statement Block }
END
```

Different Types of Parameters in SQL Server Stored Procedure.

1. Input parameters
2. Output parameters

Example: Create a stored procedure that returns all employees whose department location is Mumbai.

```
SELECT * FROM employee e
inner join department d
ON e.emp_deptid = d.deptid
WHERE dept_location = 'mumbai';
```

```
CREATE PROCEDURE
proc_allEmployeesDetails(@location AS VARCHAR(100))
AS
BEGIN
   SELECT * FROM employee e inner join department d
   ON e.emp_deptid = d.deptid
   WHERE dept_location = @location;
END
```

```
--Out Parameter

CREATE PROCEDURE spGetResult
  @No1 INT,
  @No2 INT,
  @Result INT OUTPUT
AS
BEGIN
  SET @Result = @No1 + @No2
END
```

Incorrect syntax: 'CREATE PROCEDURE' must be the only statement in the batch.

```
---- To Execute Procedure
DECLARE @Result INT
EXECUTE spGetResult 10, 20, @Result OUT
PRINT @Result

--Stored Procedure with Default Values

CREATE PROCEDURE spAddNumber(@No1 INT= 100, @No2 INT)
```

```
Messages
30

Completion time: 2023-05-31T23:14:43.5018482+05:30
```

```
--Stored Procedure with Default Values

CREATE PROCEDURE spAddNumber(@No1 INT= 100, @No2 INT)
AS
BEGIN
  DECLARE @Result INT
  SET @Result = @No1 + @No2
  PRINT 'The SUM of the 2 Numbers is: '+ CAST(@Result AS VARCHAR)
END
```

```
EXEC spAddNumber 3200, 25
EXEC spAddNumber @No1=200, @No2=25
EXEC spAddNumber @No1=DEFAULT, @No2=25
EXEC spAddNumber @No2=25


--Functions

--Scalar Valued Function
```

```
Messages
The SUM of the 2 Numbers is: 3225

Completion time: 2023-05-31T23:15:54.1925125+05:30
```

How to call a Stored Procedure in SQL Server?

Once we create the stored procedure, then it is physically stored on the server as a “database object” which can be called from anywhere connecting to the server.

```
EXECUTE ProcedureName VALUES
OR
EXEC ProcedureName VALUES
OR
ProcedureName VALUES
```

Alter (Modify) a Stored Procedure

Syntax (Alter Procedure)

```
ALTER PROCEDURE procedure_name
AS
BEGIN
    { SQL Statement or Statement Block }
END
```

Example: Modify an existing simple stored procedure proc_allEmployeesDetails.

```
CREATE PROCEDURE
proc_allEmployeesDetails
AS
BEGIN
    SELECT * FROM employee;
END
```

```
ALTER PROCEDURE proc_allEmployeesDetails
AS
BEGIN
    SELECT e.emp_name, e.emp_salary, d.dept_location
    FROM employee e
    inner join department d
    ON e.emp_deptid = d.dept_id;
END
```

Example: Modify an existing parameterized stored procedure proc_employeeDetailsLocationWise.

```
CREATE PROCEDURE
proc_employeeDetailsLocationWise(@location AS VARCHAR(100))
AS
BEGIN
        SELECT * FROM employee e
        inner join department d
        ON e.emp_deptid = d.dept_id
        WHERE d.dept_location = @location;
END;
```

```
ALTER PROCEDURE
proc_employeeDetailsLocationWise(@location AS VARCHAR(100))
AS
BEGIN
        SELECT e.emp_name, e.emp_salary, d.dept_location
        FROM employee e
        inner join department d
        ON e.emp_deptid = d.dept_id
        WHERE d.dept_location = @location;
```

Rename a Stored Procedure

Rename a Stored Procedure

- To rename the existing stored procedure, we need to use **system** procedure

sp_rename

Syntax (Rename Procedure)

```
EXEC sp_rename 'COMPANY_DB.proc_allEmployeeDetails', 'proc_displayEmployeeDetails';
```

Drawbacks (or Limitations)

- Renaming a stored procedure does not change the name of the corresponding object name in the definition column of the **sys.sql_modules** catalog view. To do that, you must drop and re-create the stored procedure with its new name.
- Changing the name or definition of a procedure can cause dependent objects to fail when the objects are not updated to reflect the changes that have been made to the procedure.

User-defined Functions in T-SQL

User-Defined Functions (UDFs)

- **User-defined functions (UDFs)** are routines that accept parameters, perform an action (complex calculation), and return the result of that action as a value. The return value can either be a single scalar value or a result set.

Why UDFs

- ✓ Modular Programming
- ✓ Faster execution
- ✓ Reduce network traffic
- ✓ **Modular Programming**
  - create the function once,
  - store it in the database, and
  - call it any number of times in your program.
- ✓ **Faster execution**
  - reduce the compilation cost of Transact-SQL code

    (UDFs does not need to be reparsed and reoptimized with each use resulting in much faster execution times).
- ✓ **Reduce network traffic**
  - function can be invoked in the WHERE clause to reduce the number of rows sent to the client.

TYPES

- ✓ **System**
  - SQL Server provides many system functions that you can use to perform a variety of operations. They cannot be modified
- ✓ **Scalar**
  - Scalar functions return a single data value of the type defined in the **RETURNS** clause.
- ✓ **Table-Valued**
  - table-valued functions return a **table** data type.

Before creating a function, things to know...

- User-defined function always returns a value.
- User-defined functions have only input parameters for it.
- User-defined functions can not return multiple result sets.
- Error handling is restricted in a user-defined function. A UDF does not support **TRY...CATCH**, **@ERROR** or **RAISERROR**.
- SET statements are not allowed in a user-defined function.
- User-defined functions cannot call a stored procedure.
- User-defined functions can be nested. User-defined functions can be nested up to 32 levels.

Scalar-Valued Function (Create)

Syntax (Scalar Function)

```
CREATE FUNCTION function_name(parameter datatype, . . . )
RETURNS return_datatype
[ WITH <function_option> [ ,...n ] ]
[ AS ]
    BEGIN
            function_body
            RETURN scalar_expression
    END;
```

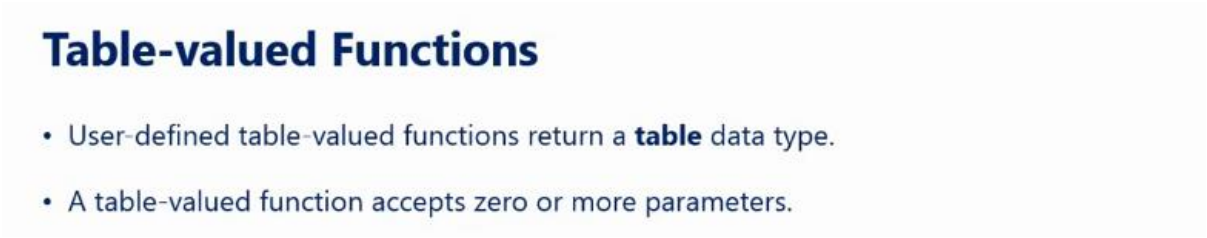
Example: Create a function to get employee salary by passing employee name.

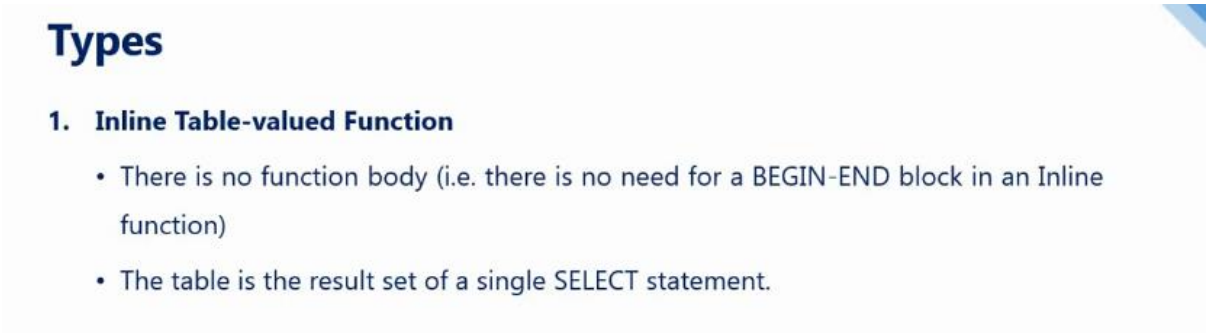
```
CREATE FUNCTION salary(@name as varchar(50))
RETURNS decimal
BEGIN
    DECLARE @sal decimal;
    SELECT @sal = emp_salary FROM employee
    WHERE emp_name = @name;

    RETURN @sal;
END
```

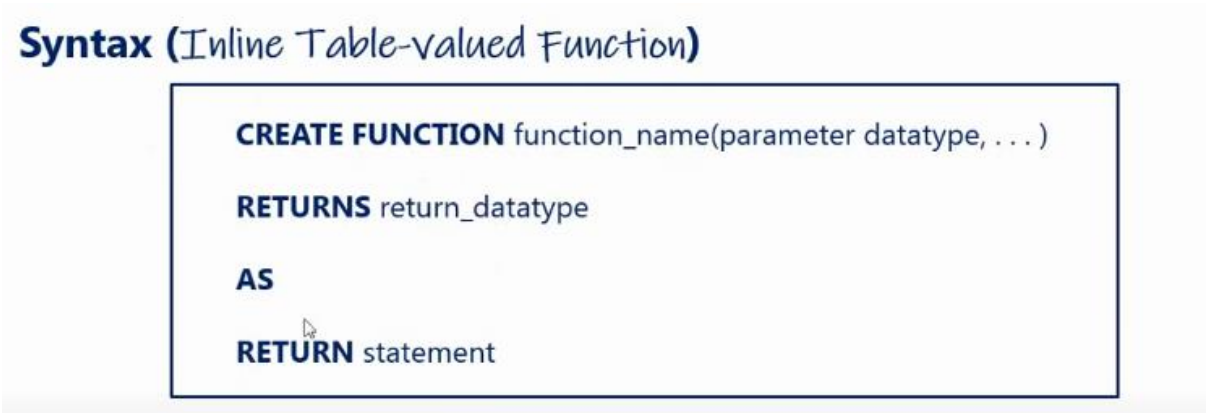
```
--Scalar Valued Function

CREATE FUNCTION SVF1(@X INT)
RETURNS INT
AS
BEGIN
  RETURN @X * @X *@X
END

--Execute Function

SELECT dbo.SVF1(5)
```

A stored procedure can also accept the DOB of an employee and return the age but we cannot use a stored procedure in a select clause or where clause. This is one of the differences between a function and a stored procedure.

Table-valued Function in T-SQL







Example: Create a function to get employee information by passing employee salary.

```
CREATE FUNCTION getAllEmployees(@salary decimal)
RETURNS TABLE
AS
RETURN
    SELECT * FROM employee WHERE emp_salary = @salary;
```

To execute the function,

```
SELECT * FROM getAllEmployees(20000);
```

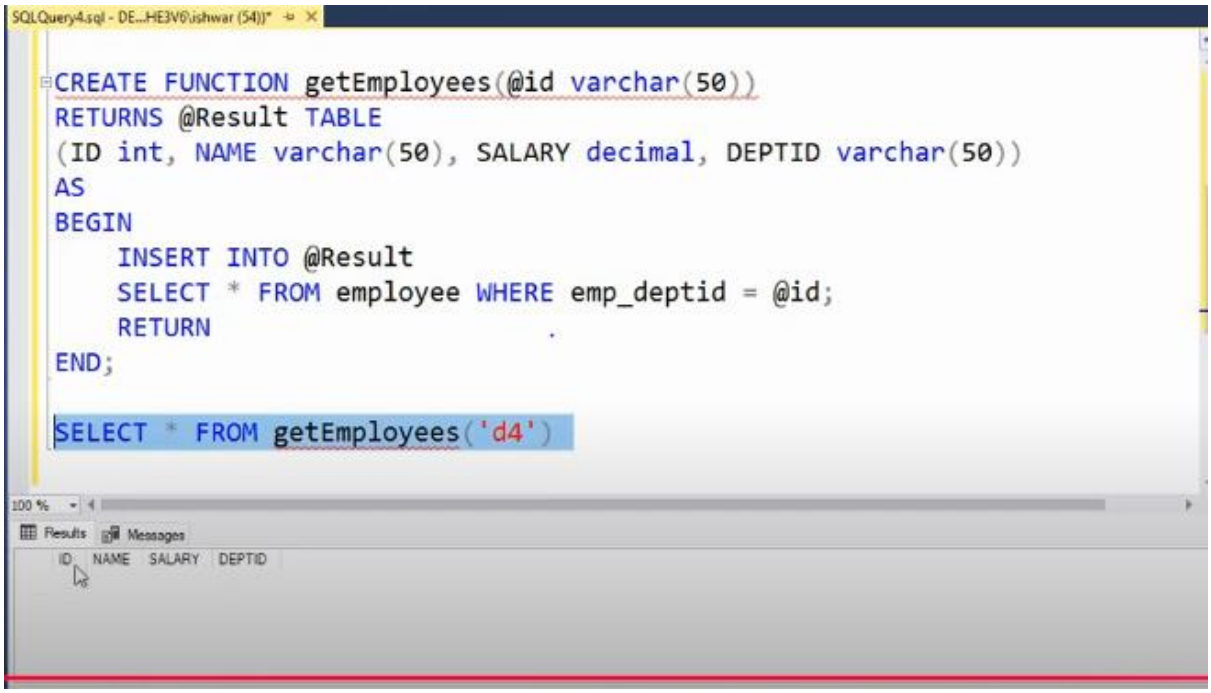
2. **Multi-statement Table-valued Function**
   - It contains multiple SQL statements enclosed in **BEGIN-END** blocks.
   - The return value is declared as a **table variable**. The **RETURN** statement is without a value and the declared table variable is returned.

Syntax (Multi-Statement Table-valued Function)

```
CREATE FUNCTION function_name(parameter datatype, . . . )
RETURNS @table_variable TABLE
(column_1 datatype, column_2 datatype, . . .)
AS
    BEGIN
        SQL-statement(s)
        RETURN
    END;
```

Example: Create a function to get list of employees by passing department id.

```
CREATE FUNCTION getAllEmployees(@id varchar(50))
RETURNS @Table TABLE
(ID int, NAME varchar(50), SALARY decimal, DEPTID varchar(50))
AS
BEGIN
    INSERT INTO @Table
    SELECT * FROM employee WHERE emp_deptid = @id;
    RETURN
END;
```



What is the Difference Between Functions and Procedures in SQL Server?

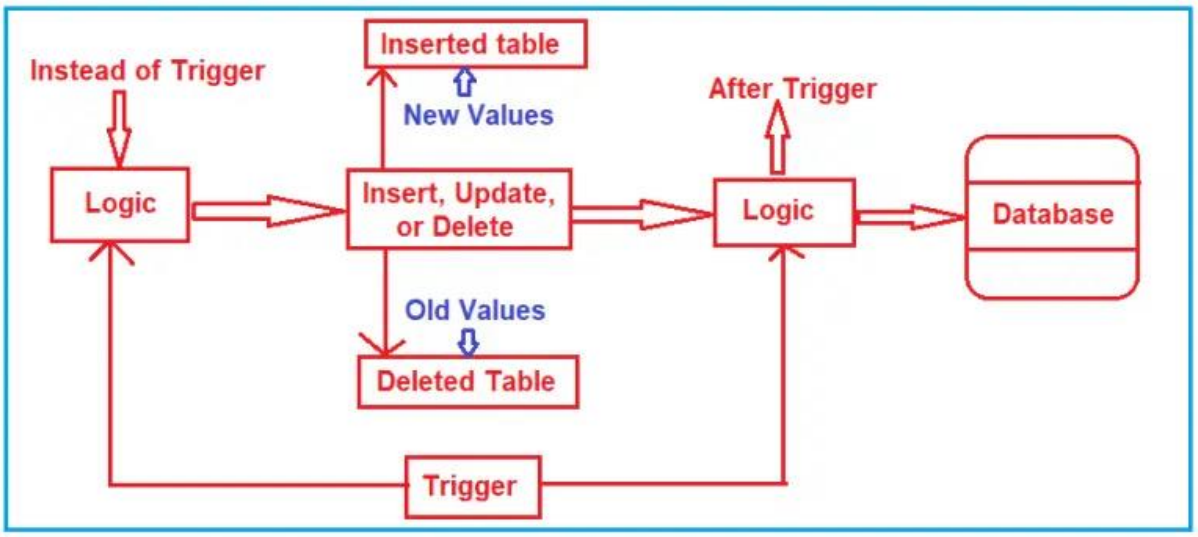
1. A function must return a value, it is mandatory whereas a procedure returning a value is optional.
2. The procedure can have parameters of both input and output whereas a function can have only input parameters.
3. In a procedure, we can perform Select. Update, Insert and Delete operations whereas function can only be used to perform select operations. It cannot be used to perform Insert, Update, and Delete operations that can change the state of the database.

4. A procedure provides the options to perform Transaction Management, Error Handling, etc whereas these operations are not possible in a function.
5. We call a procedure using EXECUTE/ EXEC command whereas a function is called by using SELECT command only.
6. From a procedure, we can call another procedure or a function whereas from a function we can call another function but not a procedure.
7. User-Defined Functions can be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section where as Stored procedures cannot be.

Introduction to Triggers in T-SQL

Trigger

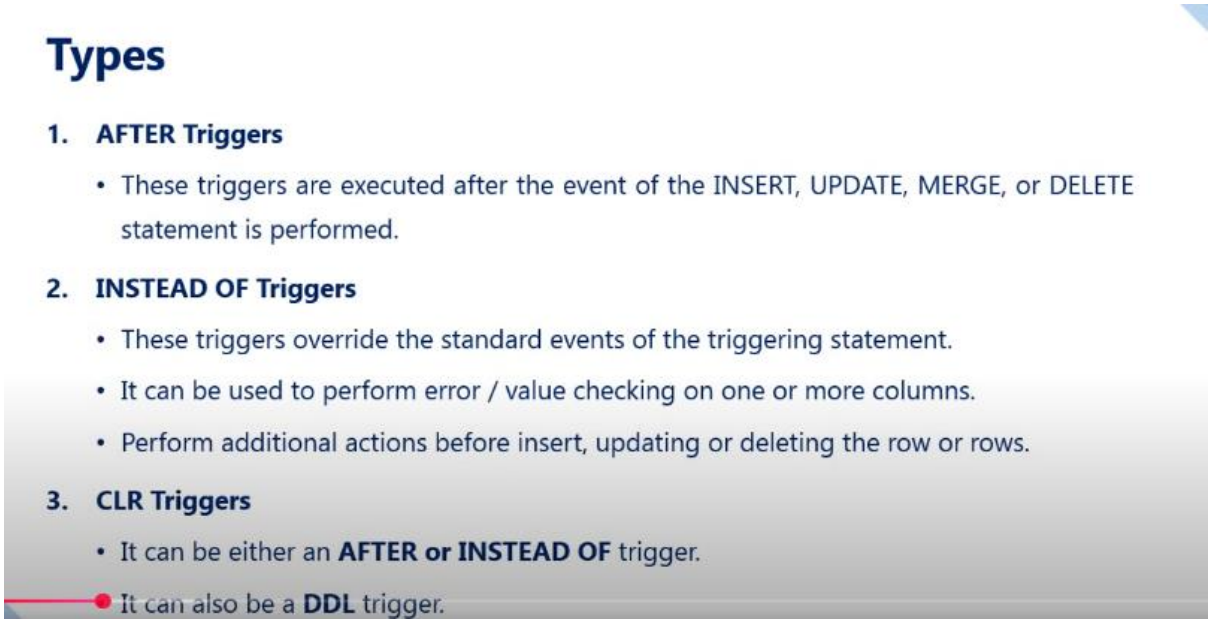
- A trigger is a type of stored procedure, that automatically runs when an event occurs in the database server.
- Here, events are DML operations (INSERT, DELETE, UPDATE).



DML Trigger:

DML Trigger

- A trigger is a type of stored procedure, that automatically runs when an event occurs in the database server.



Types of Triggers in SQL Server:

There are four types of triggers available in SQL Server. They are as follows:

1. **DML Triggers** – Data Manipulation Language Triggers.
2. **DDL Triggers** – Data Definition Language Triggers
3. **CLR triggers** – Common Language Runtime Triggers
4. **Logon triggers**

In this article, we are going to discuss the DML triggers and the rest are going to discuss in our upcoming articles

Syntax

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH DML_trigger_option ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] , [ UPDATE ] , [ DELETE ] }
AS
BEGIN
    { SQL Statement or Statement Block }
END;
```

Let us understand the syntax in detail.

