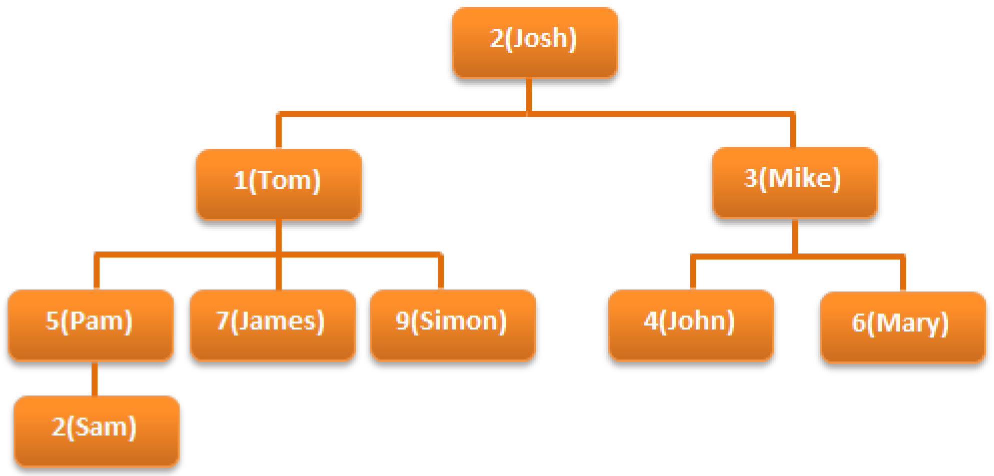
- **Recursive CTE - Part 51**

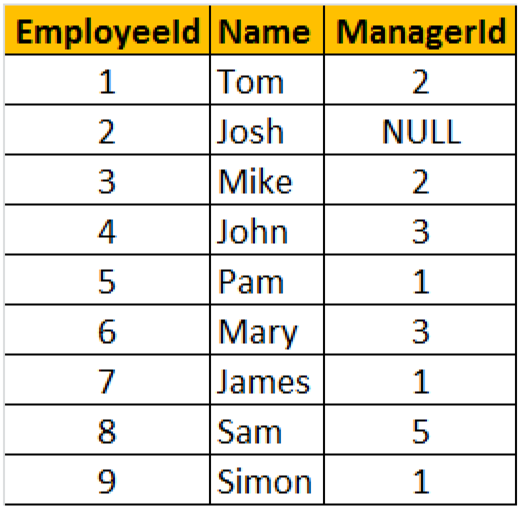
A CTE that references itself is called as recursive CTE. Recursive CTE's can be of great help when displaying hierarchical data. Example, displaying employees in an organization hierarchy. A simple organization chart is shown below.



Let's create tblEmployee table, which holds the data, that's in the organization chart.

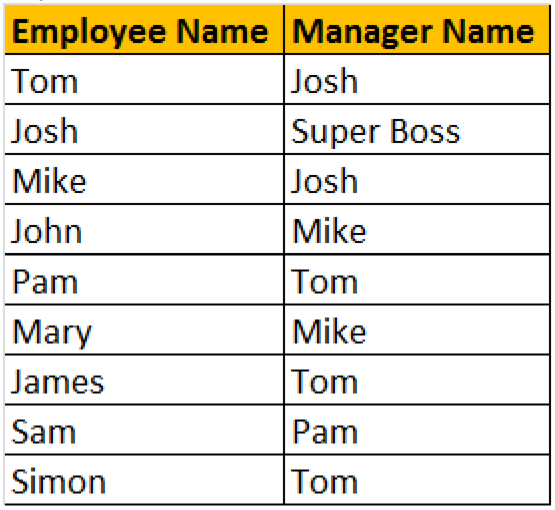
```
Create Table tblEmployee
(
  EmployeeId int Primary key,
  Name nvarchar(20),
  ManagerId int
)

Insert into tblEmployee values (1, 'Tom', 2)
Insert into tblEmployee values (2, 'Josh', null)
Insert into tblEmployee values (3, 'Mike', 2)
Insert into tblEmployee values (4, 'John', 3)
Insert into tblEmployee values (5, 'Pam', 1)
Insert into tblEmployee values (6, 'Mary', 3)
Insert into tblEmployee values (7, 'James', 1)
Insert into tblEmployee values (8, 'Sam', 5)
Insert into tblEmployee values (9, 'Simon', 1)
```

Since, a MANAGER is also an EMPLOYEE, both manager and employee details are stored in tblEmployee table. Data from tblEmployee is shown below.



EmployeeId	Name	ManagerId
1	Tom	2
2	Josh	NULL
3	Mike	2
4	John	3
5	Pam	1
6	Mary	3
7	James	1
8	Sam	5
9	Simon	1

Let's say, we want to display, EmployeeName along with their ManagerName. The ouptut should be as shown below.



Employee Name	Manager Name
Tom	Josh
Josh	Super Boss
Mike	Josh
John	Mike
Pam	Tom
Mary	Mike
James	Tom
Sam	Pam
Simon	Tom

To achieve this, we can simply join tblEmployee with itself. Joining a table with itself is called as self join. We discussed about **Self Joins in Part 14** of this video series. In the output, notice that since **JOSH** does not have a Manager, we are displaying **'Super Boss'**, instead of **NULL**. We used **IsNull**(), function to replace NULL with 'Super Boss'. If you want to learn more about **replacing NULL values, please watch Part 15**.

SELF JOIN QUERY:

```
Select Employee.Name as [Employee Name],
IsNull(Manager.Name, 'Super Boss') as [Manager Name]
from tblEmployee Employee
left join tblEmployee Manager
on Employee.ManagerId = Manager.EmployeeId
```

Along with Employee and their Manager name, we also want to display their level in the organization. The output should be as shown below.

Employee	Manager	Level
Josh	Super Boss	1
Tom	Josh	2
Mike	Josh	2
John	Mike	3
Mary	Mike	3
Pam	Tom	3
James	Tom	3
Simon	Tom	3
Sam	Pam	4

We can easily achieve this using a self referencing CTE.

```
With
  EmployeesCTE (EmployeeId, Name, ManagerId, [Level])
  as
  (
    Select EmployeeId, Name, ManagerId, 1
    from tblEmployee
    where ManagerId is null

    union all

    Select tblEmployee.EmployeeId, tblEmployee.Name,
    tblEmployee.ManagerId, EmployeesCTE.[Level] + 1
    from tblEmployee
    join EmployeesCTE
    on tblEmployee.ManagerID = EmployeesCTE.EmployeeId
  )
Select EmpCTE.Name as Employee, Isnull(MgrCTE.Name, 'Super Boss') as Manager,
EmpCTE.[Level]
from EmployeesCTE EmpCTE
left join EmployeesCTE MgrCTE
on EmpCTE.ManagerId = MgrCTE.EmployeeId
```

The **EmployeesCTE** contains 2 queries with **UNION ALL** operator. The first query selects the EmployeeId, Name, ManagerId, and 1 as the level from **tblEmployee** where ManagerId is NULL. So, here we are giving a LEVEL = 1 for **super boss** (Whose Manager Id is NULL). In the second query, we are joining **tblEmployee** with **EmployeesCTE** itself, which allows us to loop thru the hierarchy. Finally to get the reuired output, we are joining **EmployeesCTE** with itself.

- <u>**Database Normalization - Part 52**</u>

What are the goals of database normalization?
or
Why do we normalize databases?
or
What is database normalization?

Database normalization is the process of organizing data to minimize data redundancy (data duplication), which in turn ensures data consistency

Let's understand with an example, how **redundant data** can cause **data inconsistency**. Consider **Employees** table below. For every employee with in the same department, we are repeating, all the 3 columns (DeptName, DeptHead and DeptLocation). Let's say for example, if there 50 thousand employees in the IT department, we would have unnecessarily repeated all the 3 department columns (DeptName, DeptHead and DeptLocation) data 50 thousand times. The obvious problem with redundant data is the disk space wastage.

EmployeeName	Gender	Salary	DeptName	DeptHead	DeptLocation
Sam	Male	4500	IT	John	London
Pam	Female	2300	HR	Mike	Sydney
Simon	Male	1345	IT	John	London
Mary	Female	2567	HR	Mike	Sydney
Todd	Male	6890	IT	John	London

Another common problem, is that data can become inconsistent. For example, let's say, JOHN has resigned, and we have a new department head (STEVE) for IT department. At present, there are 3 IT department rows in the table, and we need to update all of them. Let's assume I updated only one row and forgot to update the other 2 rows, then obviously, the data becomes inconsistent.

EmployeeName	Gender	Salary	DeptName	DeptHead	DeptLocation
Sam	Male	4500	IT	John	London
Pam	Female	2300	HR	Mike	Sydney
Simon	Male	1345	IT	STEVE	London
Mary	Female	2567	HR	Mike	Sydney
Todd	Male	6890	IT	John	London

Another problem, DML queries (Insert, update and delete), **could become slow**, as there could many records and columns to process.

So, to reduce the data redundancy, we can divide this large badly organised table into two (Employees and Departments), as shown below. Now, we have reduced redundant department data. So, if we have to update department head name, we only have one row to update, even if there are 10 million employees in that department.

Normalized Departments Table

DeptId	DeptName	DeptHead	DeptLocation
1	IT	John	London
2	HR	Mike	Sydney

Normalized Employees Table

EmployeeId	EmployeeName	Gender	Salary	DeptId
1	Sam	Male	4500	1
2	Pam	Female	2300	2
3	Simon	Male	1345	1
4	Mary	Female	2567	2
5	Todd	Male	6890	1

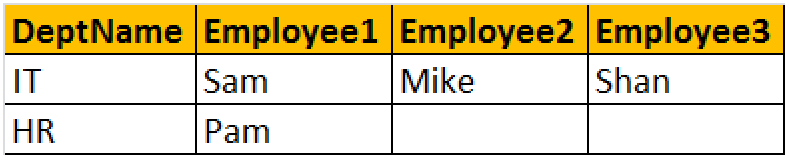
Database normalization is a step by step process. There are 6 normal forms, First Normal form (1NF) thru Sixth Normal Form (6NF). Most databases are in third normal form (3NF). There are certain rules, that each normal form should follow.

Now, let's explore the first normal form (1NF). A table is said to be in 1NF, if
1. The data in each column should be **atomic**. No multiple values, sepearated by comma.
2. The table does not contain any **repeating column groups**
3. Identify each record **uniquely using primary key**.

In the table below, data in Employee column is not atomic. It contains multiple employees seperated by comma. From the data you can see that in the IT department, we have 3 employees - Sam, Mike, Shan. Now, let's say I want to change just, SHAN name. **It is not possible, we have to update the entire cell.** Similary it is not possible to select or delete just one employee, as the data in the cell is not atomic.

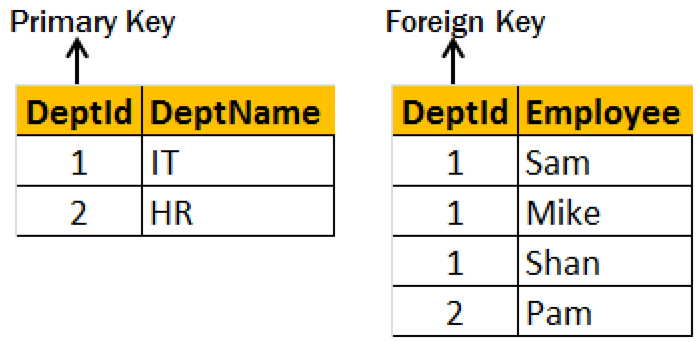
DeptName	Employee
IT	Sam, Mike, Shan
HR	Pam

The 2nd rule of the first normal form is that, the table should not contain any repeating column groups. Consider the Employee table below. We have repeated the Employee column, from Employee1 to Employee3. The problem with this design is that, if a department is going to have more than 3 employees, then we have to **change the table structure** to add Employee4 column. Employee2 and Employee3 columns in the HR department are NULL, as there is only employee in this department. The **disk space is simply wasted.**



DeptName	Employee1	Employee2	Employee3
IT	Sam	Mike	Shan
HR	Pam		

To eliminate the repeating column groups, we are dividing the table into 2. The repeating Employee columns are moved into a seperate table, with a foreign key pointing to the primary key of the other table. We also, introduced primary key to uniquely identify each record.



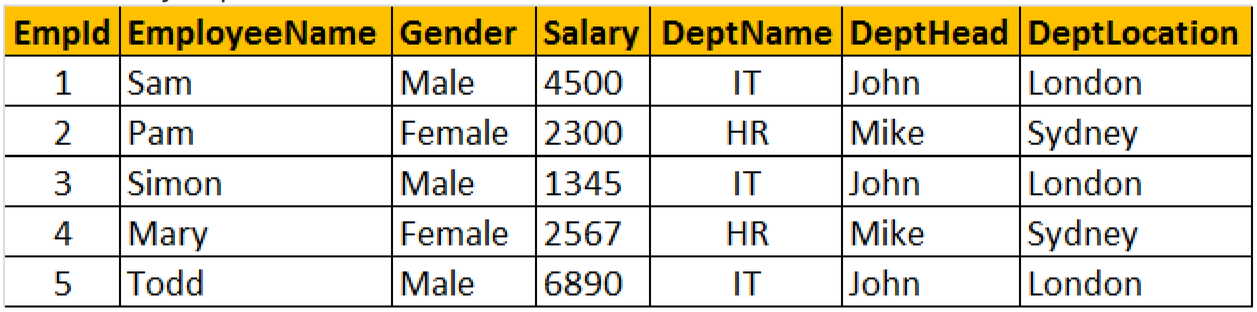
DeptId	DeptName
1	IT
2	HR

DeptId	Employee
1	Sam
1	Mike
1	Shan
2	Pam

Second Normal Form and Third Normal Form - Part 53

In this video will learn about second normal form (2NF) and third normal form (3NF)
A table is said to be in 2NF, if
1. The table meets all the **conditions of 1NF**
2. Move **redundant** data to a separate table
3. Create **relationship** between these tables using foreign keys.