1. **ON TableName or ViewName**: It refers to the table or view name **on which we are defining the trigger.**
2. **For/After/InsteadOf:** The **For/After** option **specifies that the trigger fires only after the SQL statements are executed** whereas the InsteadOf option specifies that the trigger is executed on behalf of the triggering SQL statement. You ***cannot create After Trigger on views.***
3. INSERT, UPDATE, DELETE: T**he INSERT, UPDATE, DELETE specify which SQL statement will fire this trigger** and **we need to use at least one option or the combination of multiple options is also accepted.**

Note: The Insert, Update and Delete statements **are also known as Triggering SQL statements** as these statements are responsible for the trigger to fire.

Before creating a trigger, things to know...

Magic Tables

SQL Server automatically creates and manages magic tables. DML trigger statements use two magic tables.

1. **Inserted Table**
   - This table stores copies of the affected rows during INSERT and UPDATE statements. During these transactions, **new rows are added to both the inserted table and the trigger table**.
2. **Deleted Table**
   - This table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of these statements, **rows are deleted from the trigger table and stored into the deleted table**.

Example: Create a trigger when new employee added to employee table.

```
CREATE TRIGGER tr_message
ON employee
AFTER INSERT
AS
BEGIN
    PRINT 'New employee added to the Employee Table';
END;
```

```
AFTER INSERT
AS
    BEGIN
        INSERT INTO employeeLogs
        SELECT emp_id, emp_name, emp_salary, emp_deptid, 'ishwar', getdate()
        FROM inserted;
    END;
```

****** inserted is a magic table ********

Instead of Trigger

What is Instead Of Triggers in SQL Server?

The INSTEAD OF triggers are the DML triggers that are fired instead of the triggering event such as the INSERT, UPDATE or DELETE events. So, when you fire any DML statements such as Insert, Update, and Delete, then on behalf of the DML statement, the instead of trigger is going to execute. In real-time applications, **Instead Of Triggers are used to correctly update a complex view.**

Example to understand Instead of Triggers in SQL Server:

We are going to **use the Department and Employee table to understand the complex views in SQL Server.**

Department

ID	Name
1	IT
2	HR
3	Sales

Employee

ID	Name	Gender	DOB	Salary	DeptID
1	Pranaya	Male	1996-02-29 10:53:27.060	25000.00	1
2	Priyanka	Female	1995-05-25 10:53:27.060	30000.00	2
3	Anurag	Male	1995-04-19 10:53:27.060	40000.00	2
4	Preety	Female	1996-03-17 10:53:27.060	35000.00	3
5	Sambit	Male	1997-01-15 10:53:27.060	27000.00	1
6	Hina	Female	1995-07-12 10:53:27.060	33000.00	2

create and populate the Department and Employee table with sample data.
We want to retrieve the following data from the Employee and Department table.

let's create a view that will return the above results.

let's try to insert a record into the view vwEmployeeDetails by executing the following query.

INSERT INTO vwEmployeeDetails VALUES(7, 'Saroj', 'Male', 50000, 'IT')

When we execute the above query it gives us the error as 'View or function vwEmployeeDetails is not updatable because the modification affects multiple base tables.'

How to overcome the above problem?

By using Instead of Insert Trigger.

DDL Trigger

DDL Trigger

- DDL triggers fire in response to a different **DDL events** correspond to SQL statements such as **CREATE**, **ALTER**, **DROP**, **GRANT**, **REVOKE** etc.
- Also, some system stored procedures that perform **DDL-like operations** (for example, **sp_rename**) can also fire DDL triggers.

- **DDL events Reference: https://bit.ly/3I1EHay**

Why DDL Trigger?

- Prevent certain changes to your database schema.
- Have something occur in the database in response to a change in your database schema.
- Record changes or events in the database schema.

Syntax

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH DDL_trigger_option ]
{ FOR | AFTER } { event_type1, event_type2, . . . }
AS
BEGIN
    { SQL Statement or Statement Block }
END;
```

Example: Create a trigger when new table is created in a database.

```
CREATE TRIGGER tr_onTableCreate
ON DATABASE
FOR CREATE_TABLE
AS
BEGIN
    PRINT 'New table is created successfully';
END;
```

```
--Create Server Scopped DDL Trigger

CREATE TRIGGER trServerScopedDDLTrigger
ON ALL SERVER
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   PRINT 'You cannot create, alter or drop a table in any database of this server'
   ROLLBACK TRANSACTION
END
```

EVENTDATA( )

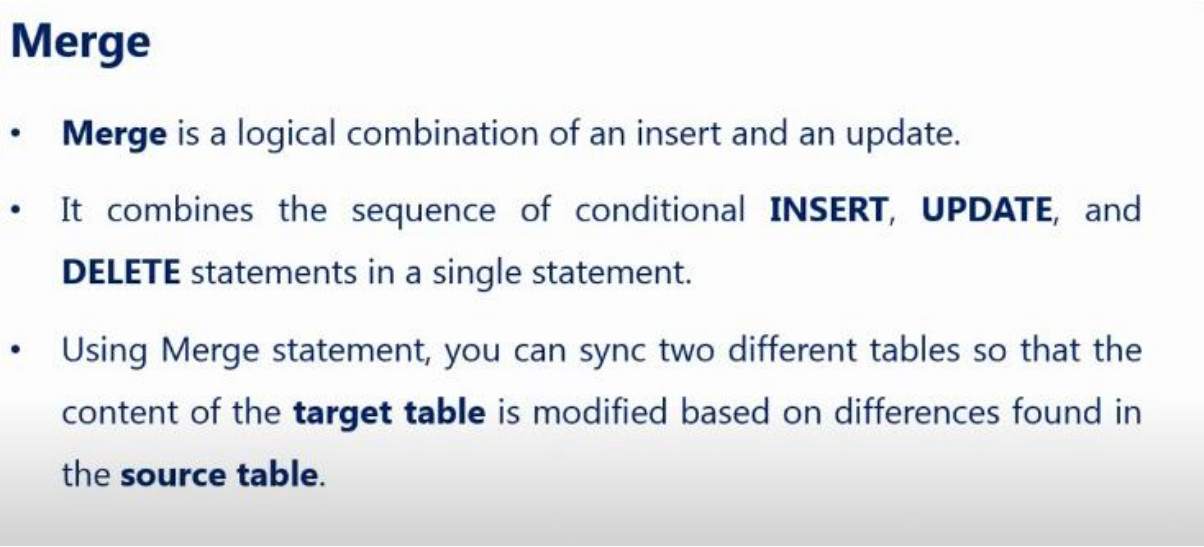
- It is a built-in function.
- It returns the information about the events executed the DDL trigger.
- The information is in XML format.

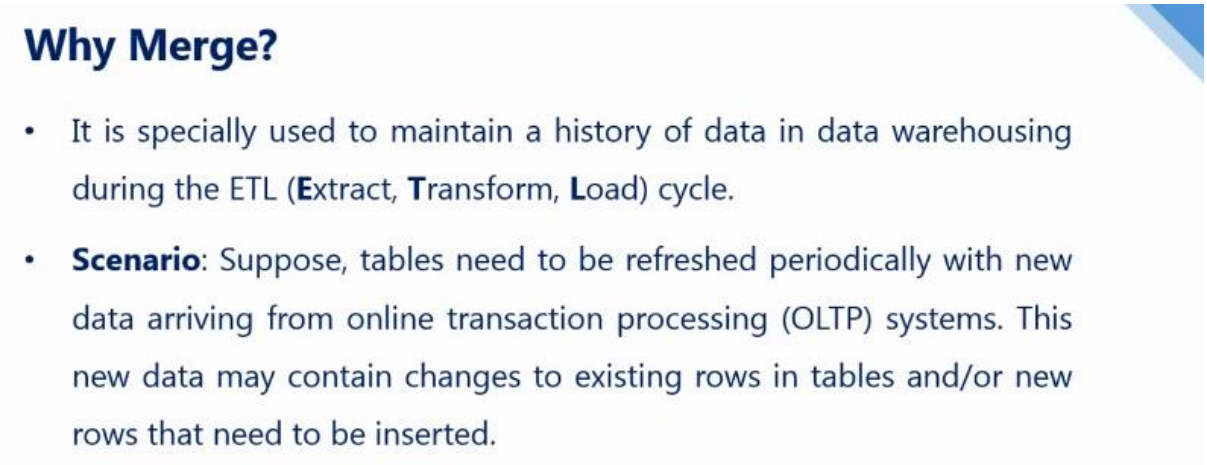
- **EventType** (Create Table, Alter Table, Drop Table, etc...)
- **PostTime** (Event trigger time)
- **SPID** (SQL Server session ID)
- **ServerName** (SQL Server instance name)
- **LoginName** (SQL Server Login name)
- **UserName** (username for login, by default dbo schema as username)
- **DatabaseName** (name of database where DDL Trigger was executed)
- **SchemaName** (schema name of the table)
- **ObjectName** (Name of the table)
- **ObjectType** (Object types. such as Table, view, procedure, etc...)
- **TSQLCommand** (Schema deployment Query which is executed by user)
- **SetOptions** (SET Option which are applied while Creating table or Modify it)
- **CommandText** (Create, Alter or Drop object command)

```
CREATE TABLE ddl_logs(
    id INT IDENTITY PRIMARY KEY,
    event_data XML,
    performed_by SYSNAME,
    event_type VARCHAR(200)
);

CREATE OR ALTER TRIGGER tr_ddlEventTrigger
ON DATABASE
FOR CREATE_TABLE, RENAME, DROP_TABLE
AS
    BEGIN
        INSERT INTO ddl_logs(event_data, performed_by)
        VALUES (EVENTDATA(), USER);
    END;
```

```
CREATE OR ALTER TRIGGER tr_ddlEventTrigger
ON DATABASE
FOR CREATE_TABLE, RENAME, DROP_TABLE
AS
BEGIN
    INSERT INTO ddl_logs(event_data, performed_by, event_type)
    VALUES (EVENTDATA(), USER, EVENTDATA().value('(/EVENT_INSTANCE/EventType)[1]', 'varchar(max)'));
END;
```

MERGE Statement in MS SQL Server





Source Table

CHECK-IN

FIRSTNAME	FLIGHTCODE	FLIGHTDATE	SEAT
STEVE	SQL2022	2022-03-27	7F
ADAM	SQL2022	2022-03-27	19A
JOHN	SQL2022	2022-03-27	4B
MIKE	SQL2022	2022-03-27	20A

Target Table

FLIGHT-PASSENGER

FLIGHTID	FIRSTNAME	FLIGHTCODE	FLIGHTDATE	SEAT
1	STEVE	SQL2022	2022-03-27	7F
2	ADAM	SQL2022	2022-03-27	19A
3	JOHN	SQL2022	2022-03-27	20B
4	MIKE	SQL2022	2022-03-27	2A

Actions / Conditions

1. Rows in the **source table** are not found in the **target table**. Then, rows from the source will be added to the target table.
2. Rows in the **target table** are not found in the **source table**. Then, delete rows from the target table.
3. Rows in the **source and target table** have the same keys but, they have different values in the non-key columns. Then, update the rows in the target table with data from the source table.

Syntax

```
MERGE TargetTable
USING SourceTable
ON join-conditions
WHEN Matched
    THEN DML Statement
WHEN NOT MATCHED BY TARGET
    THEN DML Statement
WHEN NOT MATCHED BY SOURCE
    THEN DML Statement;
```

Note

It must be terminated by a semicolon.

States

1. **MATCHED**: These are the rows that match the merge condition. For the matching rows, you need to update the rows columns in the target table with values from the source table.
2. **NOT MATCHED BY TARGET**: These are the rows from the source table that does not have any matching rows in the target table. In this case, you need to add the rows from the source table to the target table.
3. **NOT MATCHED BY SOURCE**: These are the rows in the target table that does not match any rows in the source table. If you want to synchronize the target table with the data from the source table, then you will need to use this match condition to delete rows from the target table.

Syntax

MERGE TargetTable
USING SourceTable
ON join-conditions
WHEN Matched
    THEN DML Statement
WHEN NOT MATCHED BY TARGET
    THEN DML Statement
WHEN NOT MATCHED BY SOURCE
    THEN DML Statement;

MERGE TargetTable
USING SourceTable
ON join-conditions
WHEN Matched
    THEN Update
WHEN NOT MATCHED BY TARGET
    THEN Insert
WHEN NOT MATCHED BY SOURCE
    THEN Delete;

Example:

```
CREATE TABLE flightPassengers
(
    flightId INT IDENTITY PRIMARY KEY,
    firstName VARCHAR(20) NOT NULL,
    flightCode VARCHAR(20) NOT NULL,
    flightDate DATE NOT NULL,
    seat VARCHAR(5)
);

INSERT INTO flightPassengers (firstName,flightCode,flightDate,seat)
VALUES ('SMITH','SQL2022', GETDATE(), '7F'),
       ('ADAM','SQL2022', GETDATE(), '20A'),
       ('MIKE','SQL2022', GETDATE(), '48')
```

```sql
CREATE TABLE checkIn
(
    firstName VARCHAR(20) NOT NULL,
    flightCode VARCHAR(20) NOT NULL,
    flightDate DATE NOT NULL,
    seat VARCHAR(5)
);

INSERT INTO checkIn (firstName,flightCode,flightDate,seat)
VALUES ('SMITH','SQL2022', GETDATE(), '7F'),
       ('ADAM','SQL2022', GETDATE(), '2B'),
       ('MIKE','SQL2022', GETDATE(), '17A');
```

```sql
SELECT * FROM flightPassengers;
SELECT * FROM checkIn;
```

100 %

Results | Messages

flightId	firstName	flightCode	flightDate	seat
1	SMITH	SQL2022	2022-03-27	7F
2	ADAM	SQL2022	2022-03-27	20A
3	MIKE	SQL2022	2022-03-27	4B

firstName	flightCode	flightDate	seat
SMITH	SQL2022	2022-03-27	7F
ADAM	SQL2022	2022-03-27	2B
MIKE	SQL2022	2022-03-27	17A

```sql
MERGE flightPassengers F
USING checkIn C
ON C.firstName = F.firstName
AND C.flightCode = F.flightCode
AND C.flightDate = F.flightDate
WHEN MATCHED
    THEN UPDATE SET F.seat = C.seat
WHEN NOT MATCHED BY TARGET
    THEN INSERT (firstName,flightCode,flightDate,seat)
        VALUES (firstName,flightCode,flightDate,seat)
WHEN NOT MATCHED BY SOURCE
    THEN DELETE;
```

```sql
SELECT * FROM flightPassengers;
SELECT * FROM checkIn;

MERGE flightPassengers F
```

100 %

Results | Messages

	flightId	firstName	flightCode	flightDate	seat
1	1	SMITH	SQL2022	2022-03-27	7F
2	2	ADAM	SQL2022	2022-03-27	2B
3	3	MIKE	SQL2022	2022-03-27	17A

	firstName	flightCode	flightDate	seat
1	SMITH	SQL2022	2022-03-27	7F
2	ADAM	SQL2022	2022-03-27	2B
3	MIKE	SQL2022	2022-03-27	17A

Introduction to Index in MS SQL Server

Index

- In general, **index** is used to measure the performance.
- Database systems uses indices to provide fast access to relational data.
- It is a special type of physical data structure used to access one or more data rows fast.
- Database index can change each time the corresponding data is changed.

Syntax

```
CREATE INDEX index_name ON table_name;
```

Example Find employee names whose salary is greater than 50000.

ID	NAME	SALARY
1001	SMITH	65000
1002	JOHN	30000
1003	MIKE	48000
1004	JACK	52000

SALARY	ROW ADDRESS
30000	ROW ADDRESS
48000	ROW ADDRESS
52000	ROW ADDRESS
65000	ROW ADDRESS

```
SELECT * FROM EMPLOYEE
WHERE SALARY > 50000;
```

```
CREATE INDEX IDX_EMPLOYEE_SALARY
ON EMPLOYEE(SALARY ASC);
```

```sql
CREATE TABLE employee
(ID INT,
 NAME VARCHAR(50),
 SALARY DECIMAL,
 LOCATION VARCHAR(50));

insert into employee values
(1003, 'SMITH',65000,'USA'),
(1001, 'JAMES',30000,'INDIA'),
(1002, 'MIKE',48000,'INDIA'),
(1004, 'JOHN',55000,'USA');

SELECT * FROM employee;
DROP TABLE employee
```

Results | Messages

	ID	NAME	SALARY	LOCATION
1	1003	SMITH	65000	USA
2	1001	JAMES	30000	INDIA
3	1002	MIKE	48000	INDIA
4	1004	JOHN	55000	USA

```sql
SELECT * FROM employee;

CREATE INDEX idx_employee_salary
ON employee(SALARY ASC);
```

Results | Messages

	ID	NAME	SALARY	LOCATION
1	1003	SMITH	65000	USA
2	1001	JAMES	30000	INDIA
3	1002	MIKE	48000	INDIA
4	1004	JOHN	55000	USA

Clustered Index in MS SQL Server

Clustered Index

- A **clustered index** determines the physical order of the data in a table. Hence, a table can have only one clustered index.
- When a clustered index is created, the database engine sorts the data in the table based on the defined index key(s) and stores the table in that order.
- **Example**, a telephone book. A telephone book is always in sorted order, based on the last name of the individual followed by the first name. The sorted order makes it easy to find the phone number of the person you are looking for.

Things to remember...

- ✓ A **Primary key** constraint creates clustered index automatically if there is no clustered index exists on the table.
- ✓ An index can contain multiple columns, known as **composite index** (we will see this in later video).

Syntax