The table below violates second normal form. There is lot of redundant data in the table. Let's say, in my organization there are 100,000 employees and only 2 departments (**IT & HR**). Since we are storing **DeptName, DeptHead and DeptLocation** columns also in the same table, all these columns should also be repeated 100,000 times, which results in unnecessary duplication of data.

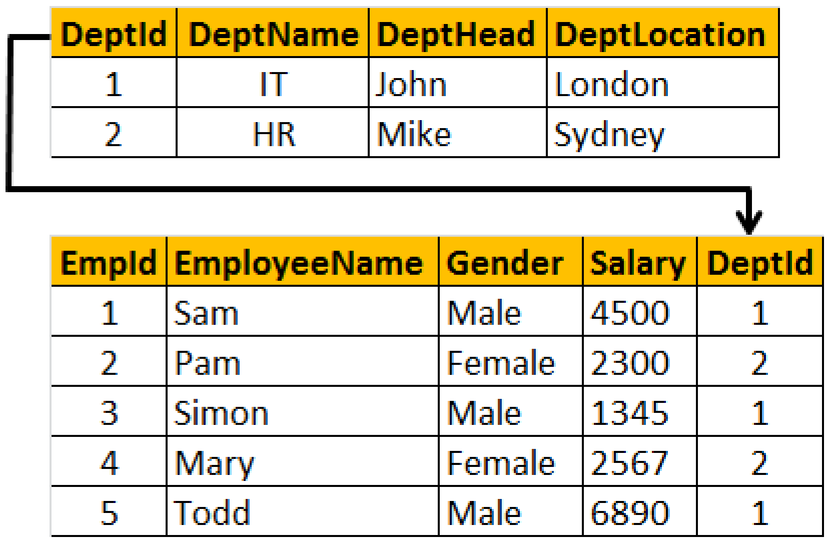


EmpId	EmployeeName	Gender	Salary	DeptName	DeptHead	DeptLocation
1	Sam	Male	4500	IT	John	London
2	Pam	Female	2300	HR	Mike	Sydney
3	Simon	Male	1345	IT	John	London
4	Mary	Female	2567	HR	Mike	Sydney
5	Todd	Male	6890	IT	John	London

So this table is clearly violating the rules of the second normal form, and the redundant data can cause the following issues.
1. Disk space wastage
2. Data inconsistency
3. DML queries (Insert, Update, Delete) can become slow

Now, to put this table in the second normal form, we need to break the table into 2, and move the redundant department data (**DeptName, DeptHead and DeptLocation**) into it's own table. To link the tables with each other, we use the **DeptId** foreign key. The tables below are in 2NF.

Table design in Second Normal Form (2NF)



DeptId	DeptName	DeptHead	DeptLocation
1	IT	John	London
2	HR	Mike	Sydney

EmpId	EmployeeName	Gender	Salary	DeptId
1	Sam	Male	4500	1
2	Pam	Female	2300	2
3	Simon	Male	1345	1
4	Mary	Female	2567	2
5	Todd	Male	6890	1

Third Normal Form (3NF):

A table is said to be in 3NF, if the table
1. Meets all the conditions of **1NF and 2NF**
2. Does not contain columns (attributes) that are not fully **dependent upon the primary key**

The table below, violates third normal form, because **AnnualSalary** column is not fully dependent on the primary key **EmpId**. The **AnnualSalary** is also dependent on the **Salary** column. In fact, to compute the **AnnualSalary**, we multiply the **Salary** by **12**. Since **AnnualSalary** is not fully dependent on the primary key, and it can be computed, we can remove this column from the table, which then, will adhere to 3NF.

EmpId	EmployeeName	Gender	Salary	AnnualSalary	DeptId
1	Sam	Male	4500	54000	1
2	Pam	Female	2300	27600	2
3	Simon	Male	1345	16140	1
4	Mary	Female	2567	30804	2
5	Todd	Male	6890	82680	1

Let's look at another example of Third Normal Form violation. In the table below, **DeptHead** column is not fully dependent on **EmpId** column. **DeptHead** is also dependent on **DeptName**. So, this table is not in **3NF**.

EmpId	EmployeeName	Gender	Salary	DeptName	DeptHead
1	Sam	Male	4500	IT	John
2	Pam	Female	2300	HR	Mike
3	Simon	Male	1345	IT	John
4	Mary	Female	2567	HR	Mike
5	Todd	Male	6890	IT	John

To put this table in 3NF, we break this down into 2, and then move all the columns that are not fully dependent on the primary key to a separate table as shown below. This design is now in 3NF.

Table design in Third Normal Form (3NF)

DeptId	DeptName	DeptHead
1	IT	John
2	HR	Mike

EmpId	EmployeeName	Gender	Salary	DeptId
1	Sam	Male	4500	1
2	Pam	Female	2300	2
3	Simon	Male	1345	1
4	Mary	Female	2567	2
5	Todd	Male	6890	1

Pivot operator in sql server - Part 54

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

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

```
Create Table tblProductSales
(
 SalesAgent nvarchar(50),
 SalesCountry nvarchar(50),
 SalesAmount int
)

Insert into tblProductSales values('Tom', 'UK', 200)
Insert into tblProductSales values('John', 'US', 180)
Insert into tblProductSales values('John', 'UK', 260)
Insert into tblProductSales values('David', 'India', 450)
Insert into tblProductSales values('Tom', 'India', 350)
Insert into tblProductSales values('David', 'US', 200)
Insert into tblProductSales values('Tom', 'US', 130)
Insert into tblProductSales values('John', 'India', 540)
Insert into tblProductSales values('John', 'UK', 120)
Insert into tblProductSales values('David', 'UK', 220)
Insert into tblProductSales values('John', 'UK', 420)
Insert into tblProductSales values('David', 'US', 320)
Insert into tblProductSales values('Tom', 'US', 340)
Insert into tblProductSales values('Tom', 'UK', 660)
Insert into tblProductSales values('John', 'India', 430)
Insert into tblProductSales values('David', 'India', 230)
Insert into tblProductSales values('David', 'India', 280)
Insert into tblProductSales values('Tom', 'UK', 480)
Insert into tblProductSales values('John', 'US', 360)
Insert into tblProductSales values('David', 'UK', 140)
```

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

SalesAgent	SalesCountry	SalesAmount
Tom	UK	200
John	US	180
John	UK	260
David	India	450
Tom	India	350
David	US	200
Tom	US	130
John	India	540
John	UK	120
David	UK	220
John	UK	420
David	US	320
Tom	US	340
Tom	UK	660
John	India	430
David	India	230
David	India	280
Tom	UK	480
John	US	360
David	UK	140

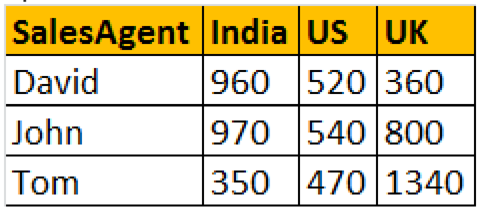
Now, let's write a query which returns TOTAL SALES, grouped by **SALESCOUNTRY** and **SALESAGENT**. The output should be as shown below.

SalesCountry	SalesAgent	Total
India	David	960
India	John	970
India	Tom	350
UK	David	360
UK	John	800
UK	Tom	1340
US	David	520
US	John	540
US	Tom	470

A simple GROUP BY query can produce this output.

```
Select SalesCountry, SalesAgent, SUM(SalesAmount) as Total
from tblProductSales
group by SalesCountry, SalesAgent
order by SalesCountry, SalesAgent
```

At, this point, let's try to present the same data in different format using PIVOT operator.



SalesAgent	India	US	UK
David	960	520	360
John	970	540	800
Tom	350	470	1340

Query using PIVOT operator:

```
Select SalesAgent, India, US, UK
from tblProductSales
Pivot
(
   Sum(SalesAmount) for SalesCountry in ([India],[US],[UK])
) as PivotTable
```

This PIVOT query is converting the unique column values (India, US, UK) in **SALESCOUNTRY** column, **into Columns** in the output, along with performing aggregations on the **SALESAMOUNT** column. The Outer query, simply, selects **SALESAGENT** column from **tblProductSales** table, along with pivoted columns from the PivotTable.

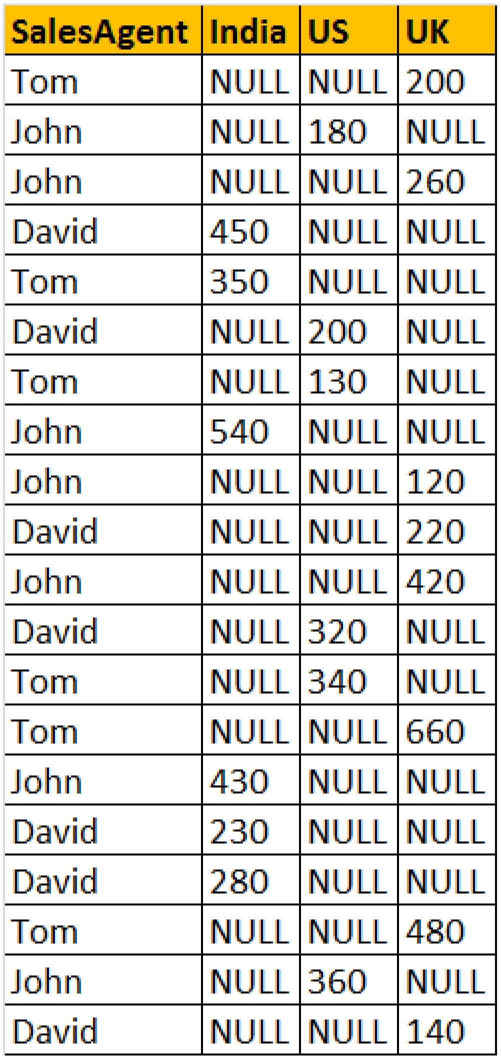
Having understood the basics of PIVOT, let's look at another example. Let's create **tblProductsSale**, a slight variation of **tblProductSales**, that we have already created. The table, that we are creating now, has got an additional **Id** column.

```
Create Table tblProductsSale
(
   Id int primary key,
   SalesAgent nvarchar(50),
   SalesCountry nvarchar(50),
   SalesAmount int
)

Insert into tblProductsSale values(1, 'Tom', 'UK', 200)
Insert into tblProductsSale values(2, 'John', 'US', 180)
Insert into tblProductsSale values(3, 'John', 'UK', 260)
Insert into tblProductsSale values(4, 'David', 'India', 450)
Insert into tblProductsSale values(5, 'Tom', 'India', 350)
Insert into tblProductsSale values(6, 'David', 'US', 200)
Insert into tblProductsSale values(7, 'Tom', 'US', 130)
Insert into tblProductsSale values(8, 'John', 'India', 540)
Insert into tblProductsSale values(9, 'John', 'UK', 120)
Insert into tblProductsSale values(10, 'David', 'UK', 220)
Insert into tblProductsSale values(11, 'John', 'UK', 420)
Insert into tblProductsSale values(12, 'David', 'US', 320)
Insert into tblProductsSale values(13, 'Tom', 'US', 340)
Insert into tblProductsSale values(14, 'Tom', 'UK', 660)
Insert into tblProductsSale values(15, 'John', 'India', 430)
Insert into tblProductsSale values(16, 'David', 'India', 230)
Insert into tblProductsSale values(17, 'David', 'India', 280)
Insert into tblProductsSale values(18, 'Tom', 'UK', 480)
Insert into tblProductsSale values(19, 'John', 'US', 360)
Insert into tblProductsSale values(20, 'David', 'UK', 140)
```

Now, run the same PIVOT query that we have already created, just by changing the name of the table to **tblProductsSale** instead of **tblProductSales**

```
Select SalesAgent, India, US, UK
from tblProductsSale
Pivot
(
   Sum(SalesAmount) for SalesCountry in ([India],[US],[UK])
)
as PivotTable
```

This output is not what we have expected.



SalesAgent	India	US	UK
Tom	NULL	NULL	200
John	NULL	180	NULL
John	NULL	NULL	260
David	450	NULL	NULL
Tom	350	NULL	NULL
David	NULL	200	NULL
Tom	NULL	130	NULL
John	540	NULL	NULL
John	NULL	NULL	120
David	NULL	NULL	220
John	NULL	NULL	420
David	NULL	320	NULL
Tom	NULL	340	NULL
Tom	NULL	NULL	660
John	430	NULL	NULL
David	230	NULL	NULL
David	280	NULL	NULL
Tom	NULL	NULL	480
John	NULL	360	NULL
David	NULL	NULL	140

This is because of the presence of **Id** column in **tblProductsSale**, which is also considered when performing pivoting and group by. To eliminate this from the calculations, we have used derived table, which only selects, **SALESAGENT, SALESCOUNTRY**, and **SALESAMOUNT**. The rest of the query is very similar to what we have already seen.

```
Select SalesAgent, India, US, UK
from
(
   Select SalesAgent, SalesCountry, SalesAmount from tblProductsSale
) as SourceTable
Pivot
(
 Sum(SalesAmount) for SalesCountry in (India, US, UK)
) as PivotTable
```

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

The syntax of PIVOT operator from MSDN

```
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>;
```

Note- Part 55 is the Error handling that's why I skipped that.

Error handling in sql server 2005, and later versions - Part 56

In Part 55, of this video series we have seen Handling errors in SQL Server using @@**Error** system function. In this session we will see, how to achieve the same using Try/Catch blocks.

Syntax:

```
BEGIN TRY
    { Any set of SQL statements }
END TRY
BEGIN CATCH
    [ Optional: Any set of SQL statements ]
END CATCH
[Optional: Any other SQL Statements]
```

Any set of SQL statements, that can possibly throw an exception are wrapped between BEGIN TRY and END TRY blocks. If there is an exception in the TRY block, the control immediately, jumps to the CATCH block. If there is no exception, CATCH block will be skipped, and the statements, after the CATCH block are executed.