```
CREATE CLUSTERED INDEX index_name
ON table_name(column_name <ASC | DESC> );
```

```
SELECT * FROM employee;

CREATE CLUSTERED INDEX idx_employee_name
ON employee(name asc);
```

100 %

Results | Messages

	ID	NAME	SALARY	LOCATION
1	1001	JAMES	30000	INDIA
2	1004	JOHN	55000	USA
3	1002	MIKE	48000	INDIA
4	1003	SMITH	65000	USA

It sorted row based on name column

Composite Clustered INDEX or MULTIPLE Column Clustered Index

```
CREATE CLUSTERED INDEX idx_employee_loc
ON employee(id desc, location asc)
```

100 %

Results | Messages

	ID	NAME	SALARY	LOCATION
1	1004	JOHN	55000	USA
2	1003	SMITH	65000	USA
3	1002	MIKE	48000	INDIA
4	1001	JAMES	30000	INDIA

Nonclustered Index in MS SQL Server

Nonclustered Index

- A **nonclustered index** does not change the physical order of the rows in the table.
- In other (simple) words, A nonclustered index is similar to an index in a textbook. The data is stored in one place, the index in another place. The index will have pointer to the storage location of the data.
- Since, the index is stored separately from the actual data, a table can have more than one nonclustered index.

ID	SALARY	LOCATION	NAME
1003	65000	USA	SMITH
1002	30000	INDIA	JOHN
1001	48000	INDIA	MIKE
1004	52000	USA	JACK

NAME	ROW ADDRESS
JACK	ROW ADDRESS
JOHN	ROW ADDRESS
MIKE	ROW ADDRESS
SMITH	ROW ADDRESS

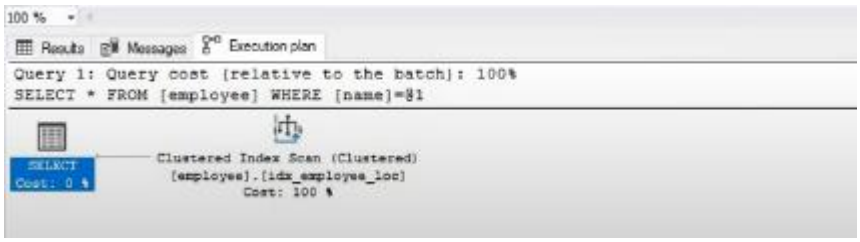
It helps in access row faster using name column

Syntax

```
CREATE NONCLUSTERED INDEX index_name
ON table_name(column_name <ASC | DESC> );
```

```
CREATE NONCLUSTERED INDEX idx_employee_name
ON employee(name ASC);
```

```
SELECT * FROM employee;

CREATE NONCLUSTERED INDEX idx_employee_name
ON employee(NAME ASC);

select * from employee where name='mike';
```



Unique Index in MS SQL Server

Unique Index

- A **unique index** ensures that the index key contains no duplicate values.
- There are no differences between creating a **UNIQUE constraint** and creating a **unique index**.
- Creating a UNIQUE constraint on the column makes the purpose of the index clear.

Implementation

1. **PRIMARY KEY**
   - When you create a **PRIMARY KEY constraint**, a **unique clustered index** on the column or columns is automatically.
2. **UNIQUE constraint**
   - When you create a **UNIQUE constraint**, a **unique nonclustered index** is created to enforce a UNIQUE constraint by default.
3. **Index independent of a constraint**
   - Multiple unique nonclustered indexes can be defined on a table.

Syntax

```
CREATE UNIQUE NONCLUSTERED INDEX index_name
ON table_name(column_name);
```

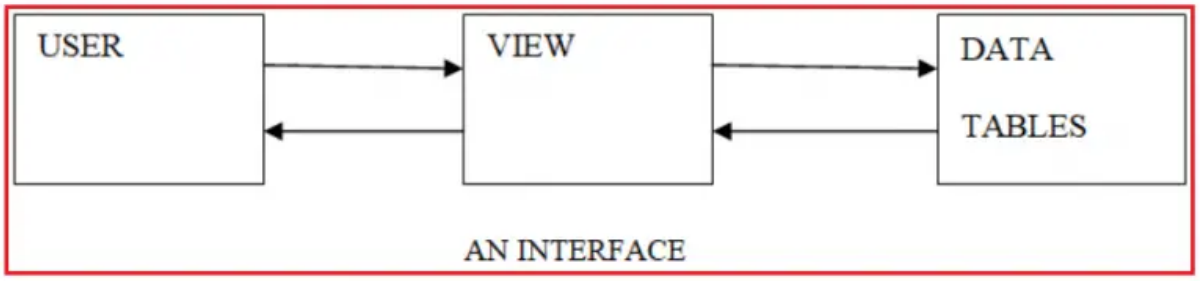
```
CREATE UNIQUE NONCLUSTERED INDEX
idx_employee_id ON employee(id);
```

View in MS SQL Server

What is a View in SQL Server?

The views in SQL Server are nothing more than a compiled SQL query. We can also consider the Views as virtual tables. As a virtual table, the Views do not store any data physically by default. So when we query a view it actually gets the data from the underlying database tables as shown in the below image.

Simply we can say that the views in SQL Server act as an interface between the Table(s) and the user.



View

- A **view** does not require any storage in a database.
- It is a **Saved Query.** It acts as a filter on the tables referenced in the view query.
- The main use case of a view to **maintain the security at row and column level**.

Syntax

```
CREATE VIEW view_name
AS [simple select query] | [ join query ];
```

```
CREATE VIEW v_EmployeeDetails
AS SELECT empName, empSal, loc FROM employee;
```

```
CREATE VIEW v_EmpDetailsWithDeptName
AS select name, salary, location, dname
from employee e inner join department d
on e.deptid = d.did where d.dname = 'sales';

SELECT * FROM v_EmpDetailsWithDeptName;
```

How many types of views are there in SQL Server?

There are two types of views in SQL Server, they are

1. **Simple Views – single table**
2. **Complex Views – multiple table using joins or other function**

In simple View, We can insert,update,delete function can be done in VIEW

Conclusion:

1. **In a Complex View, if your update statement affects one base table, then the update succeeded but it may or may not update the data correctly.**
2. **if your update statement affects more than one table, then the update failed and we will get an error message stating "View or function 'vwEmployeesByDepartment' is not updatable because the modification affects multiple base tables".**

Can we drop a table that has dependent views on it?

→Yes, you can drop a table even if any dependent views are associated with it, but the views that are associated with it will not be dropped. They will still execute in the database only with the status as inactive object and all those views become active and start functioning provided the table is recreated.

Can we create a view based on other views?

→Yes, It is possible in SQL Server to create a view based on other views.

Advantages of Views in SQL Server

We are getting the following advantages of using Views in SQL Server.

1. **Hiding the complexity**
2. **Implementing Row and Column Level Security.**
3. **Presenting the aggregated data by hiding the detailed data.**

CTE in MS SQL Server

CTE (Common Table Expression)

- **CTE** is a temporary named result set.
- A CTE must be followed by a single SELECT, INSERT, UPDATE, or DELETE statement that references some or all the CTE columns. A CTE can also be specified in a CREATE VIEW statement as part of the defining SELECT statement of the view.
- Multiple CTE query definitions can be defined.

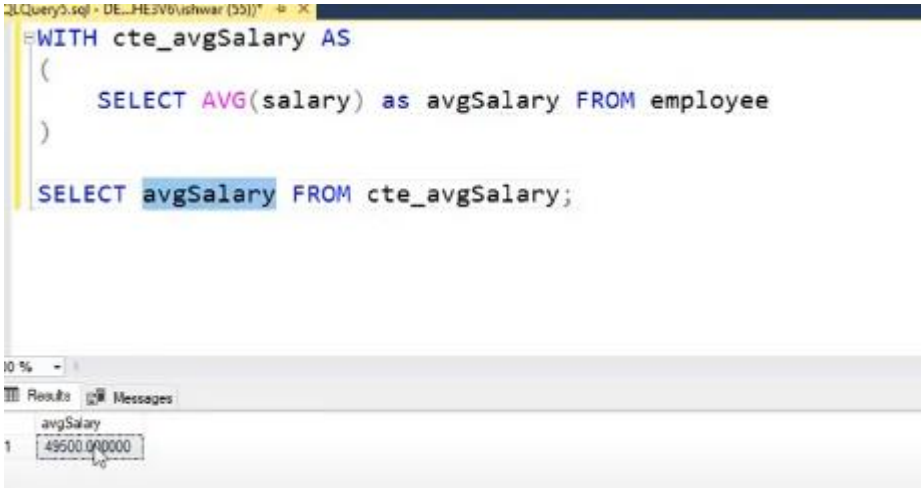
Syntax

```
WITH cte_name AS ( cte_query ) followed_query
```

```
WITH cte_name(col1, col2, ....) AS ( cte_query )
followed_query
```

```
WITH cte_name AS ( cte_query ) followed_query
```

```
WITH cte_avgSalary AS
(
    SELECT avg(salary) as AvgSalary FROM employee
)
SELECT AvgSalary FROM cte_avgSalary;
```




```
WITH cte_name(col1, col2, ....) AS ( cte_query )
followed_query
```

```
WITH cte_empCount(deptid, employeeCount) AS
(
    SELECT deptid, count(*) AS employeeCount
    FROM employee group by deptid
)
SELECT dname, employeeCount FROM
department JOIN cte_empCount
ON department.did = cte_empCount.deptid;
```

```
WITH cte_empCount(deptid, employeeCount) AS
(
    SELECT deptid, count(*) as employeeCount
    FROM employee GROUP BY deptid
)

SELECT dname, employeeCount
FROM department JOIN cte_empCount
ON department.did = cte_empCount.deptid
```

TCL (Commit, Rollback, Save)

Transaction Control Language

- If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.
- **COMMIT**
- **ROLLBACK**
- **SAVE**

To manage the transaction in SQL Server, we have provided transaction control language (TCL). TCL provides the following 4 commands which we can use to implement transactions in SQL Server.

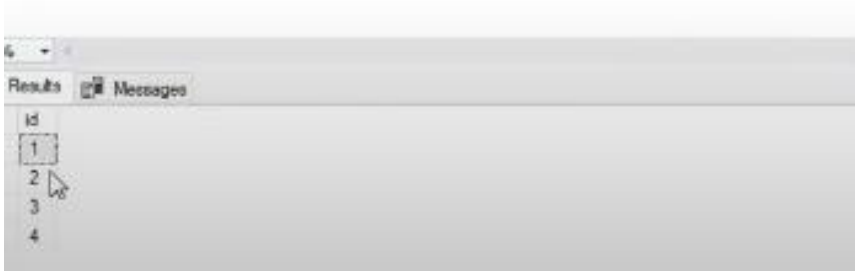
BEGIN TRANSACTION: To start the transaction
COMMIT: To save the changes.
ROLLBACK TRANSACTION: To roll back the changes.
SAVE TRANSACTION: Creates points within groups of transactions in which to ROLLBACK

1. **Begin Transaction:** It indicates that the transaction is started.
2. **Commit Transaction:** It indicates that the transaction was completed successfully and all the data manipulation operations performed since the start of the transaction are committed to the database **and frees the resources held by the transaction.**
3. **Rollback Transaction: It indicates that the transac**tion was Failed and will roll back the data to its previous state.
4. **Save Transaction:** This is used **for dividing or breaking a transaction into multiple units so that the user has a chance of roll backing a transaction up to a point or location.**

Syntax

```
BEGIN TRANSACTION;
    SQL statements
COMMIT | ROLLBACK | SAVE
```

```
BEGIN TRANSACTION;
    INSERT INTO sampleTable values(3);
    INSERT INTO sampleTable values(4);
SAVE TRANSACTION A;

    INSERT INTO sampleTable values(5);
    INSERT INTO sampleTable values(6);
SAVE TRANSACTION B;

ROLLBACK TRANSACTION A;
SELECT * FROM sampleTable;
```

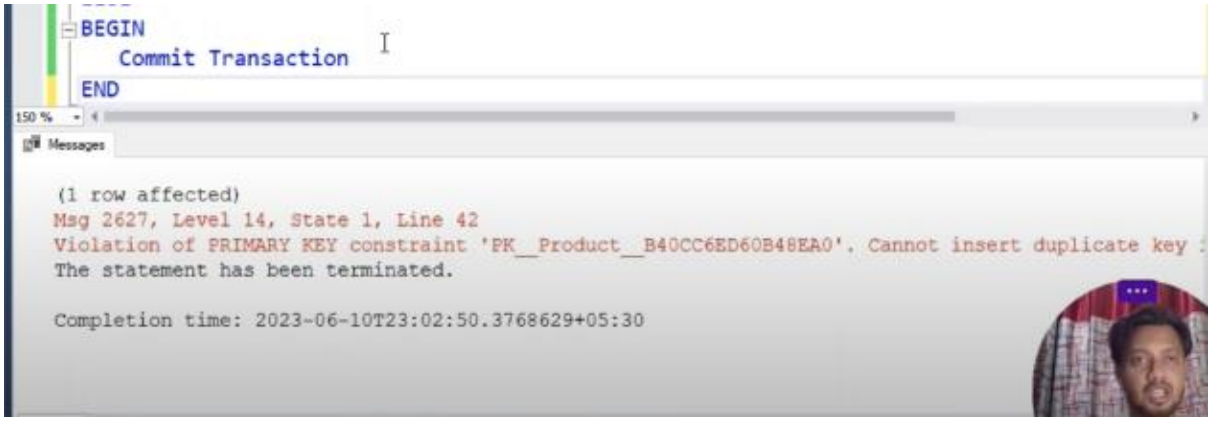


Understanding @@Error Global variable in SQL Server:

This is a global variable and we can use this variable to check if there is any error or not. Let us see an example to understand this. As you can see in the below example, first we start the transaction using the Begin Transaction statement. Then we write two insert statements. Then we check **if there is an error using the global system variable @@ERROR. A value greater than 0 means, there is some error. If there is some error then we roll back the transaction else we commit the transaction.**

Once you execute the above transaction, then you will see that the transaction is rollback. This is because we try to insert a duplicate value into the Primary key column.

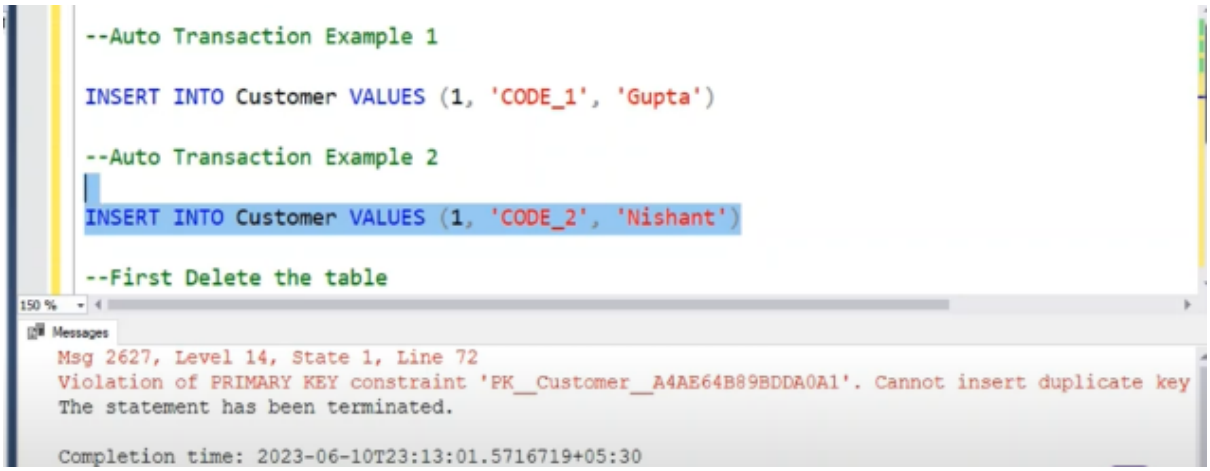
```
BEGIN TRANSACTION
INSERT INTO Product VALUES(110,'Product-10',600, 30)
INSERT INTO Product VALUES(110,'Product-10',600, 30)

IF(@@ERROR > 0)
BEGIN
    Rollback Transaction
END
ELSE
BEGIN
```



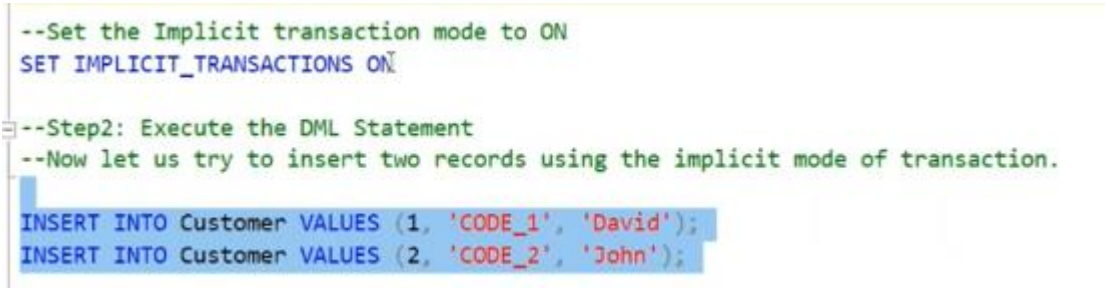
Types of Transactions in SQL Server:

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

1. **Auto Commit Transaction Mode (default)**



2. **Implicit Transaction Mode – Need specific command to like commit or rollback**



COMMIT TRANSACTION or ROLLBACK

3. **Explicit Transaction Mode – Similar to normal TCL using begin and end**

```
BEGIN TRANSACTION
INSERT INTO Product VALUES(110,'Product-10',600, 30)
INSERT INTO Product VALUES(110,'Product-10',600, 30)