Errors trapped by a CATCH block are not returned to the calling application. If any part of the error information must be returned to the application, the code in the CATCH block must do so by using RAISERROR() function.

1. **In procedure spSellProduct**, Begin Transaction and Commit Transaction statements are wrapped between Begin Try and End Try block. If there are no errors in the code that is enclosed in the TRY block, then COMMIT TRANSACTION gets executed and the changes are made permanent. On the other hand, if there is an error, then the control immediately jumps to the CATCH block. In the CATCH block, we are rolling the transaction back. So, it's much easier to handle errors with Try/Catch construct than with @@Error system function.

2. Also notice that, in the scope of the CATCH block, there are several system functions, that are used to retrieve more information about the error that occurred These functions return NULL if they are executed outside the scope of the CATCH block.

3. TRY/CATCH cannot be used in a user-defined functions.

```
Create Procedure spSellProduct
@ProductId int,
@QuantityToSell int
as
Begin
 -- Check the stock available, for the product we want to sell
 Declare @StockAvailable int
 Select @StockAvailable = QtyAvailable
 from tblProduct where ProductId = @ProductId

 -- Throw an error to the calling application, if enough stock is not available
 if(@StockAvailable < @QuantityToSell)
  Begin
  Raiserror('Not enough stock available',16,1)
  End
 -- If enough stock available
 Else
  Begin
   Begin Try
    Begin Transaction
        -- First reduce the quantity available
  Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)
  where ProductId = @ProductId

  Declare @MaxProductSalesId int
  -- Calculate MAX ProductSalesId
  Select @MaxProductSalesId = Case When
          MAX(ProductSalesId) IS NULL
```

```
         Then 0 else MAX(ProductSalesId) end
        from tblProductSales
  --Increment @MaxProductSalesId by 1, so we don't get a primary key violation
  Set @MaxProductSalesId = @MaxProductSalesId + 1
  Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)
    Commit Transaction
   End Try
   Begin Catch
  Rollback Transaction
  Select
   ERROR_NUMBER() as ErrorNumber,
   ERROR_MESSAGE() as ErrorMessage,
   ERROR_PROCEDURE() as ErrorProcedure,
   ERROR_STATE() as ErrorState,
   ERROR_SEVERITY() as ErrorSeverity,
   ERROR_LINE() as ErrorLine
   End Catch
  End
End
```

Transactions in SQL Server - Part 57

What is a Transaction?
A transaction is a group of commands that change the data stored in a database. A transaction, is treated as a single unit. A transaction ensures that, either all of the commands succeed, or none of them. If one of the commands in the transaction fails, all of the commands fail, and any data that was modified in the database is rolled back. In this way, transactions maintain the integrity of data in a database.

Transaction processing follows these steps:
1. Begin a transaction.
2. Process database commands.
3. Check for errors.
   If errors occurred,
       rollback the transaction,
   else,
       commit the transaction

Let's understand transaction processing with an example. For this purpose, let's Create and populate, tblMailingAddress and tblPhysicalAddress tables

```
Create Table tblMailingAddress
(
   AddressId int NOT NULL primary key,
   EmployeeNumber int,
   HouseNumber nvarchar(50),
   StreetAddress nvarchar(50),
   City nvarchar(10),
   PostalCode nvarchar(50)
)

Insert into tblMailingAddress values (1, 101, '#10', 'King Street', 'Londoon', 'CR27DW')


Create Table tblPhysicalAddress
(
 AddressId int NOT NULL primary key,
 EmployeeNumber int,
 HouseNumber nvarchar(50),
 StreetAddress nvarchar(50),
 City nvarchar(10),
 PostalCode nvarchar(50)
)

Insert into tblPhysicalAddress values (1, 101, '#10', 'King Street', 'Londoon', 'CR27DW')
```

An employee with EmployeeNumber 101, has the same address as his physical and mailing address. His city name is mis-spelled as **Londoon** instead of **London**. The following stored procedure **'spUpdateAddress'**, updates the physical and mailing addresses. Both the UPDATE statements are wrapped between **BEGIN TRANSACTION** and **COMMIT TRANSACTION** block, which in turn is wrapped between **BEGIN TRY** and **END TRY** block.

So, if both the UPDATE statements succeed, without any errors, then the transaction is committed. If there are errors, then the control is immediately transferred to the catch block. The **ROLLBACK TRANSACTION** statement, in the CATCH block, rolls back the transaction, and any data that was written to the database by the commands is backed out.

```
Create Procedure spUpdateAddress
as
Begin
 Begin Try
  Begin Transaction
   Update tblMailingAddress set City = 'LONDON'
   where AddressId = 1 and EmployeeNumber = 101

   Update tblPhysicalAddress set City = 'LONDON'
   where AddressId = 1 and EmployeeNumber = 101
  Commit Transaction
 End Try
 Begin Catch
  Rollback Transaction
 End Catch
End
```

Let's now make the second UPDATE statement, fail. CITY column length in tblPhysicalAddress table is 10. The second UPDATE statement fails, because the value for CITY column is more than 10 characters.

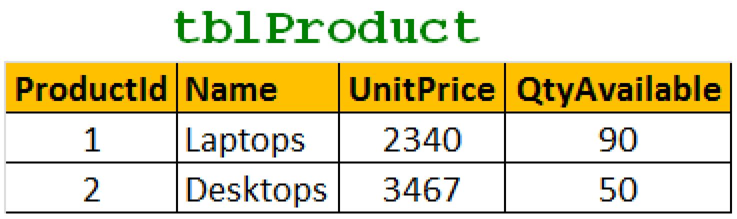
```
Alter Procedure spUpdateAddress
as
Begin
 Begin Try
  Begin Transaction
   Update tblMailingAddress set City = 'LONDON12'
   where AddressId = 1 and EmployeeNumber = 101

   Update tblPhysicalAddress set City = 'LONDON LONDON'
   where AddressId = 1 and EmployeeNumber = 101
  Commit Transaction
 End Try
 Begin Catch
  Rollback Transaction
 End Catch
End
```

Now, if we execute spUpdateAddress, the first **UPDATE** statements succeeds, but the second **UPDATE** statement fails. As, soon as the second UPDATE statement fails, the control is immediately transferred to the CATCH block. The CATCH block rolls the transaction back. So, the change made by the first UPDATE statement is undone.

Transaction Acid Tests - Part 58

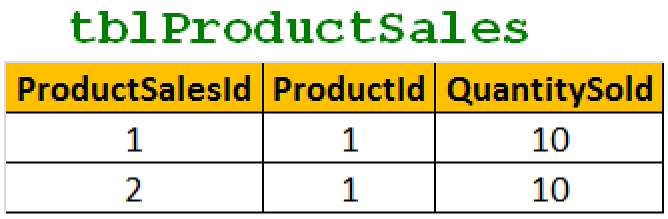
A transaction is a group of database commands that are treated as a single unit. A successful transaction must pass the "ACID" test, that is, it must be
A - Atomic
C - Consistent
I - Isolated
D - Durable

Atomic - All statements in the transaction either completed successfully or they were all rolled back. The task that the set of operations represents is either accomplished or not, but in any case not left half-done. For example, in the **spUpdateInventory_and_Sell** stored procedure, both the UPDATE statements, should succeed. If one UPDATE statement succeeds and the other UPDATE statement fails, the database should undo the change made by the first UPDATE statement, by rolling it back. In short, the transaction should be ATOMIC.



tblProduct

ProductId	Name	UnitPrice	QtyAvailable
1	Laptops	2340	90
2	Desktops	3467	50



tblProductSales

ProductSalesId	ProductId	QuantitySold
1	1	10
2	1	10

```
Create Procedure spUpdateInventory_and_Sell
as
Begin
  Begin Try
    Begin Transaction
      Update tblProduct set QtyAvailable = (QtyAvailable - 10)
      where ProductId = 1

      Insert into tblProductSales values(3, 1, 10)
    Commit Transaction
  End Try
  Begin Catch
    Rollback Transaction
  End Catch
End
```

Consistent - All data touched by the transaction is left in a **logically consistent state**. For example, if stock available numbers are decremented from **tblProductTable**, then, there has to be a related entry in **tblProductSales** table. The inventory can't just disappear.

Isolated - The transaction must affect data without interfering with other concurrent transactions, or being interfered with by them. This prevents transactions from making changes to data based on uncommitted information, for example changes to a record that are subsequently rolled back. **Most databases use locking to maintain transaction isolation**.

Durable - Once a change is made, it is permanent. If a system error or power failure occurs before a set of commands is complete, those commands are undone and the data is restored to its original state once the system begins running again.

Subqueries in sql - Part 59

In this video we will discuss about subqueries in sql server. Let us understand subqueris with an example. Please create the required tables and insert sample data using the script below.

```
Create Table tblProducts
(
 [Id] int identity primary key,
 [Name] nvarchar(50),
 [Description] nvarchar(250)
)

Create Table tblProductSales
(
 Id int primary key identity,
 ProductId int foreign key references tblProducts(Id),
 UnitPrice int,
 QuantitySold int
)

Insert into tblProducts values ('TV', '52 inch black color LCD TV')
Insert into tblProducts values ('Laptop', 'Very thin black color acer laptop')
Insert into tblProducts values ('Desktop', 'HP high performance desktop')

Insert into tblProductSales values(3, 450, 5)
Insert into tblProductSales values(2, 250, 7)
Insert into tblProductSales values(3, 450, 4)
Insert into tblProductSales values(3, 450, 9)
```

Write a query to retrieve products that are not at all sold?
This can be very easily achieved using subquery as shown below. Select [Id], [Name], [Description]
from tblProducts
where Id not in (Select Distinct ProductId from tblProductSales)

Most of the times subqueries can be very easily replaced with joins. The above query is rewritten using joins and produces the same results. Select tblProducts.[Id], [Name], [Description]
from tblProducts

```
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
where tblProductSales.ProductId IS NULL
```

In this example, we have seen how to use a subquery in the where clause.

Let us now discuss about using a sub query in the SELECT clause. Write a query to retrieve the NAME and TOTALQUANTITY sold, using a subquery.Select [Name], (Select SUM(QuantitySold) from tblProductSales where ProductId = tblProducts.Id) as TotalQuantity
from tblProducts
order by Name

Query with an equivalent join that produces the same result.

```
Select [Name], SUM(QuantitySold) as TotalQuantity
from tblProducts
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
group by [Name]
order by Name
```

From these examples, it should be very clear that, a subquery is simply a select statement, that returns a single value and can be nested inside a SELECT, UPDATE, INSERT, or DELETE statement.

It is also possible to nest a subquery inside another subquery.

According to MSDN, subqueries can be nested upto 32 levels.

Subqueries are always encolsed in paranthesis and are also called as inner queries, and the query containing the subquery is called as outer query.

The columns from a table that is present only inside a subquery, cannot be used in the SELECT list of the outer query

Correlated subquery in sql - Part 60

In Part 59, we discussed about 2 examples that uses subqueries. Please watch Part 59, before proceeding with this video. We will be using the same tables and queries from Part 59.

In the example below, sub query is executed first and only once. The sub query results are then used by the outer query. A non-corelated subquery can be executed independently of the outer query.

```
Select [Id], [Name], [Description]
from tblProducts
where Id not in (Select Distinct ProductId from tblProductSales)
```

If the subquery depends on the outer query for its values, then that sub query is called as a correlated subquery. In the where clause of the subquery below, **"ProductId"** column get it's value from **tblProducts** table that is present in the outer query. So, here the subquery is dependent on the outer query for it's value, hence this subquery is a correlated subquery. Correlated subqueries get executed, once for every row that is selected by the outer query. Corelated subquery, cannot be executed independently of the outer query.

```
Select [Name],
(Select SUM(QuantitySold) from tblProductSales where ProductId = tblProducts.Id) as
TotalQuantity
from tblProducts
order by Name
```

Creating a large table with random data for performance testing - Part 61\

In this video we will discuss about inserting large amount of random data into sql server tables for performance testing.

```
-- If Table exists drop the tables
If (Exists (select *
        from information_schema.tables
        where table_name = 'tblProductSales'))
Begin
 Drop Table tblProductSales
End

If (Exists (select *
        from information_schema.tables
        where table_name = 'tblProducts'))
Begin
 Drop Table tblProducts
End


-- Recreate tables
Create Table tblProducts
(
 [Id] int identity primary key,
 [Name] nvarchar(50),
 [Description] nvarchar(250)
)

Create Table tblProductSales
(
 Id int primary key identity,
 ProductId int foreign key references tblProducts(Id),
 UnitPrice int,
 QuantitySold int
)

--Insert Sample data into tblProducts table
Declare @Id int
Set @Id = 1

While(@Id <= 300000)
Begin
 Insert into tblProducts values('Product - ' + CAST(@Id as nvarchar(20)),
 'Product - ' + CAST(@Id as nvarchar(20)) + ' Description')

 Print @Id
 Set @Id = @Id + 1
End

-- Declare variables to hold a random ProductId,
-- UnitPrice and QuantitySold
declare @RandomProductId int
declare @RandomUnitPrice int
declare @RandomQuantitySold int

-- Declare and set variables to generate a
-- random ProductId between 1 and 100000
declare @UpperLimitForProductId int
declare @LowerLimitForProductId int

set @LowerLimitForProductId = 1
set @UpperLimitForProductId = 100000

-- Declare and set variables to generate a
-- random UnitPrice between 1 and 100
declare @UpperLimitForUnitPrice int
declare @LowerLimitForUnitPrice int

set @LowerLimitForUnitPrice = 1
set @UpperLimitForUnitPrice = 100

-- Declare and set variables to generate a
-- random QuantitySold between 1 and 10
declare @UpperLimitForQuantitySold int
declare @LowerLimitForQuantitySold int

set @LowerLimitForQuantitySold = 1
set @UpperLimitForQuantitySold = 10

--Insert Sample data into tblProductSales table
Declare @Counter int
Set @Counter = 1

While(@Counter <= 450000)
Begin
 select @RandomProductId = Round(((@UpperLimitForProductId -
@LowerLimitForProductId) * Rand() + @LowerLimitForProductId), 0)
 select @RandomUnitPrice = Round(((@UpperLimitForUnitPrice -
@LowerLimitForUnitPrice) * Rand() + @LowerLimitForUnitPrice), 0)
 select @RandomQuantitySold = Round(((@UpperLimitForQuantitySold -
@LowerLimitForQuantitySold) * Rand() + @LowerLimitForQuantitySold), 0)

 Insert into tblProductsales
 values(@RandomProductId, @RandomUnitPrice, @RandomQuantitySold)

 Print @Counter
 Set @Counter = @Counter + 1
End
```