IF(@@ERROR > 0)
BEGIN
    Rollback Transaction
END
ELSE
BEGIN
```

Backup and Restore Database in MS SQL Server.

Using Microsoft SQL Server 2019

ACID Properties in SQL Server

In the context of transaction processing, the acronym ACID refers to the four key properties of a transaction, such as

1. **Atomicity**

   ensures that either all the DML Statements (i.e. insert, update, delete) inside a transaction are completed successfully or all of them are rolled back.

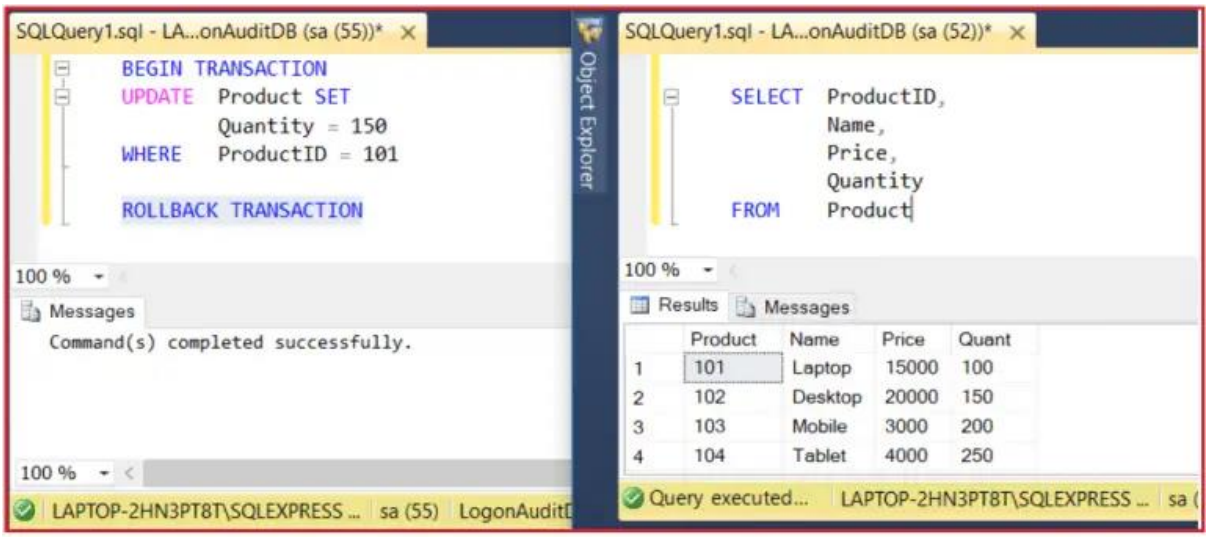
2. **Consistency**

   ensures that the database data is in a consistent state before the transaction started and also left the data in a consistent state after the transaction is completed. If the transaction violates the rules then it should be rolled back. For example, if stocks available are decremented from the Product table then there has to be a related entry in the ProductSales table

   ANOTHER EXAMPLE:BANK TRANSACTION detected from one account and added on other account

3. **Isolation**

   ensures that the intermediate state of a transaction is invisible to other transactions. The Data modifications made by one transaction must be isolated from the data modifications made by all other transactions. Most databases use locking to maintain transaction isolation.



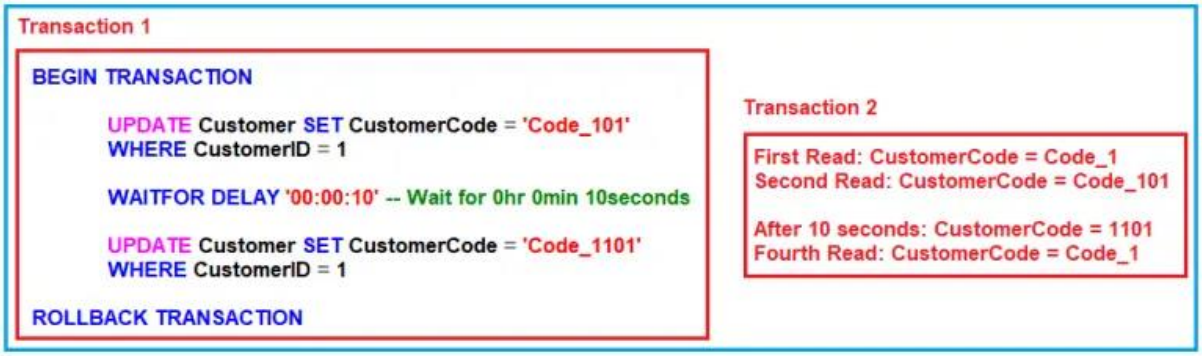
4. **Durability.**

   ensures that once the transaction is successfully completed, then the changes it made to the database will be permanent. Even if there is a system failure or power failure or any abnormal changes, it should safeguard the committed data.

Concurrency in SQL Server

What is Concurrency in SQL Server?

When we talk about transactions, one more thing which we need to handle is concurrency. **Concurrency is nothing but is a situation where two users are trying to access the same information and while they are accessing the same information we do not want any kind of inconsistency result or abnormal behavior.**



Different Types of Concurrency Problems in SQL Server:

1. Dirty Reads
2. Lost Updates
3. Non-repeatable Reads
4. Phantom Reads

We will discuss what all these concurrency problems are in SQL Server and when these concurrency problems have occurred in detail with real-time examples from our next article.

How to Overcome the Concurrency Problems in SQL Server?

To overcome the above Concurrency Problems, SQL Server **provides different types of Transaction Isolation Levels,** to balance the concurrency problems and performance depending on our application's need. The Transaction Isolation Levels provided by SQL Server are as follows

1. Read Uncommitted
2. Read Committed
3. Repeatable Read
4. Snapshot
5. Serializable

Depending upon the Transaction Isolation Level you choose for your transaction, you will get varying degrees of performance and concurrency problems. The following table shows the list of isolation levels along with the concurrency problems.

Isolation Level	Dirty Reads	Lost Update	Non repeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

Dirty Read Concurrency Problem in SQL Server

What is Dirty Read Concurrency Problem in SQL Server?

The Dirty Read Concurrency Problem in SQL Server happens w**hen one transaction is allowed to read the uncommitted data of another transaction.** That is the data has been modified by another transaction but not yet committed or rollback. In most of the scenarios, it would not cause any problem. However, **if the first transaction is rolled back after the second transaction reads the uncommitted data**, **then the second transaction has dirty data that does not exist anymore in the database.**

Example :

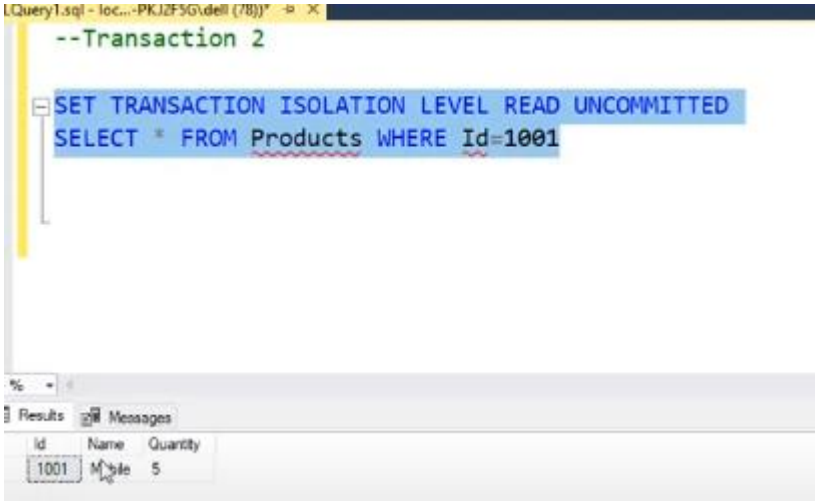
create and populate the Products table with the required sample data.

Id	Name	Quantity
1001	Mobile	10
1002	Tablet	20
1003	Laptop	30

Transaction 1:

```
BEGIN TRANSACTION
  UPDATE Products SET Quantity = 5 WHERE Id=1001

  -- Billing the customer
  Waitfor Delay '00:00:15'
  -- Insufficient Funds. Rollback transaction

ROLLBACK TRANSACTION
```

Transaction 2:



How to Overcome the Dirty Read Concurrency Problem Example in SQL Server?

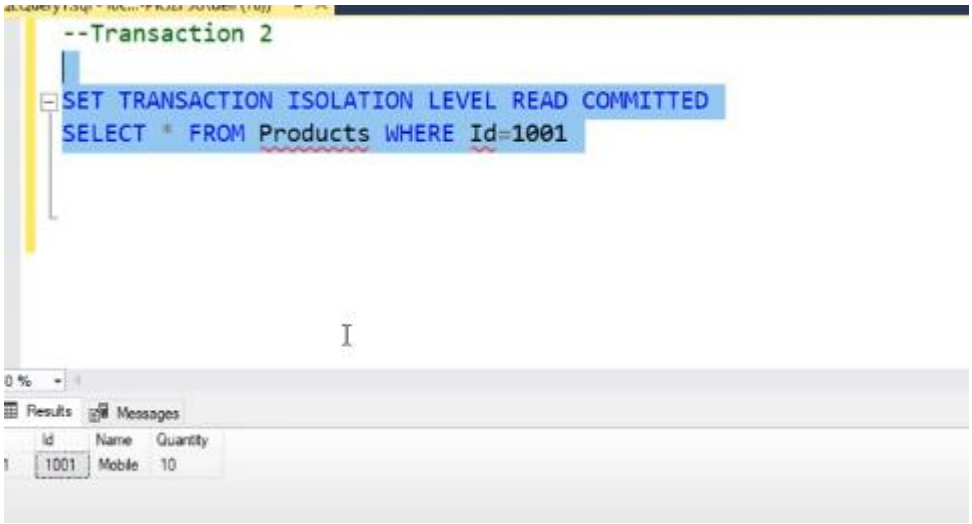
If you want to restrict the dirty read concurrency problem in SQL Server, then you have to use any Transaction Isolation Level except the Read Uncommitted Transaction Isolation Level. So,

modify transaction 2 as shown below and hear we are using Read Committed Transaction Isolation Level

Transaction 1:

```
BEGIN TRANSACTION
  UPDATE Products SET Quantity = 5 WHERE Id=1001

  -- Billing the customer
  Waitfor Delay '00:00:15'
  -- Insufficient Funds. Rollback transaction

ROLLBACK TRANSACTION
```

Transaction 2:



NOLOCK table hint in SQL Server:

Another option provided by SQL Server to read the dirty data is by using the NOLOCK table hint option. The below query is equivalent to the query that we wrote in Transaction 2.

```
SELECT * FROM Products (NOLOCK) WHERE Id=1001
```

Lost Update Concurrency Problem in SQL Server

What is the Default Transaction Isolation Level in SQL Server?

The **default Transaction Isolation level in SQL Server is Read committed.** That means, if we do not specify any Transaction Isolation Level in SQL Server then by default it is Read Committed Transaction Isolation Level. **With the Read Committed Transaction Isolation Level, we will get all sorts of Concurrency Problems (Lost Update, Non-Reaptable Read, and Phantom Read) except the Dirty Read Concurrency Problem.**

What is Lost Update Concurrency Problem in SQL Server?

The Lost Update Concurrency Problem happens in SQL Server **when two or more transactions are allowed to read and update the same data.**

Example:

Id	Name	Quantity
1001	Mobile	10
1002	Tablet	20
1003	Laptop	30

Transaction 1:

```
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  -- Transaction takes 10 seconds
  WAITFOR DELAY '00:00:10'

  SET @QunatityAvailable = @QunatityAvailable - 1
  UPDATE Products SET Quantity = @QunatityAvailable  WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

Transaction 2:

```
-- Transaction 2
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  SET @QunatityAvailable = @QunatityAvailable - 2
  UPDATE Products SET Quantity = @QunatityAvailable WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

Table:

```
select * from products
```

50 %

Results | Messages

	Id	Name	Quantity
1	1001	Mobile	9
2	1002	Tablet	20
3	1003	Laptop	30

How to Overcome the Lost Update Concurrency Problem?

Both **Read Uncommitted and Read Committed Transaction Isolation Levels have the Lost Update Concurrency Problem**. The other Isolation Levels such as **Repeatable Read, Snapshot, and Serializable do not have the Lost Update Concurrency Problem**. So, if we run the above Transactions using any of the higher Transaction Isolation Levels such as Repeatable Read, Snapshot, or Serializable, then we will not get the lost update concurrency problem.

Transaction 1:

```
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001

  -- Transaction takes 10 seconds
  WAITFOR DELAY '00:00:10'

  SET @QunatityAvailable = @QunatityAvailable - 1
  UPDATE Products SET Quantity = @QunatityAvailable  WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

Transaction 2:

```
---- Transaction 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  DECLARE @QunatityAvailable int
  SELECT @QunatityAvailable = Quantity FROM Products WHERE Id=1001
```

```
  SET @QunatityAvailable = @QunatityAvailable - 2
  UPDATE Products SET Quantity = @QunatityAvailable WHERE Id=1001
  Print @QunatityAvailable
COMMIT TRANSACTION
```

Table:

	Id	Name	Quantity
1	1001	Mobile	7
2	1002	Tablet	20
3	1003	Laptop	30

Non-Repeatable Read Concurrency

What is a Non-Repeatable Read Concurrency Problem in SQL Server?

The Non-Repeatable Read Concurrency Problem happens in SQL Server **when one transaction reads the same data twice while another transaction updates that data in between the first and second read of the first transaction.**

Understanding Non-Repeatable Read Concurrency Problem in SQL Server

We are going to use the following Products table to understand this concept.

Id	Name	Quantity
1001	Mobile	10
1002	Tablet	20
1003	Laptop	30

Example:

Transaction 1

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
BEGIN TRANSACTION
SELECT Quantity FROM Products WHERE Id = 1001
-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT Quantity FROM Products WHERE Id = 1001
COMMIT TRANSACTION
```

136 %

Results | Messages

	Quantity
1	10

	Quantity
1	5

Transaction 2

```
-- Transaction 2
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
UPDATE Products SET Quantity = 5 WHERE Id = 1001
```

How to Solve the Non-Repeatable Read Concurrency Problem in SQL Server?

In order to solve the Non-Repeatable Read Problem in SQL Server, **we need to use either Repeatable Read Transaction Isolation Level or any other higher isolation level such as Snapshot or Serializable**. So, let us s**et the transaction isolation level of both Transactions to repeatable read** (you can also use any higher transaction isolation level). **This will ensure that the data that Transaction 1 has read will be prevented from being updated or deleted elsewhere. This solves the non-repeatable read concurrency issue.** Let us rewrite both the transactions using the Repeatable Read Transaction Isolation Level.

Modify the Transactions

Transaction 1

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT Quantity FROM Products WHERE Id = 1001
-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT Quantity FROM Products WHERE Id = 1001
COMMIT TRANSACTION
```

136 %

Results | Messages

	Quantity
1	5

	Quantity
1	5

Transaction 2

```
-- Transaction 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
UPDATE Products SET Quantity = 5 WHERE Id = 1001
```

Phantom Read Concurrency

What is Phantom Read Concurrency Problem in SQL Server?

The Phantom Read Concurrency Problem happens in SQL Server **when one transaction executes a query twice and it gets a different number of rows in the result set each time.** This generally happens **when a second transaction inserts some new rows in between the first and second query execution of the first transaction that matches the WHERE clause of the query executed by the first transaction.**

Understanding Phantom Read Concurrency Problem in SQL Server

Let's understand Phantom Read Concurrency Problem in SQL Server with an example. We are going to use the following Employees table to understand this concept.

Phantom Read Concurrency Problem in SQL Server with an Example:

Transaction 1

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
SELECT * FROM Employees where Gender = 'Male'
-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT * FROM Employees where Gender = 'Male'
COMMIT TRANSACTION
```

136 %

Results | Messages

	Id	Name	Gender
1	1001	Anurag	Male
2	1003	Pranaya	Male

	Id	Name	Gender
1	1001	Anurag	Male
2	1003	Pranaya	Male
3	1006	Sambit	Male

Transaction 2

```
-- Transaction 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
INSERT into Employees VALUES(1006, 'Sambit', 'Male')
COMMIT TRANSACTION
```

How to solve the Phantom Read Concurrency Problem in SQL Server?

You can use the Serializable or Snapshot Transaction Isolation Level to solve the Phantom Read Concurrency Problem in SQL Server.

Transaction 1

```
-- Transaction 1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
SELECT * FROM Employees where Gender = 'Male'
-- Do Some work
WAITFOR DELAY '00:00:15'
SELECT * FROM Employees where Gender = 'Male'
COMMIT TRANSACTION
```

136 %

Results | Messages

	Id	Name	Gender
1	1001	Anurag	Male
2	1003	Pranaya	Male

	Id	Name	Gender
1	1001	Anurag	Male
2	1003	Pranaya	Male

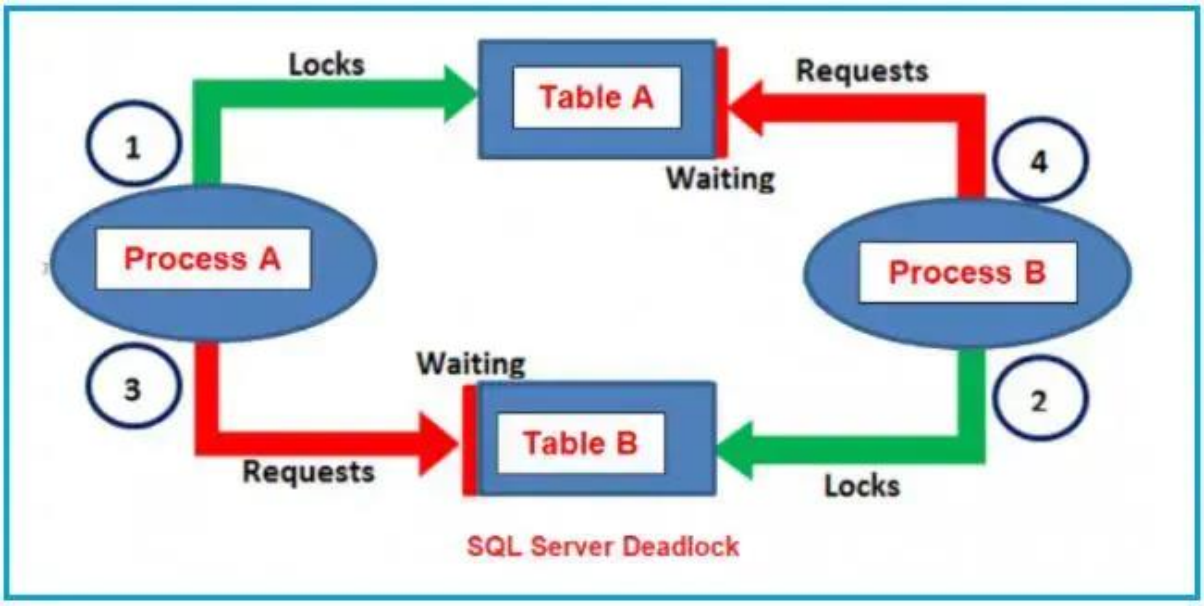
Transaction 2

```
-- Transaction 2
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
INSERT into Employees VALUES(1007, 'Sambit', 'Male')
COMMIT TRANSACTION
```

Deadlock in SQL Server

When a deadlock occurs in SQL Server?

A deadlock occurs in a database **when two or more processes have already a resource locked, and then each process wants to acquire a lock on the resource that the other process has already locked**. In such cases, **neither of the processes can move forward, as each process is waiting for the other process to release the lock resulting in a deadlock in SQL Server**. If you are confused, then just have a look at the following diagram which explains the above points i.e. when a deadlock occurs in a database.



Understanding Deadlock in SQL Server with Examples.

TableA

ID	Name
101	Anurag
102	Mohanty
103	Pranaya
104	Rout
105	Sambit

TableB

ID	Name
1001	Priyanka
1002	Dewagan
1003	Preety

Tranaction 1;

```
---- Transaction 1
BEGIN TRANSACTION
UPDATE TableA SET Name = 'Anurag From Transaction1' WHERE Id = 101

WAITFOR DELAY '00:00:15'

UPDATE TableB SET Name = 'Priyanka From Transaction1' WHERE Id = 1001
COMMIT TRANSACTION
```

Tranaction 2:

```
-- Transaction 2
BEGIN TRANSACTION
UPDATE TableB SET Name = 'Priyanka From Transaction2' WHERE Id = 1001

WAITFOR DELAY '00:00:15'

UPDATE TableA SET Name = 'Anurag From Transaction2' WHERE Id = 101
Commit Transaction
```

Now, **first, execute the Transaction 1 code and then immediately execute the Transaction 2 code.** Once **Transaction 1 starts its execution, it acquires a lock on TableA and then waits for 15 seconds, At the same time, Transaction 2 starts its execution, it acquires a lock on TableB and then waits for 15 seconds. After 15 seconds, Transaction 1 wants to acquire a lock on TableB which is already acquired by Transaction 2 and at the same time, Transaction 2 wants to acquire a lock on TableA which is already acquired by Transaction 1. I**n this situation, **neither of the transactions can move forward.** So, after some time, You will notice that **one of the transactions was completed successfully while the other transaction is chosen as the deadlock victim by giving the following error.**

```
Msg 1205, Level 13, State 45, Line 7
Transaction (Process ID 53) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.
```

How Deadlocks are detected by SQL Server?

The **Lock Monitor thread** **of SQL Server by default runs in every 5 seconds to detect if there are any deadlocks occurred in the database.** If the Lock Monitor thread finds any deadlocks in the database, then the deadlock **detection interval will be a drop from 5 seconds to as low as 100 milliseconds depending on the frequency of the deadlocks**. If the Lock Monitor thread stops finding deadlocks, then the Database Engine increases the intervals to 5 seconds.

What is DEADLOCK_PRIORITY in SQL Server?

As we already discussed when a deadlock occurs, then by default, SQL Server chooses one of the transactions as the deadlock victim and **it will choose the transaction that is least expensive to roll back by default**. However, **as a user, we can also specify the priority of the transactions in a deadlock situation using the SET DEADLOCK_PRIORITY statement**. Once we set the deadlock priority, then the **transaction with the lowest deadlock priority will be chosen as the deadlock victim**.

Example: SET DEADLOCK_PRIORITY NORMAL

DEADLOCK_PRIORITIES in SQL Server:

1. The default priority is Normal
2. It can be set to LOW, NORMAL, or HIGH
3. It can also be set to an integer value in the range of -10 to 10. (LOW: -10, NORMAL: 0, and HIGH: 10

What are the Deadlock Victim Selection Criteria in SQL Server?

The SQL Server uses the following criteria to select the deadlock victim transaction

1. If the DEADLOCK_PRIORITY is different, then the **transaction with the lowest priority will be selected as the deadlock victim.**
2. When both the transaction **having the same priority, then the transaction that is least expensive to rollback is selected** as the deadlock victim transaction.
3. If both the transactions **having the same deadlock priority as well as the same cost, then SQL Server chooses one of the transactions as victim randomly.**

Deadlock Logging in SQL Server

How to Find Deadlock Queries in SQL Server?

There are **many ways** available in SQL Server **to track down the queries which are causing the deadlocks**. One of the options that is available in SQL Server is **to use the SQL Server Trace Flag 1222 to log the deadlock information** to the SQL Server Error Log. Let discuss how to enable the Trace Flag in SQL Server.

Enable Trace Flag in SQL Server:

To enable the trace flag in SQL Server we **need to use the DBCC command**. The **-1 parameter indicates that the trace flag must be set at the global level**.

Example: Trace Flag in SQL Server

Let us understand how to enable and use Trace Flag in SQL Server with an example. We are going to use the following two tables to understand this concept. **Create and populate the tables with the test data.**

ID	Name
101	Anurag
102	Mohanty

Table A

ID	Name
1001	Priyanka
1002	Dewagan

Table B

Now create two stored procedure

First, enable the Trace Flag by executing the following command

```
DBCC Traceon(1222, -1)
```

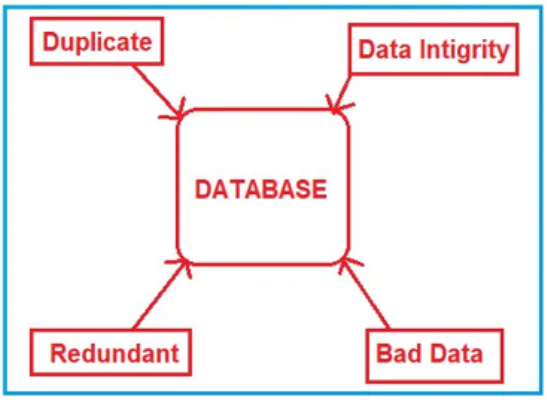
Then open 2 instances of SQL Server Management Studio. **From the first instance execute the spTransaction1 stored procedure and from the second instance execute the spTransaction2 stored procedure.** After a few seconds, you will notice that one of the transactions completes its execution successfully while the other one is chosen as the deadlock victim and rollback.

The information about this deadlock now should have been logged in SQL Server Error Log. **To read the error log you need to use the sp_readerrorlog system stored procedure** as shown below.

```
EXECUTE sp_readerrorlog
```

To prevent the deadlock that we have in our example, we need to ensure that the database objects such as Table A & Table B are accessed in the same order every time.

Normalization:



The **database is a centralized place to store the data and it should not accept duplicate, bad, and redundant data to store in it**. So, that the **end-user can trust the data i.e. the Data Integrity should be there**. In order to achieve this, the first and most important thing is how you design your database? What kind of database design principles you are following?

DESIGN MISTAKE 1

Before going to understand the first, second, and third normal forms, let us first understand what are the basic design mistakes database developers normally does. Once you understand the design mistakes, then you can easily understand the first, second, and third normal forms.

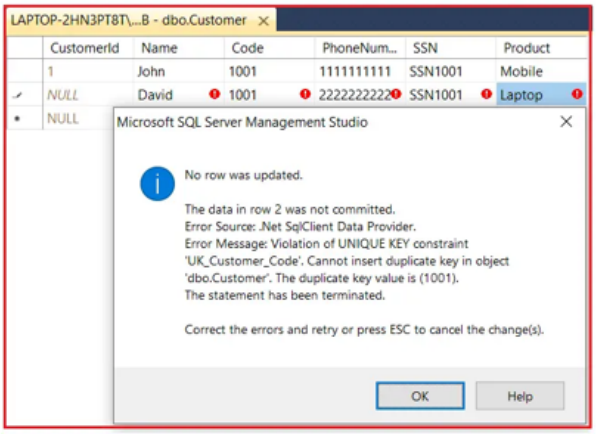
Design Mistake 1: No Proper Primary and Unique Key

The first database design mistake the developers are doing is **they don't put the proper primary key and unique keys or candidate keys.** In order to understand this, please have a look at the following customer table.

LAPTOP-2HN3PT8T\...B - dbo.Customer

	CustomerId	Name	Code	PhoneNum...	SSN	Product
	1	John	1001	1111111111	SSN1001	Mobile
	2	David	1001	2222222222	SSN1001	Laptop
▸*	NULL	NULL	NULL	NULL	NULL	NULL

Now, let us **solve the above problem by adding a unique key on Code and SSN column. But before that, we need to truncate the table.** So, please execute the following SQL Script.
Now, with the above design changes, it will not accept duplicate values in Code and SSN column and when you try to insert duplicate data, it will give you the following error by saying that you are violating the uniqueness of customer code.



DESIGN MISTAKE 2

Design Mistake 2: Multiple values into a single column

The second design mistake people do is, **they are adding multiple values into a single column**. For example, l**et say a customer buys multiple products, then what people are doing is, they are adding all products in a single column separated them either by comma or pipe symbol** as shown below.

CustomerId	Name	Code	PhoneNum...	SSN	Product
1	John	1001	1111111111	SSN1001	Mobile,Laptop,Tablet
4	David	1002	2222222222	SSN1002	Shirt|Jeans|Shooes

As you can see in the Product column we are adding three values. We need to understand that, it's a column, not a whole. **It should have an atomic value.** The problem is that you can put duplicate data. In other to solve the above problem, what people are doing is, they are adding three columns to the table as shown below.

	Column Name	Data Type	Allow Nulls
🔑	CustomerId	int	☐
	Name	varchar(200)	☑
	Code	varchar(200)	☑
	PhoneNumber	varchar(200)	☑
	SSN	varchar(200)	☑
	Product1	varchar(200)	☑
	Product2	varchar(50)	☑
	Product3	varchar(50)	☑

And then insert each product into the respective table as shown below.

CustomerId	Name	Code	PhoneNum...	SSN	Product1	Product2	Product3
1	John	1001	1111111111	SSN1001	Mobile	Laptop	Tablet
4	David	1002	2222222222	SSN1002	Shirt	Jeans	*NULL*

This is not a good design approach. The problem here is redundant data. In our example, Product1, Product2, and Product3 are redundant columns. In the first record, there are three products and in the second record, there are 2 products but the Product3 column also exists and it takes Null value which is nothing but redundant value. Because of our bad database design, we came into redundant problems. **This problem also technically termed repeating group problems.**

DESIGN MISTAKE 3

Design Mistake 3: Repeating Group problems

Repeating group problems means we are creating columns that are exactly the same (Product1, Product2, and Product3). Why these three columns are existing, now one knows. **Again, tomorrow, if you want to add the fourth product, then again you need to add one more column into the table.**

Product Table:

ProductId	ModelId	ProductName	ProductCost	ModelName	ManufacturerName
101	10001	Mobile	2000.0000	Samsung	Samsung
102	10002	Laptop	3000.0000	Lenovo	Lenovo

Customer Table:

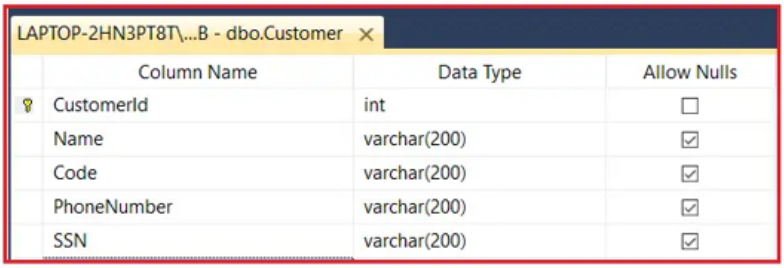
Now you can insert the ProductId and ModelId values into the ProductId and ModelId column of the Customer table as shown below.

CustomerId	Name	Code	PhoneNumber	SSN	ProductId	ModelId
1	John	1001	1111111111	SSN1001	101	10001
2	David	1002	2222222222	SSN1002	102	10002

That's fine. There is no redundancy data or column. But this design is not what we are expecting. One customer can buy multiple products. But putting the ProductId and ModelId column here means one customer can buy only one product. That is one to one relationship.

So, this design will not fit here. Here, we need to introduce an intermediate table that has reference to both Product (ProductId and ModelId) and Customer (CustomerId) table. That means we need to create a table that has many to many relationships between Customer and Product table i.e. one customer can buy multiple products and multiple products can be bought by multiple customers.

First, delete the ProductId and ModelId column from the Customer table. Once you delete those two columns, your Customer table should look as below.



Creating ProductCustomerMapping table:

Please execute the below script to create the ProductCustomerMapping table. This table maintains many to many relationships between the Customer and Product table. It has a reference to the CustomerId column of the Customer table as well as a reference to the ProductId and ModelId column of the Product table.

Customer

CustomerId	Name	Code	PhoneNumber	SSN
1	John	1001	1111111111	SSN1001
2	David	1002	2222222222	SSN1002

Product

ProductId	ModelId	ProductName	ProductCost	ModelName	ManufacturerName
101	10001	Mobile	2000.0000	Samsung	Samsung
102	10002	Laptop	3000.0000	Lenovo	Lenovo

ProductCustomerMapping

ProductCustomerId	CustomerId	ModelId	ProductId
1	1	10001	101
2	1	10002	102
3	2	10001	101

First Normal Form in Database Normalization

First Normal Form in Database Normalization in SQL Server with Examples:

1. **Design Problem 1:** Duplicate Data due to no proper primary key and unique keys.
2. **Design Problem 2:** One column containing multi-value data. It should atomic value. Atomic value means it should not be divided further.
3. **Design Problem 3:** Repeating Groups or redundant columns.

When the above design problems are solved or you can say the above rules are satisfied, that means your database is in First Normal Form.

DESIGN PROBLEM 4:

Design Problem 4: Non-Key columns are fully dependent on the primary key

The fourth design problem that lots of developers do is, **they make non-key columns (columns without primary key) in the tables which are not dependent on the primary key column.** In order to understand Here, we are adding Country and City columns to the table.

LAPTOP-2HN3PT8T\...tDB - dbo.Product

Column Name	Data Type	Allow Nulls
ProductId	int	☐
ModelId	int	☑
ProductName	nvarchar(50)	☑
ProductCost	money	☑
ModelName	nvarchar(50)	☑
ManufacturerName	nvarchar(50)	☑
Country	varchar(50)	☑
City	varchar(50)	☑

Now modify the data of the Product table as shown below.

ProductId	ModelId	ProductNa...	ProductCost	ModelName	Manufactur...	Country	City
101	10001	Mobile	2000.0000	Samsung	Samsung	India	Mumbai
102	10002	Laptop	3000.0000	Lenovo	Lenovo	USA	Lodon

If you look at the Product table, the Product Name and Product cost columns are related to the ProductId which is the primary key. The model name field is dependent on the ModelId column. And Model has nothing to do with the Product. Along the same line, the non-key column Country and Area has a relation with the ProductId column.

This is a bad design because when we say a table, a table is a logical unit. If you are putting fields that are not related to the unit then you have problems. Tomorrow if you have to add or remove fields then it becomes very difficult to change the structure. So, we should move the non-key columns which are not related to the unit to separate tables. We need to do lots of changes. Let do one by one

Step1: Creating Model table:

Please execute the below SQL Query which will create the Model table. The Model table will contain only the model-related information.

ModelId	ModelName	ManufacturerName	Country	City
10001	Mobile	Samsung	India	Mumbai
10002	Laptop	Lenovo	USA	London

Step2: Modifying the Product table.

From the Product table, delete all the columns except the ProductId, ProductName, and ProductCost column. Once you delete the columns, the Product table structure should looks as shown below.

ProductId	ProductName	ProductCost
101	Mobile	2000.0000
102	Laptop	3000.0000

Note: We need to design the database in such a way that, all the non-key columns should fully dependent on the primary key column. If you are not following this rule means one table having the data of many entities.

Step3: Modifying the ProductCustomerMapping table.

To make the thing simple, just delete and create the ProductCustomerMapping table. The only change is that the ModelId column now references the ModelId column of the Model table.

Once you modified the ProductCustomerMapping table then please fill the table with the following data.

ProductCustomerId	CustomerId	ModelId	ProductId
1	1	10001	101
2	1	10002	102
3	2	10001	101

Now, with the above database design, all the non-key columns are fully dependent on the primary key column.

Second Normal Form in Database Normalization

Second Normal Form in Database Normalization in SQL Server with Examples:

Must satisfy the first normal form:

1. **Design Problem 1:** Duplicate Data due to no proper primary key and unique keys.
2. **Design Problem 2:** One column containing multi-value data. It should atomic value. Atomic value means it should not be divided further.
3. **Design Problem 3:** Repeating Groups or redundant columns.

As well as the Design Problem 4: Non-Key columns are fully dependent on Primary key

When the above four design problems are solved or you can say when the database design satisfied the above four rules, it means your database is in Second Normal Form.

DESIGN PROBLEM 5:

Design Problem 5: Transitive Dependencies

If you look at the Model table, then the City column value depends on the Country column value, not on the ModelId column. Putting both the values in the Model column is a bad design. So, what we need to do here is, we need to move these transitive dependencies to other tables.

Step1: Creating the Country table

Please execute the below SQL Script to create the Country table with two columns. Here, we marked the CountryId as the primary key.

CountryId	CountryName
1	India
2	USA
3	UK
4	Canada

Step2: Creating the City table

Please execute the below SQL Script to create the City table with three columns. Here, we marked CityId as the primary key and CountryId as the foreign key reference to the CountryId column of the Country table.

CityId	CityName	CountryId
1	Mumbai	1
2	London	2
3	Delhi	1

Step3: Modifying the Model table structure

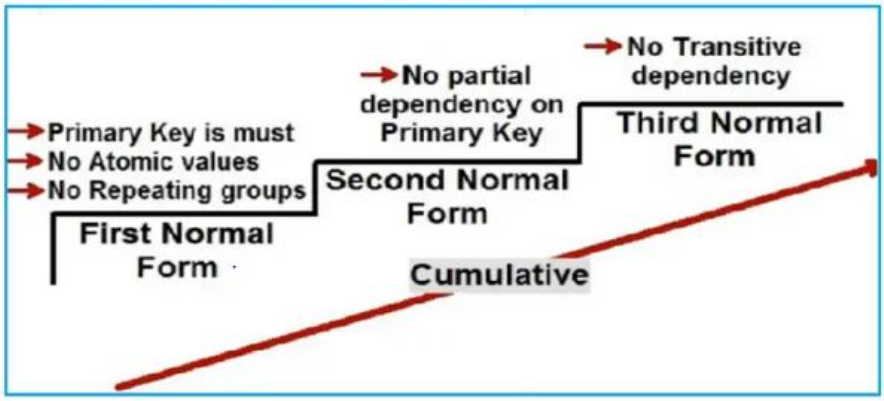
From the Model table, we need to delete the Country and City columns. Then we need to add CountryId column which will be a foreign key referencing the CountryId column of the Country table. In order to do so, please execute the below script.

ModelId	ModelName	ManufacturerName	CountryId
10001	Mobile	Samsung	1
10002	Laptop	Lenovo	2
10003	Desktop	Dell	2

Third Normal Form in Database Normalization

Third Normal Form in Database Normalization in SQL Server with Examples:

The Third Normal Form says that there should not be any transitive data and it should follow the first and second normal forms. So, Normalization is all about splitting your table into multiple tables or units. In order to remember, the first, second, and third normal form.



De-Normalization

Before understanding the other part of the story, first, we need to understand the larger classification of IT systems from the data perspective.

Classification of Data:

From the data perspective, we can classify the IT system into two classifications. They are as follows:

1. OLTP (Online Transaction Processing System)
2. OLAP (Online Analytical Processing System)

OLTP (Online Transaction Processing System):

The OLTP Systems are those systems that deal with regular transactions. In other words, the insert, update and delete kind of queries are more suitable for OLTP systems i.e. all those operations which either bring data into the system or remove data from the system. So, when the data is coming into your system, you would like to avoid redundant data, erroneous data or you like to avoid data that does not have any kind of integrity and how we can achieve by using normalization. So, for OLTP systems definitely, normalization i.e. applying 1st normal form, second normal form, and third normal form is the right way to go ahead why because we don't data to enter into the system which is redundant, error-prone, duplicate, etc

OLAP (Online Analytical Processing System)

Now, let us consider some different operations, which are not daily transaction operations. For example, the manager wants to take a large number of backdated historical data. He wants to analyze the data and he wants to do some forecasting on the data. That means he wants to do some kind of sales management on the data. In these kinds of operations, the data is not coming into the system, but there are some heavy operations done on the data such as selecting the data, creating a complicated report of data, etc. These kinds of systems are called OLAP systems.

Please have a look at the following image which summarizes the difference between OLTP and OLAP systems.

	OLTP	OLAP
Design	Normalized (Applying 1st normal form, 2nd normal form and 3rd normal form)	Denormalized (Dimension and Fact Design).
Source	Daily Transactions	OLAP
Motive	Faster Insert, Update, and Delete and improves data quality by reducing redundancy, duplicate and erroneous data.	Faster Analysis and Search by Combining tables
SQL Complexity	Simple and Medium	Highly complex due to analysis and forecasting

Why we need de-normalization in SQL Server?

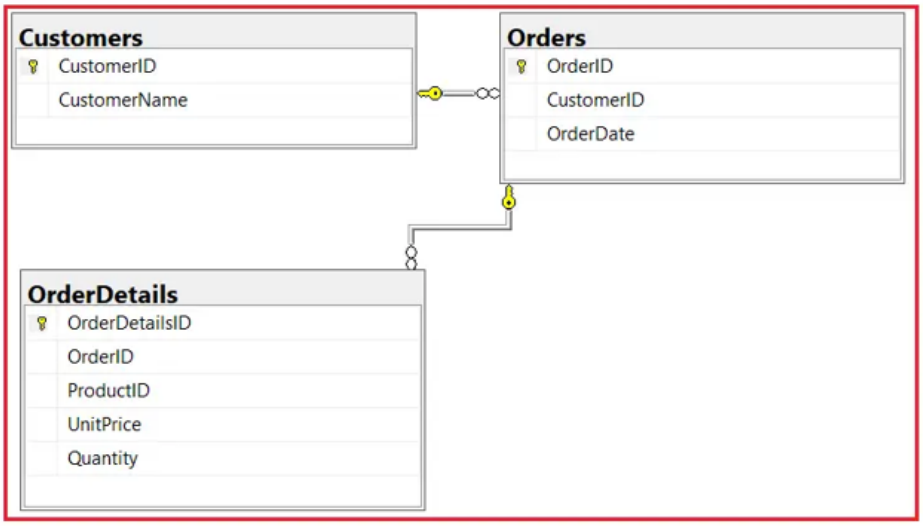
The main goal of the **OLTP system is Data Manipulation whereas the main goal of the OLAP system is search and data analysis. Normalization is not suited for the OLAP system as they bring down search performance. This is because in normalization we break down the table into multiple tables.** In the OLAP system, our main goal is a faster search. As the search is now going and pull the data from multiple tables, the search start becoming slower.

So, in the OLAP system, rather than doing normalization we need to do the opposite i.e. de-normalization.

Why normalization is not suited for the OLAP system?

Let us understand why normalization is not suited for the OLAP system with an example. Please execute the following SQL Script. **create the required database tables (Customers, Orders, and OrderDetail).**

Please have a look at the following image which shows the database diagram of the three tables we just created. This is a perfect normalized database design example.



As you can see in the above image, we have the Customers table which holds the customer name. We have **the Orders table which tells how many orders a customer has placed. We have one more table called OrderDetails which holds the detail of the order like how many products the customer bought and the quantity of each product and its unit price** etc.

Filling tables with data:

Creating Report:

Now, we need to create a very simple report as shown below. **We want to show the Customer ID, Name, the unit price, quantity, and the total amount (Unit Price * Quantity).**

CustomerID	CustomerName	UnitPrice	Quantity	TotalAmount
1	James	100	2	200
1	James	200	3	600
6	Smith	100	2	200
6	Smith	200	3	600
6	Smith	300	1	300
5	John	200	2	400
2	Pam	100	2	200
2	Pam	300	2	600
4	David	200	4	800
4	David	300	5	1500
3	Sara	100	3	300
3	Sara	200	2	400
3	Sara	300	2	600

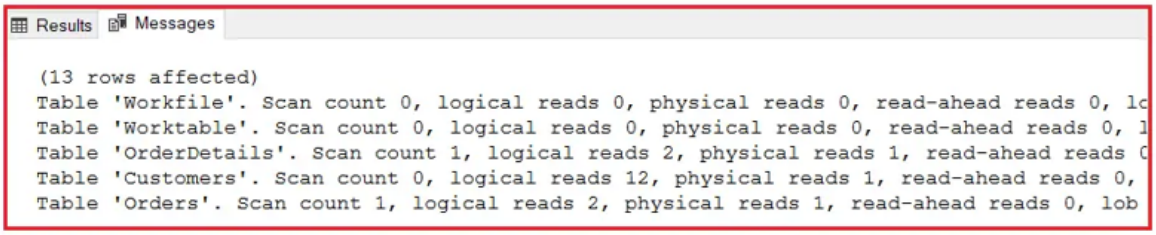
To achieve the above report, what we need to do is, we need to join the three tables as shown below.