Finally, check the data in the tables using a simple SELECT query to make sure the data has been inserted as expected.

```
Select * from tblProducts
Select * from tblProductSales
```

What to choose for performance - SubQueries or Joins - Part 62

According to MSDN, in sql server, in most cases, there is usually no performance difference between queries that uses sub-queries and equivalent queries using joins. For example, on my machine I have
400,000 records in tblProducts table
600,000 records in tblProductSales tables

The following query, returns, the list of products that we have sold atleast once. This query is formed using sub-queries. When I execute this query I get 306,199 rows in 6 seconds

```
Select Id, Name, Description
from tblProducts
where ID IN
(
 Select ProductId from tblProductSales
)
```

At this stage please clean the query and execution plan cache using the following T-SQL command.

```
CHECKPOINT;
GO
DBCC DROPCLEANBUFFERS; -- Clears query cache
Go
DBCC FREEPROCCACHE; -- Clears execution plan cache
GO
```

Now, run the query that is formed using joins. Notice that I get the exact same 306,199 rows in 6 seconds.

```
Select distinct tblProducts.Id, Name, Description
from tblProducts
inner join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
```

Please Note: I have used automated sql script to insert huge amounts of this random data. Please watch Part 61 of SQL Server tutorial, in which we have discussed about this automated script.

According to MSDN, in some cases where existence must be checked, a join produces better performance. Otherwise, the nested query must be processed for each result of the outer query. In such cases, a join approach would yield better results.

The following query returns the products that we have not sold at least once. This query is formed using sub-queries. When I execute this query I get 93,801 rows in 3 seconds

```
Select Id, Name, [Description]
from tblProducts
where Not Exists(Select * from tblProductSales where ProductId = tblProducts.Id)
```

When I execute the below equivalent query, that uses joins, I get the exact same 93,801 rows in 3 seconds.

```
Select tblProducts.Id, Name, [Description]
from tblProducts
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
where tblProductSales.ProductId IS NULL
```

In general joins work faster than sub-queries, but in reality it all depends on the execution plan that is generated by SQL Server. It does not matter how we have written the query, SQL Server will always transform it on an execution plan. If sql server generates the same plan from both queries, we will get the same result.

I would say, rather than going by theory, turn on client statistics and execution plan to see the performance of each option, and then make a decision.

Cursors in sql server - Part 63

Relational Database Management Systems, including sql server are very good at handling data in SETS. For example, the following "UPDATE" query, updates a set of rows that matches the condition in the "WHERE" clause at the same time.

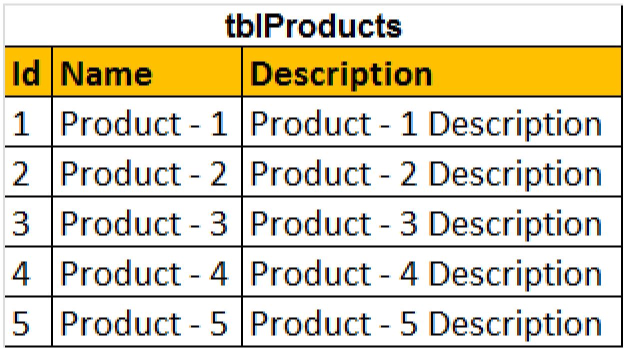
```
Update tblProductSales Set UnitPrice = 50 where ProductId = 101
```

However, if there is ever a need to process the rows, on a row-by-row basis, then cursors are your choice. Cursors are very bad for performance, and should be avoided always. Most of the time, cursors can be very easily replaced using joins.

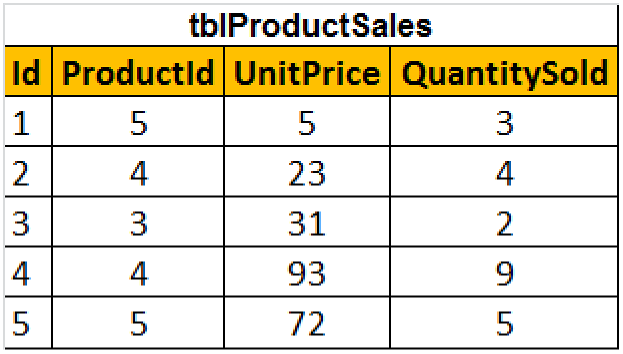
There are different types of cursors in sql server as listed below. We will talk about the differences between these cursor types in a later video session.

1. Forward-Only
2. Static
3. Keyset
4. Dynamic

Let us now look at a simple example of using sql server cursor to process one row at time. We will be using tblProducts and tblProductSales tables, for this example. The tables here show only 5 rows from each table. However, on my machine, there are 400,000 records in tblProducts and 600,000 records in tblProductSales tables. If you want to learn about generating huge amounts of random test data, **please watch Part - 61 in sql server video tutorial.**



tblProducts		
Id	Name	Description
1	Product - 1	Product - 1 Description
2	Product - 2	Product - 2 Description
3	Product - 3	Product - 3 Description
4	Product - 4	Product - 4 Description
5	Product - 5	Product - 5 Description



tblProductSales			
Id	ProductId	UnitPrice	QuantitySold
1	5	5	3
2	4	23	4
3	3	31	2
4	4	93	9
5	5	72	5

Cursor Example: Let us say, I want to update the UNITPRICE column in tblProductSales table, based on the following criteria
1. If the ProductName = 'Product - 55', Set Unit Price to 55
2. If the ProductName = 'Product - 65', Set Unit Price to 65
3. If the ProductName is like 'Product - 100%', Set Unit Price to 1000

```
Declare @ProductId int

-- Declare the cursor using the declare keyword
Declare ProductIdCursor CURSOR FOR
Select ProductId from tblProductSales

-- Open statement, executes the SELECT statment
-- and populates the result set
Open ProductIdCursor

-- Fetch the row from the result set into the variable
Fetch Next from ProductIdCursor into @ProductId

-- If the result set still has rows, @@FETCH_STATUS will be ZERO
While(@@FETCH_STATUS = 0)
Begin
 Declare @ProductName nvarchar(50)
 Select @ProductName = Name from tblProducts where Id = @ProductId
 if(@ProductName = 'Product - 55')
 Begin
  Update tblProductSales set UnitPrice = 55 where ProductId = @ProductId
 End
 else if(@ProductName = 'Product - 65')
 Begin
  Update tblProductSales set UnitPrice = 65 where ProductId = @ProductId
 End
 else if(@ProductName like 'Product - 100%')
 Begin
  Update tblProductSales set UnitPrice = 1000 where ProductId = @ProductId
 End

 Fetch Next from ProductIdCursor into @ProductId
End

-- Release the row set
CLOSE ProductIdCursor
-- Deallocate, the resources associated with the cursor
DEALLOCATE ProductIdCursor
```

The cursor will loop thru each row in tblProductSales table. As there are 600,000 rows, to be processed on a row-by-row basis, it takes around 40 to 45 seconds on my machine. We can achieve this very easily using a join, and this will significantly increase the performance. We will discuss about this in our next video session.