```
SELECT C.CustomerID, C.CustomerName, OD.UnitPrice, OD.Quantity, (OD.UnitPrice * OD.Quantity) AS TotalAmount
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID
INNER JOIN OrderDetails OD ON O.OrderID = OD.OrderID
```

Ok. Let us execute the above query using the statistics as shown below.

```
SET STATISTICS IO ON

SELECT C.CustomerID, C.CustomerName, OD.UnitPrice, OD.Quantity, (OD.UnitPrice * OD.Quantity) AS TotalAmount
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID
INNER JOIN OrderDetails OD ON O.OrderID = OD.OrderID
```

Once you execute the above query, open the message window that appears next to the result window as shown below.



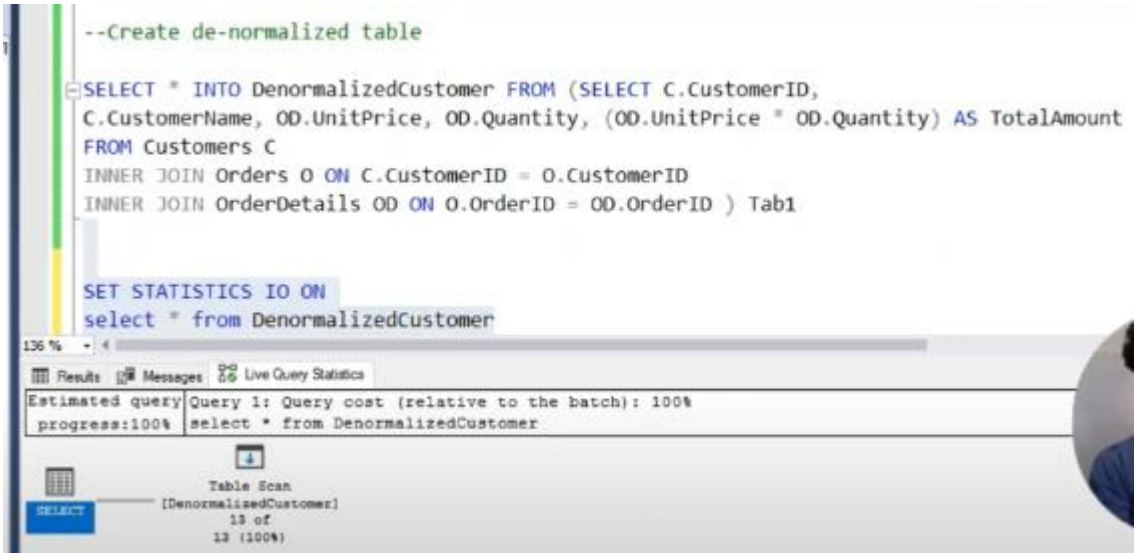
As you can see in the above image, in order to get the data, it makes inner join with the Customers, Orders, and OrderDetails tables. As well as **you can see there are lots of logical reads and physical reads are also happening.**

In OLAP systems, the search operation should be faster. Reading the data from so many tables is not an efficient way, it definitely slowdowns your select query. So, how about **rather than keeping the data in all these tables, combine them and put it into one table i.e. some de-normalized table**. So, what happens is, rather than going and make a select query into these three tables, **we can go and make the select query on that single de-normalized table which can improve the performance of the search operation.**

Creating a de-normalized table:

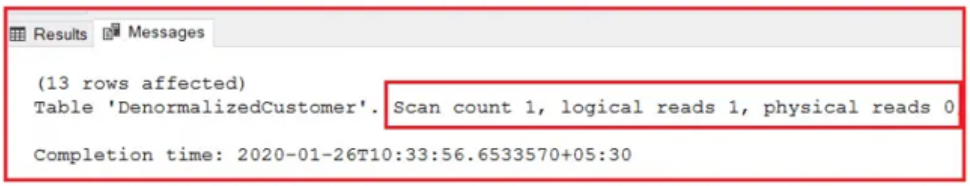
There are multiple ways to create de-normalized tables that we will discuss in our next article. Here, to make the thing simple what I am doing is, I am creating a de-normalized table by executing the following query.

The above **query will create the DenormalizedCustomer table with the required data**. Now execute the following query with statistics which will give you the same output as the previous inner join example.



```
SET STATISTICS IO ON
SELECT * FROM DenormalizedCustomer
```

Now it will get the data from a single table rather than three tables. Once you execute the query open the message window as shown below.Now, **you can see there is only one logical read which ultimately improves the search performance.**



What is De-normalization in SQL Server?

De-normalization is all about combining the data into one big table rather than going and fetching data from multiple tables. In de-normalization, **we have duplicate data, redundancy data, aggregated data, etc. i.e. we actually violated the three normal forms.**

It does not mean that when you want faster searches you start applying de-normalization i.e. if you have three tables, let's combine them and make it one table. That is not the way. **For de-normalization, there are two great techniques (Star Schema and Snow Flake) which we can apply and makes the OLAP system much better.**

Pivot and Unpivot Table in SQL Server

What is the Pivot Operator in SQL Server?

We are going to use the following **Customers table**. As you can see, the following Customers table **having three columns** (CustomerName, ProductName, and Amount). The following table tells that which customer brought which products. Some customers bought both laptops and Desktop while some customers bought either laptop or desktop. Again a few customers are there, they bought a product multiple times.

CustomerName	ProductName	Amount
James	Laptop	30000
James	Desktop	25000
David	Laptop	25000
Smith	Desktop	30000
Pam	Laptop	45000
Pam	Laptop	30000
John	Desktop	30000
John	Desktop	30000
John	Laptop	30000

create and populate the Customers table with the required data.

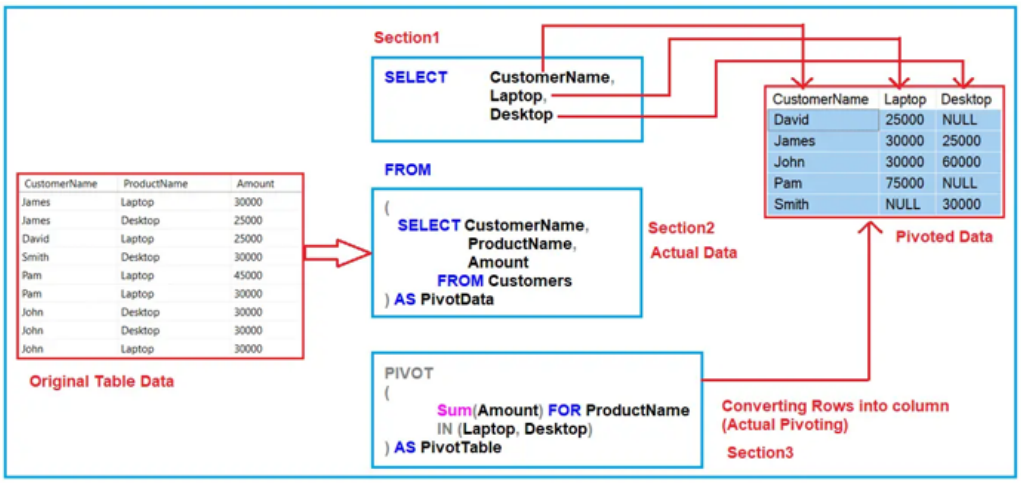
Let us visualize the above customer's data from a different angle. For example, we want to tell how many customers bought laptops and how many customers bought Desktop as shown in the below image. Here, **actually, we have to change the perspective of row-wise data into column-wise.**

CustomerName	Laptop	Desktop
David	25000	NULL
James	30000	25000
John	30000	60000
Pam	75000	NULL
Smith	NULL	30000

So, basically, **we want to convert the row-wise data into column-wise.** Now, **the question is how we can implement this in SQL Server?** The SQL Server provides one **built-in function called Pivot** which we **can use to change the row-wise data into column-wise.**

How to implement PIVOT in SQL Server?

In order to understand Pivot, please have a look at the following image. As you can see in the below image, we have divided the Pivot code into three sections.



The following is the syntax of the Pivot Operator.

```
SELECT <non-pivoted column>,
   [first pivoted column] AS <column name>,
   [second pivoted column] AS <column name>,
   ...
   [last pivoted column] AS <column name>
FROM


   (<SELECT query that produces the data>)
   AS <alias for the source query>


PIVOT
(
   <aggregation function>(<column being aggregated>)
FOR
   [<column that contains the values that will become column headers>]
   IN ( [first pivoted column], [second pivoted column], ... [last pivoted column])
)
AS <alias for the pivot table>
<optional ORDER BY clause>;
```

UNPIVOT in SQL Server:

The UNPIVOT operator performs exactly the opposite operation to PIVOT. That is, the UNPIVOT operator turns COLUMNS into ROWS. Let us understand this with an example. We are going to use the following ProductSales table to understand this concept.

AgentName	India	US	UK
Smith	9160	5220	3360
David	9770	5440	8800
James	9870	5480	8900

create and populate the ProductSales table with the required data

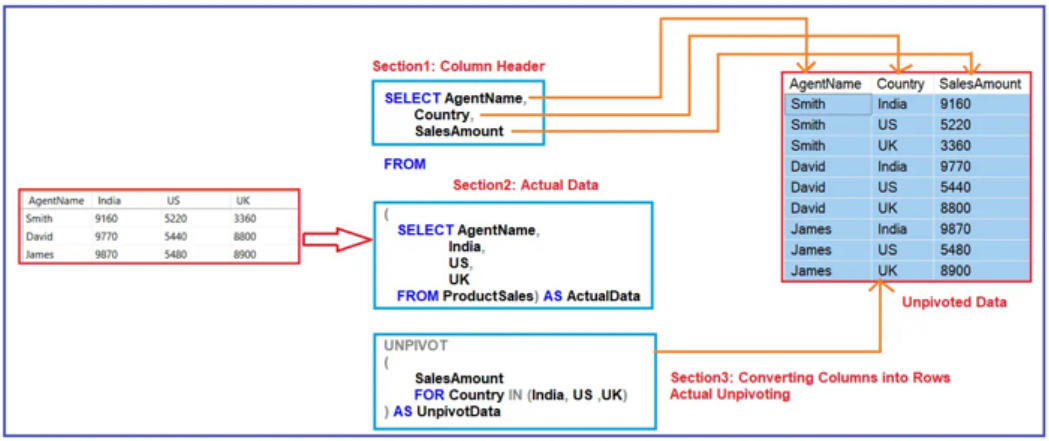
Let us visualize the above Product Sales data from a different perspective. For example, we want to tell the sales amount, per count, per agent as shown below. Here, actually, we have to change the perspective of column-wise data into row-wise

AgentName	Country	SalesAmou
Smith	India	9160
Smith	US	5220
Smith	UK	3360
David	India	9770
David	US	5440
David	UK	8800
James	India	9870
James	US	5480

Here, we need to convert the column-wise data into row-wise. Now, the question is how we can do this in SQL Server? The SQL Server provides another built-in function called UNPIVOT which we can use to change the column-wise data into row-wise.

How to implement UNPIVOT in SQL Server?

Let us understand how to use UNPIVOT operator in SQL Server. In order to understand how to use UNPIVOT; please have a look at the following diagram. As you can see in the below image, like PIVOT, here we also have divided the UNPIVOT code into three sections.



Reverse Pivot Table

Here, first, we will discuss whether it is always possible to reverse what the PIVOT operator has done using the UNPIVOT operator and how we can reverse a PIVOT table in SQL Server.

Is it always possible to reverse what the PIVOT operator has done using UNPIVOT operator in SQL Server?

The answer is No. Always it is not possible to reverse what the PIVOT operator does using the UNPIVOT operator. This basically depends on the aggregation of data. There might be two scenarios, they are as follows:

1. If the PIVOT operator aggregated the data, then you will not get the original data back using the UNPIVOT operator.
2. If the PiVOT operator has not aggregated the data, then you will get the original data back using the UNPIVOT operator.

Example:

We are going to use the following ProductSales table to understand this concept. Please have a look at the moment we don't have any SalesAgent with two records for the same country.

SalesAgenName	SalesCountryName	SalesAmount
James	India	9260
James	US	5280
Pam	India	9770
Pam	US	2540
David	India	9970
David	US	5405

create and populate ProductSales the table with the required data.

Convert Row-wise data into column-wise using the PIVOT operator

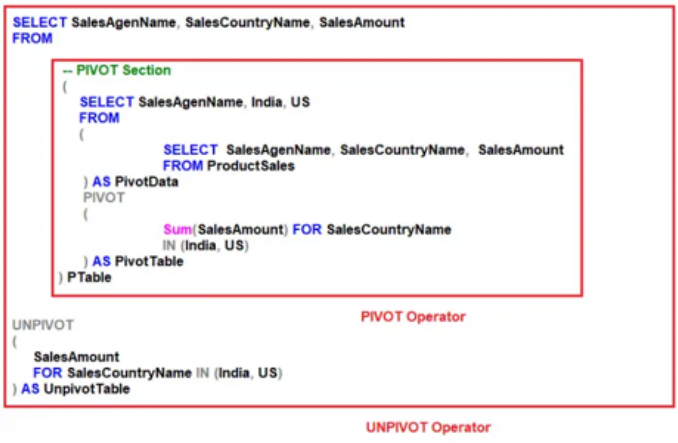
The following SQL Query will convert the row-wise data into column-wise.

Once you execute the above query, you should get the following output.

SalesAgenName	India	US
David	9970	5405
James	9260	5280
Pam	9770	2540

How to use the UNPIVOT operator to reverse what the PIVOT operator has done?

To understand this, please have a look at the following image. We already discussed how to use the PIVOT and UNPIVOT operators in our previous article. As you can see in the below image, we use the UNPIVOT operator on the result set which is return by the PIVOT operator to reverse the data.



Let us execute the following script and see the output.

Once you execute the above query, you will get the following output and you can compare this output with the original ProductSales table data. Here, we get the original data back. This is because the SUM aggregate function that we used with the PIVOT operator does not aggregate the data as there is no SalesAgent with multiple records for the same country. So, the output that we get is not aggregated.

SalesAgenName	SalesCountryName	SalesAmount
David	India	9970
David	US	5405
James	India	9260
James	US	5280
Pam	India	9770
Pam	US	2540

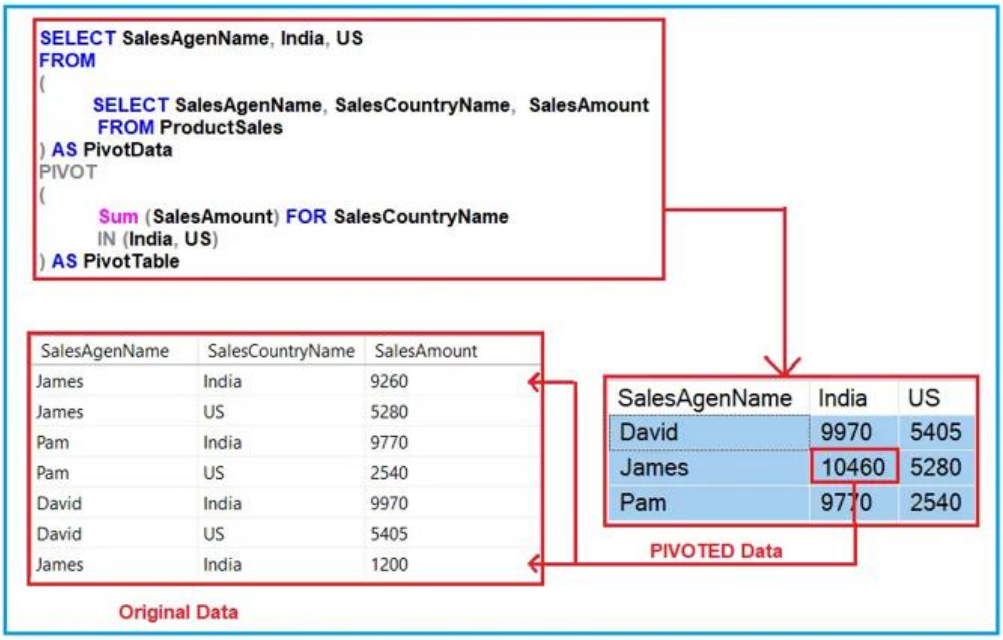
An agent with Multiple records for the same country:

Please execute the following INSERT statement to insert a new record into the ProductSales table.

```
INSERT INTO ProductSales VALUES ('James', 'India', 1200)
```

Once you execute the above INSERT statement, now the SalesAgent James has two records in the ProductSales table. Now execute the following SQL Script and see the output.

Once you execute the above query, you will get the following output. We have two records for Agent James and for India country. Please have a look at the second row (James) and India column value. Here, you can see, it sums the SalesAmount value for India country and displays it on the India column. So, the data is now aggregated.



Now if we use the UNPIVOT operator with the above query, then we wouldn't get the original data back. This is because the PIVOT operator already aggregated the data, and there is no way for SQL Server to know how to undo the aggregations.

Please execute the below SQL query.

Once you execute the above query, you will get the following output. As you can see, for the SalesAgent James and Country India, we get only one row. But in the original ProductSales table, we had 2 rows for the same combination.

SalesAgenName	SalesCountryName	SalesAmount
David	India	9970
David	US	5405
James	India	10460
James	US	5280
Pam	India	9770
Pam	US	2540

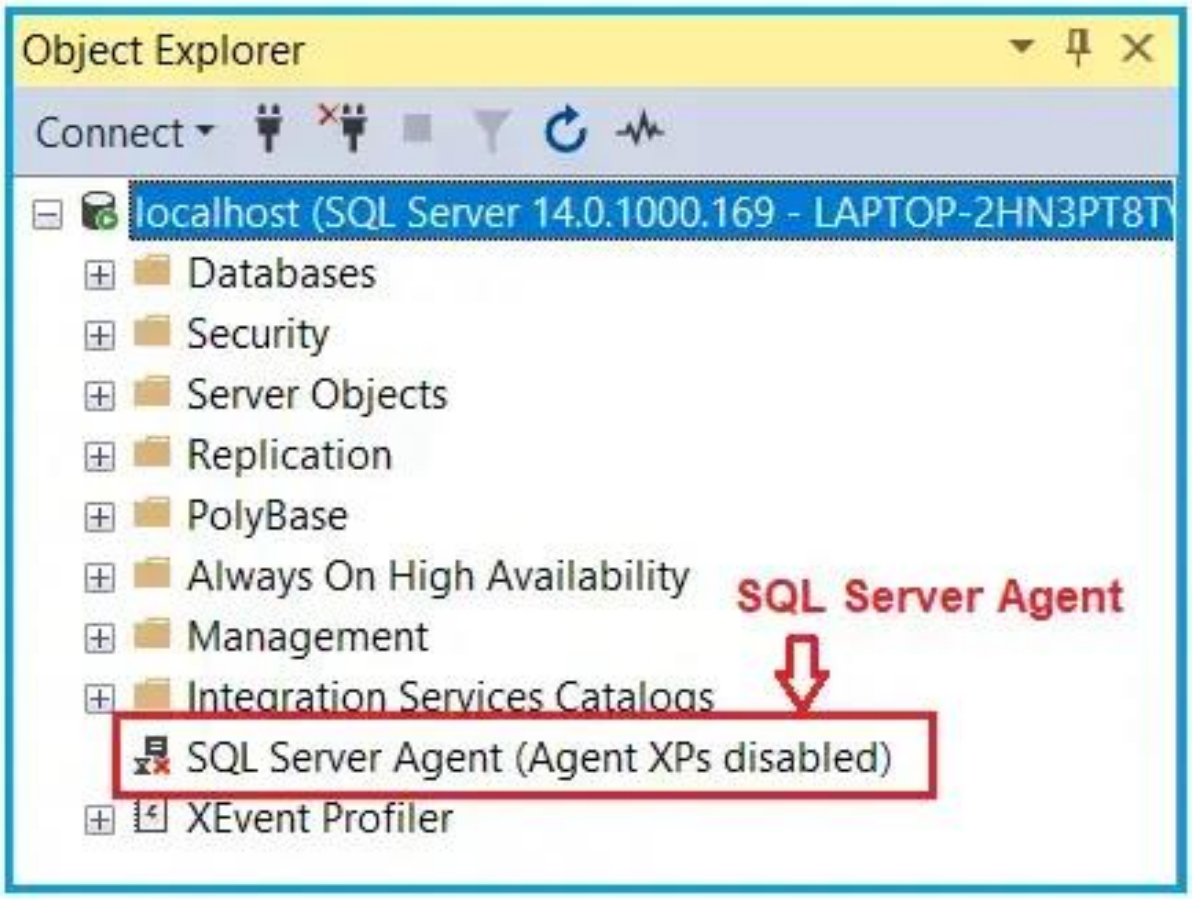
So, this proves that it is not always possible to reverse what the PIVOT operator has done using the UNPIVOT operator and get the original data back.

How to schedule a job in SQL Server using SQL Server Agent

What is the SQL Server Agent?

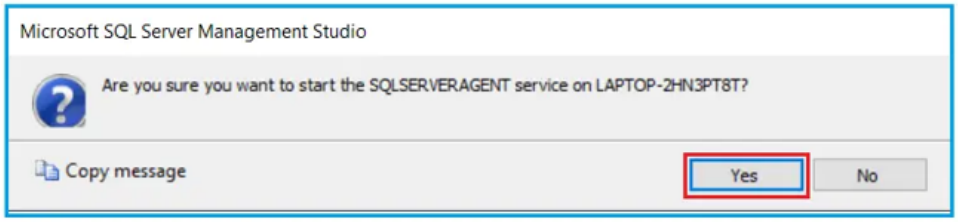
The SQL Server Agent is nothing but a feature provided by SQL Server which will help us to run a job (task) after a specific time interval. For example, if you want to run a backup process every night or if you want to execute a task after a specific time interval, then in such scenarios the SQL Server Agent comes into the picture.

When you open your SQL Server Management Studio, then you will find a small icon called SQL Server Agent as shown in the below image.

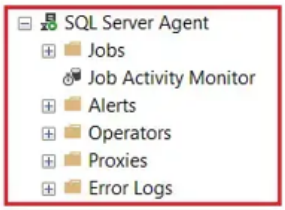


How to use SQL Server Agent to Schedule a Job?

As you can see, by default the SQL Server Agent is disabled. So, first, we need to enable SQL Server Agent. To do so, right-click on the SQL Server Agent and click on the Start option which will prompt a popup asking for do you want to start the SQL Server Agent and you simply need to click on the Yes option as shown in the below image.

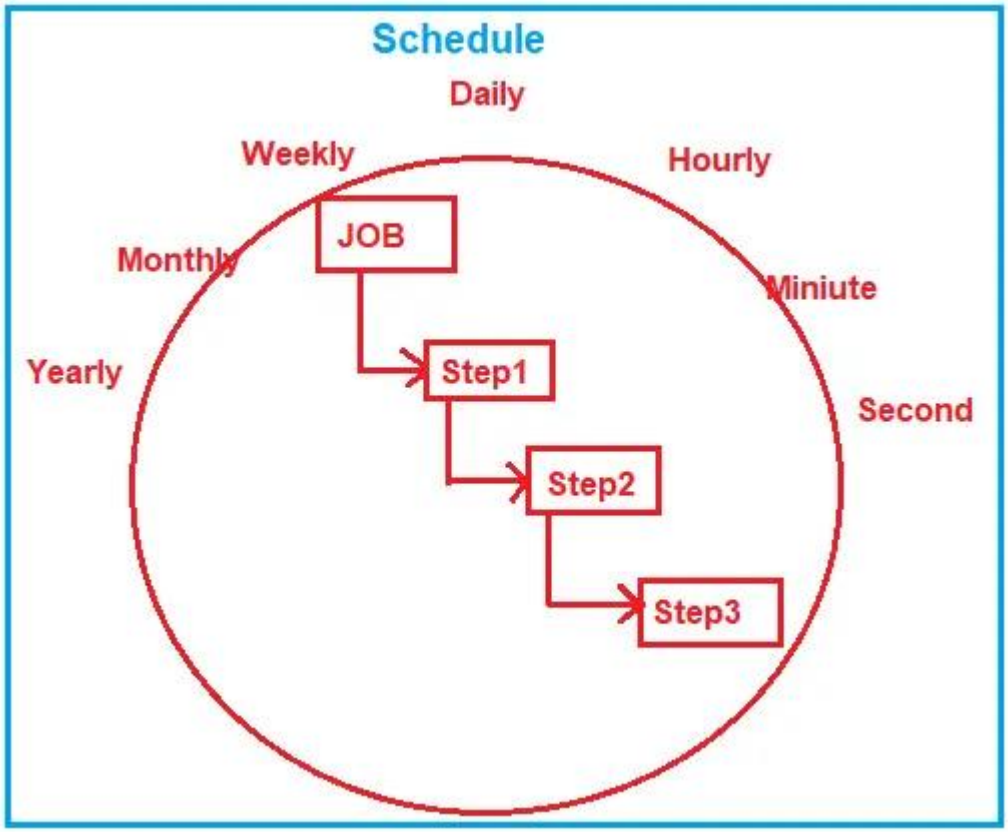


Once you click on the Yes option, then it will take some time to start the SQL Server Agent service. Once the SQL Server Agent is started, just refresh the SQL Server Agent and expand it, you will find the following things within the SQL Server Agent.



What is Job?

A job can have a series of steps or a series of logic that you want to execute one after another. So, in SQL Server Agent, you can create Jobs and inside each job, you can define one or more steps, and all these steps you can run one after another. For better understanding please have a look at the following diagram.



Understanding How to Schedule Job in SQL Server with an example:

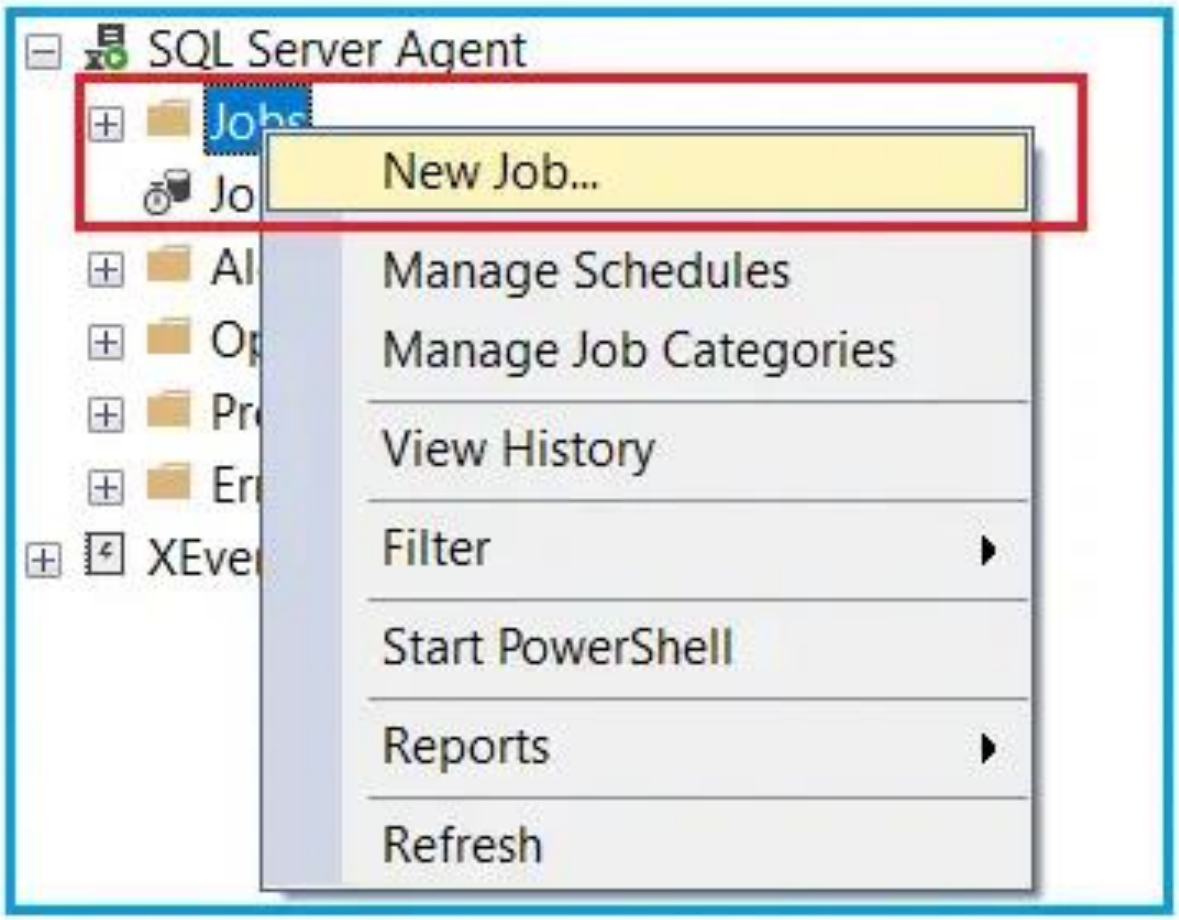
We are going to work with the following **SQLServerTutorial** database and **Orders** and **OrdersHistory** tables. we need to insert some data into the Orders table

Business Requirements:

Our business requirement is to create a job that will run every 30 minutes. In that job first, it will insert all the records which are present in the **Orders** table into the **OrdersHistory** table, and then it will delete all the records from the Orders table.

Step1: Creating Job using SQL Server Agent

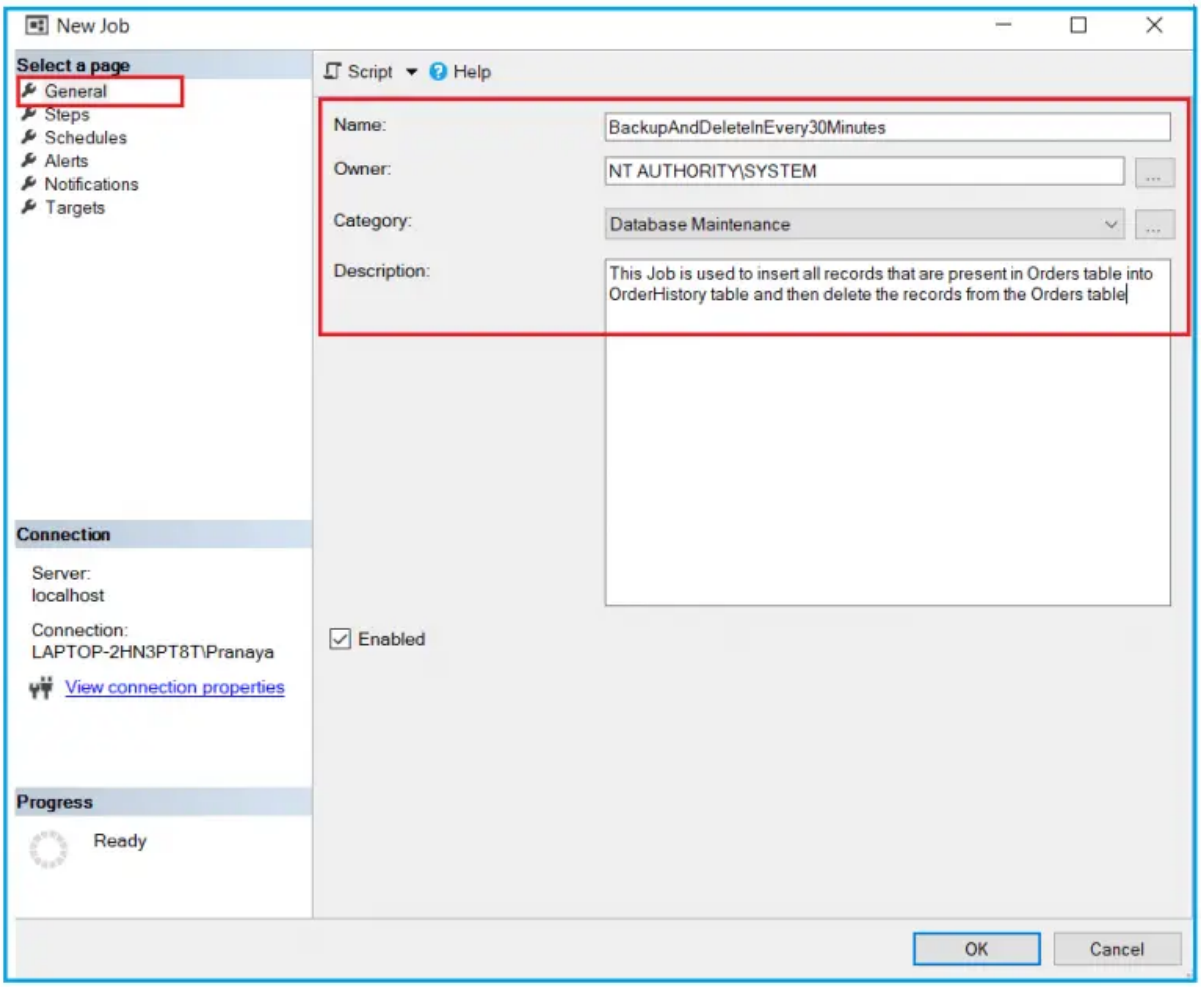
Right-click on the Jobs folder and select the new job option from the context menu as shown in the below image.



Once you click on the New Job option, it will open the Job window. From the Job window, select the General tab and provide the following detail. Provide a meaningful name to your job. Here, I am providing the name as BackupAndDeleteInEvery30Minutes.

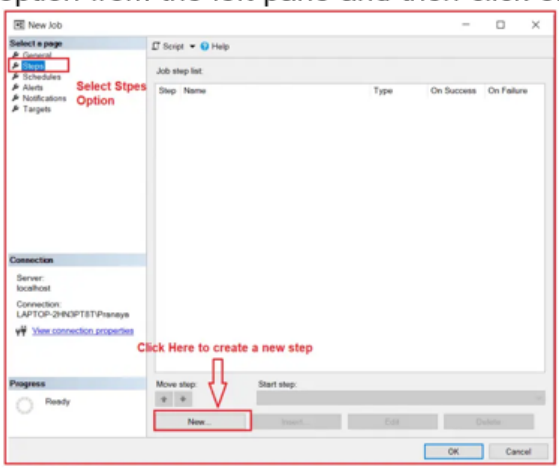
In the Owner section, you need to specify the user account on which this job is going to run. The user should have all the rights to fire the SQL statements. So, here I am selecting the NT AUTHORITY\SYSTEM account by clicking on the browse option which appears next to the Owner text box.

In the category section, you need to specify the type of Job. And here the job type is Database Maintenance. Here, I am selecting the Database Maintenance from the Category dropdown list. In the Description textbox, provide a description of your job as shown in the below image.



Step2: Creating Job Steps

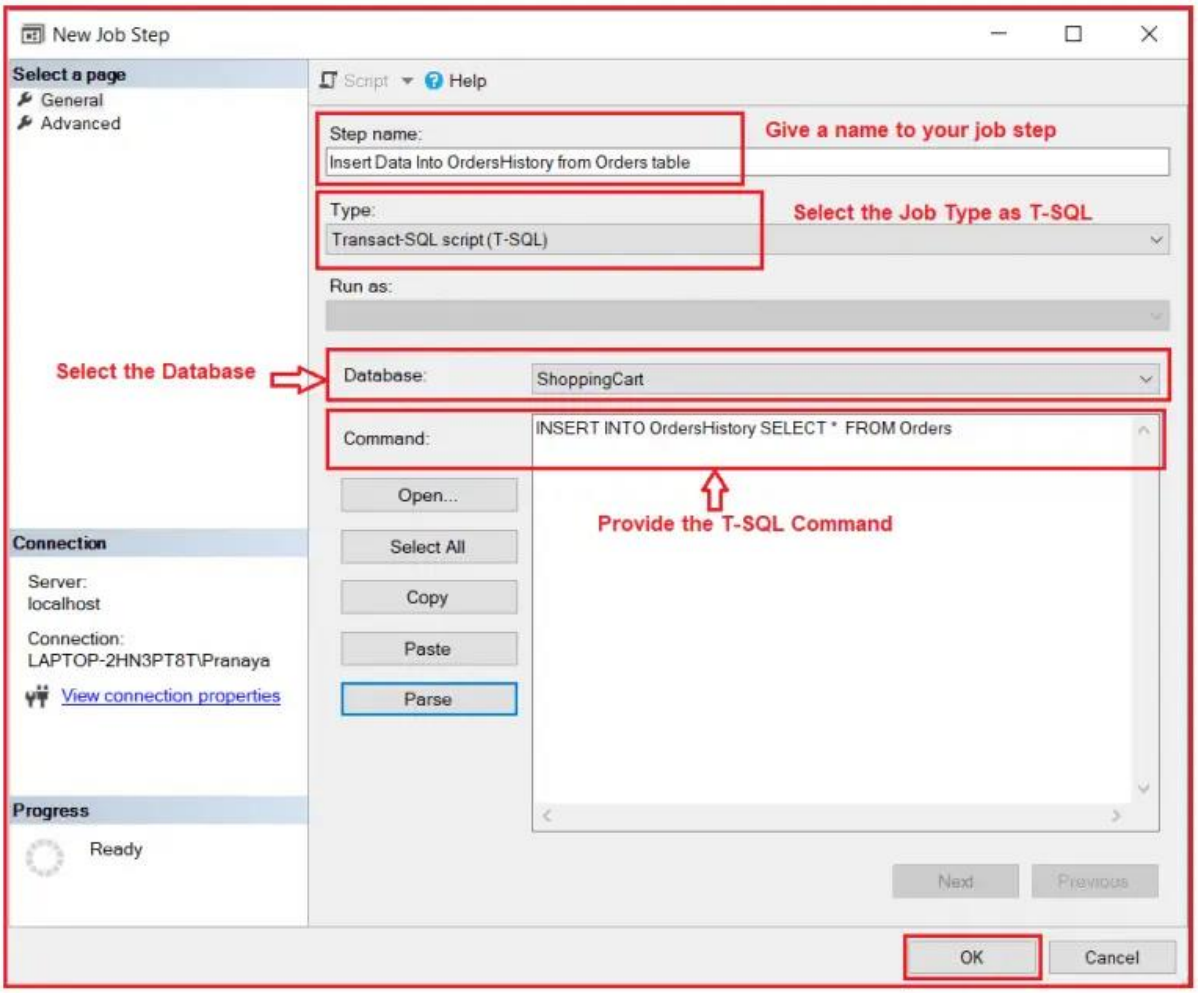
Once you create the job, next you need to define the steps as we already discussed a job is nothing but a series of steps that are going to be executed one after another. In order to create a step, first, select the Steps option from the left pane and then click on the New button as shown in the below image.



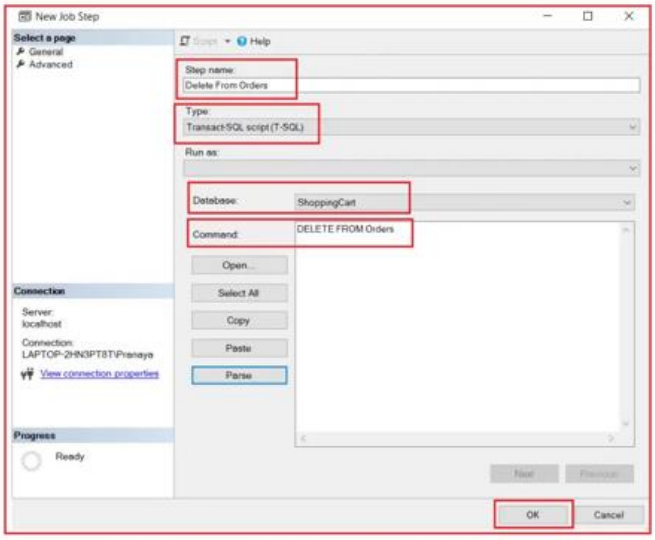
Once you click on the New button, it will open the new JOB step window. From this window provide the following details.

First, you need to provide a meaningful name to your job step. Here, I provided the name Insert Data Into OrdersHistory from Orders table. Then you need to select the type of job. Here, I am selecting the Job type as T-SQL as I am going to write a T-SQL Statement as part of this Job Step.

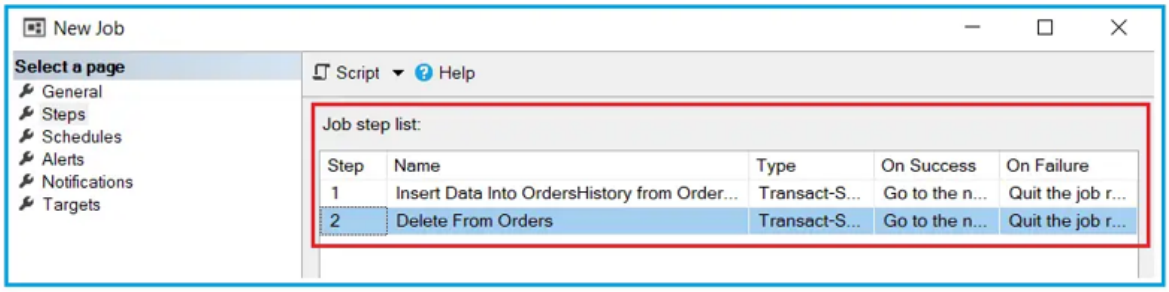
Then you need to select the database. As, my required tables are within the ShoppingCart database, so here, I am selecting the database as ShoppingCart. Finally, you need to write the required T-SQL statement in the Command Text box. Here, my T-SQL statement is INSERT INTO OrdersHistory SELECT * FROM Orders which is insert all the records into the OrderHistory table which are present in the Orders table. Then I click on the Ok button which will create job step 1 as shown in the below image.



Once you click on the Ok button, then it will create job step 1 successfully. Then we need to create Job Step 2 which is basically used to delete all the records from the Orders table. To do so, again click on the New button and provide the details as shown in the below image.



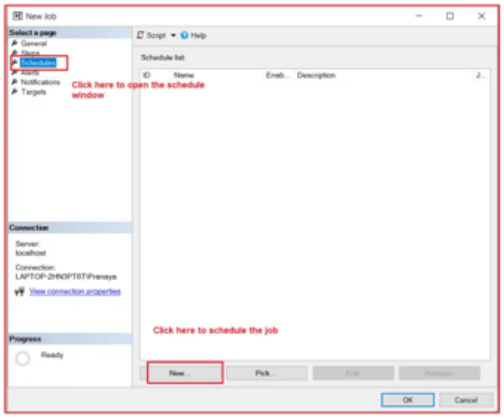
Here, we specify the command as DELETE FROM Orders. And once you click on the OK button, it should create Job Step 2 and you can see both job steps in the job step lists as shown below.



In this way, you can define a series of steps for a particular job in SQL Server and these steps are going to be executed one after another in the Step sequence. That means first step 1, then step 2, and so on. If you want then you can also change the job step sequence by using the move step buttons.

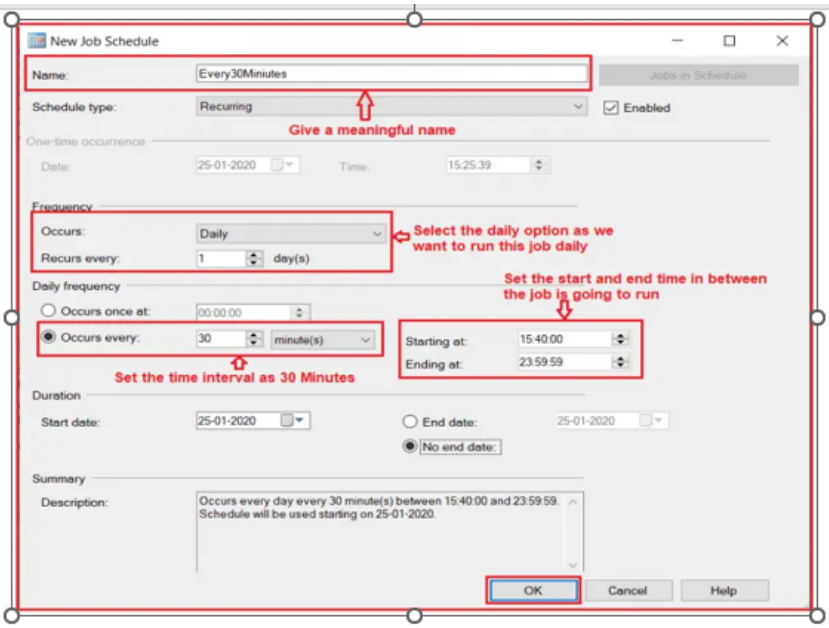
Step3: Scheduling the Job in SQL Server Agent

Once you created the job and the job steps, then the next thing that you need to do is, you need to define the schedule or time interval for this job to run. To do so, click on the Schedule option from the left pane which should open the window to create the schedule. From this scheduled window, just click on the new button as shown in the below image.

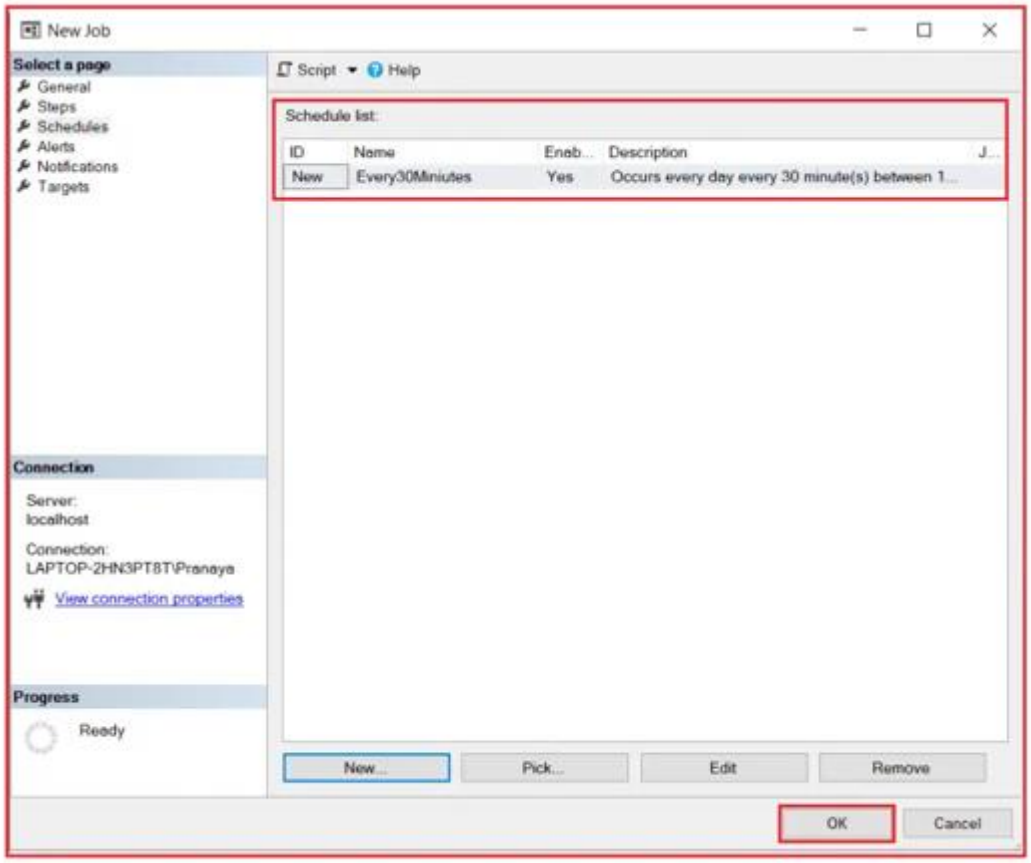


Once you click on the New Button, it will open the window where you need to provide the required information to schedule this job. First, you need to provide a meaningful name to your schedule. Here, I am providing the name as Every30Miniutes in the Name text box.

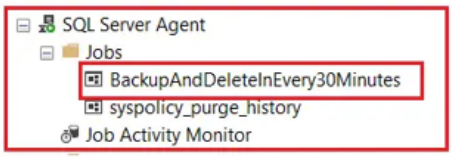
As we want to run this job daily, so here we need to set the Occurs dropdown value as Daily. Then select the occurs every radio button and provide the Occurs Every value as 30 and then select the Minutes radio button and then provide the start time i.e. when the job is going to execute for the first time. Currently, in my machine, it is 15:45 PM, so I am giving the start time as 15:50 and then click on the Ok button as shown below.



Once you click on the OK button, it will create the schedule as shown below. Now, finally, click on the Ok button which will run the job in every 30 minutes time interval.

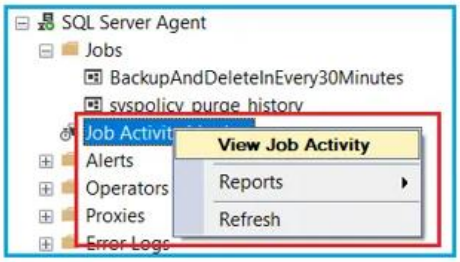


Once you click on the OK button, then you can see this job in the Jobs folder as shown below.



Job Activity Monitor in SQL Server Agent:

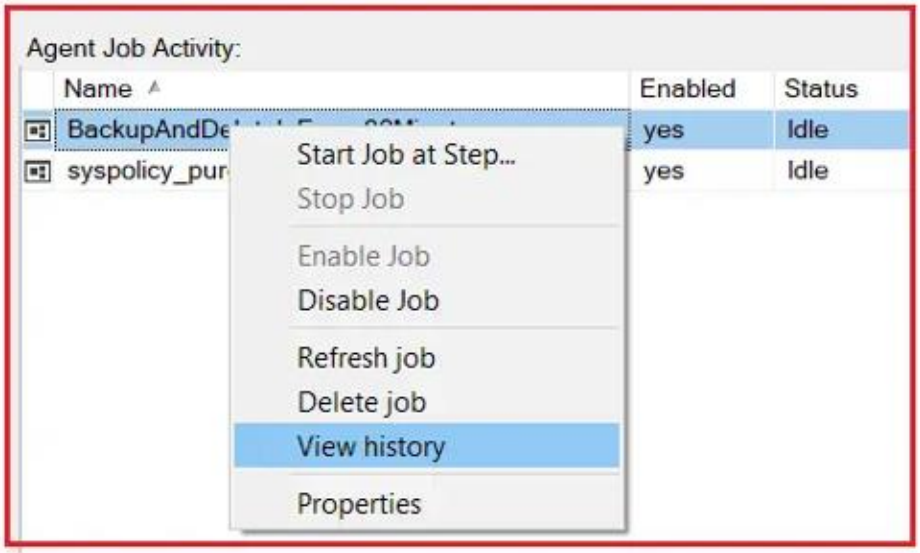
If you want to check whether the job is run successfully or whether you got any error, then you need to use the Job Activity Monitor which you can find in SQL Server Agent. To use this, simply, right-click on the Job Activity Monitor and select View Job Activity from the context menu as shown below.



Once you select the View Job Activity option, then it will open the below window.

As you can see, the status is Idle means the job is not yet run. We need to wait the time we specified. Once the job is run, we will get the following in the View Job Activity. Please click on the refresh option to check the latest status.

If you get any error, then please check the user credentials which is used to run the job. If you want to see the job history, then simply right-click on the Job in View Job Activity window and click on the View history option as shown below.



Once you click on the View history option, it will open the below window which shows the history of this job.