To check if the rows have been correctly updated, please use the following query.

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

Replacing cursors using joins in sql server - Part 64

In Part 63, we have discussed about cursors. The example, in Part 63, took around 45 seconds on my machine. Please watch Part 63, before proceeding with this video. In this video we will re-write the example, using a join.

```
Update tblProductSales
set UnitPrice =
 Case
  When Name = 'Product - 55' Then 155
  When Name = 'Product - 65' Then 165
  When Name like 'Product - 100%' Then 10001
 End
from tblProductSales
join tblProducts
on tblProducts.Id = tblProductSales.ProductId
Where Name = 'Product - 55' or Name = 'Product - 65' or
Name like 'Product - 100%'
```

When I executed this query, on my machine it took less than a second. Where as the same thing using a cursor took 45 seconds. Just imagine the amount of impact cursors have on performance. Cursors should be used as your last option. Most of the time cursors can be very easily replaced using joins.

To check the result of the UPDATE statement, use the following query.

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

Part 65 - List all tables in a sql server database using a query

In this video we will discuss, writing a **transact sql query to list all the tables in a sql server database**. This is a very common sql server interview question.

Object explorer with in sql server management studio can be used to get the list of tables in a specific database. However, if we have to write a query to achieve the same, there are 3 system views that we can use.
1. SYSOBJECTS - Supported in SQL Server version 2000, 2005 & 2008
2. SYS.TABLES - Supported in SQL Server version 2005 & 2008
3. INFORMATION_SCHEMA.TABLES - Supported in SQL Server version 2005 & 2008

```
-- Gets the list of tables only
Select * from SYSOBJECTS where XTYPE='U'
-- Gets the list of tables only
Select * from  SYS.TABLES
-- Gets the list of tables and views
Select * from INFORMATION_SCHEMA.TABLES
```

To get the list of different object types (XTYPE) in a database

```
Select Distinct XTYPE from SYSOBJECTS
```

Executing the above query on my SAMPLE database returned the following values for XTYPE column from SYSOBJECTS
IT - Internal table
P - Stored procedure
PK - PRIMARY KEY constraint
S - System table
SQ - Service queue
U - User table
V - View

Please check the following MSDN link for all possible XTYPE column values and what they represent.
http://msdn.microsoft.com/en-us/library/ms177596.aspx

Writing re-runnable sql server scripts - Part 66

What is a re-runnable sql script?
A re-runnable script is a script, that, when run more than, once will not throw errors.

Let's understand **writing re-runnable sql scripts** with an example. To create a table **tblEmployee** in **Sample** database, we will write the following **CREATE TABLE** sql script.

```
USE [Sample]
Create table tblEmployee
(
 ID int identity primary key,
 Name nvarchar(100),
 Gender nvarchar(10),
 DateOfBirth DateTime
)
```

When you run this script once, the table **tblEmployee** gets created without any errors. If you run the script again, you will get an error - There is already an object named 'tblEmployee' in the database.

To make this script re-runnable
1. Check for the existence of the table
2. Create the table if it does not exist
3. Else print a message stating, the table already exists

```
Use [Sample]
```

```
If not exists (select * from information_schema.tables where table_name = 'tblEmployee')
Begin
 Create table tblEmployee
 (
  ID int identity primary key,
  Name nvarchar(100),
  Gender nvarchar(10),
  DateOfBirth DateTime
 )
 Print 'Table tblEmployee successfully created'
End
Else
Begin
 Print 'Table tblEmployee already exists'
End
```

The above **script is re-runnable**, and can be run any number of times. If the table is not already created, the script will create the table, else you will get a message stating - **The table already exists.** You will never get a sql script error.

Sql server built-in function OBJECT_ID(), can also be used to check for the existence of the table

```
IF OBJECT_ID('tblEmployee') IS NULL
Begin
   -- Create Table Script
   Print 'Table tblEmployee created'
End
Else
Begin
   Print 'Table tblEmployee already exists'
End
```

Depending on what we are trying to achieve, sometime we may need **to drop (if the table already exists) and re-create it**. The sql script below, does exactly the same thing.

```
Use [Sample]
IF OBJECT_ID('tblEmployee') IS NOT NULL
Begin
 Drop Table tblEmployee
End
Create table tblEmployee
(
 ID int identity primary key,
 Name nvarchar(100),
 Gender nvarchar(10),
 DateOfBirth DateTime
)
```

Let's look at another example. The following sql script adds column **"EmailAddress"** to table **tblEmployee**. This script is not re-runnable because, if the column exists we get a script error.

```
Use [Sample]
ALTER TABLE tblEmployee
ADD EmailAddress nvarchar(50)
```

To make this script re-runnable, check for the column existence

```
Use [Sample]
if not exists(Select * from INFORMATION_SCHEMA.COLUMNS where
COLUMN_NAME='EmailAddress' and TABLE_NAME = 'tblEmployee' and
TABLE_SCHEMA='dbo')
Begin
 ALTER TABLE tblEmployee
 ADD EmailAddress nvarchar(50)
End
Else
BEgin
 Print 'Column EmailAddress already exists'
End
```

Col_length() function can also be used to check for the existence of a column

```
If col_length('tblEmployee','EmailAddress') is not null
Begin
 Print 'Column already exists'
End
Else
Begin
 Print 'Column does not exist'
End
```

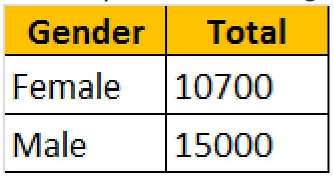
Part 67 - Alter database table columns without dropping table

In this video, we will discuss, **altering a database table column without having the need to drop the table.** Let's understand this with an example.

We will be using table **tblEmployee** for this demo. Use the sql script below, to create and populate this table with some sample data.

```
Create table tblEmployee
(
 ID int primary key identity,
 Name nvarchar(50),
 Gender nvarchar(50),
 Salary nvarchar(50)
)


Insert into tblEmployee values('Sara Nani','Female','4500')
Insert into tblEmployee values('James Histo','Male','5300')
Insert into tblEmployee values('Mary Jane','Female','6200')
Insert into tblEmployee values('Paul Sensit','Male','4200')
Insert into tblEmployee values('Mike Jen','Male','5500')
```

The requirement is to group the salaries by gender. The output should be as shown below.



Gender	Total
Female	10700
Male	15000

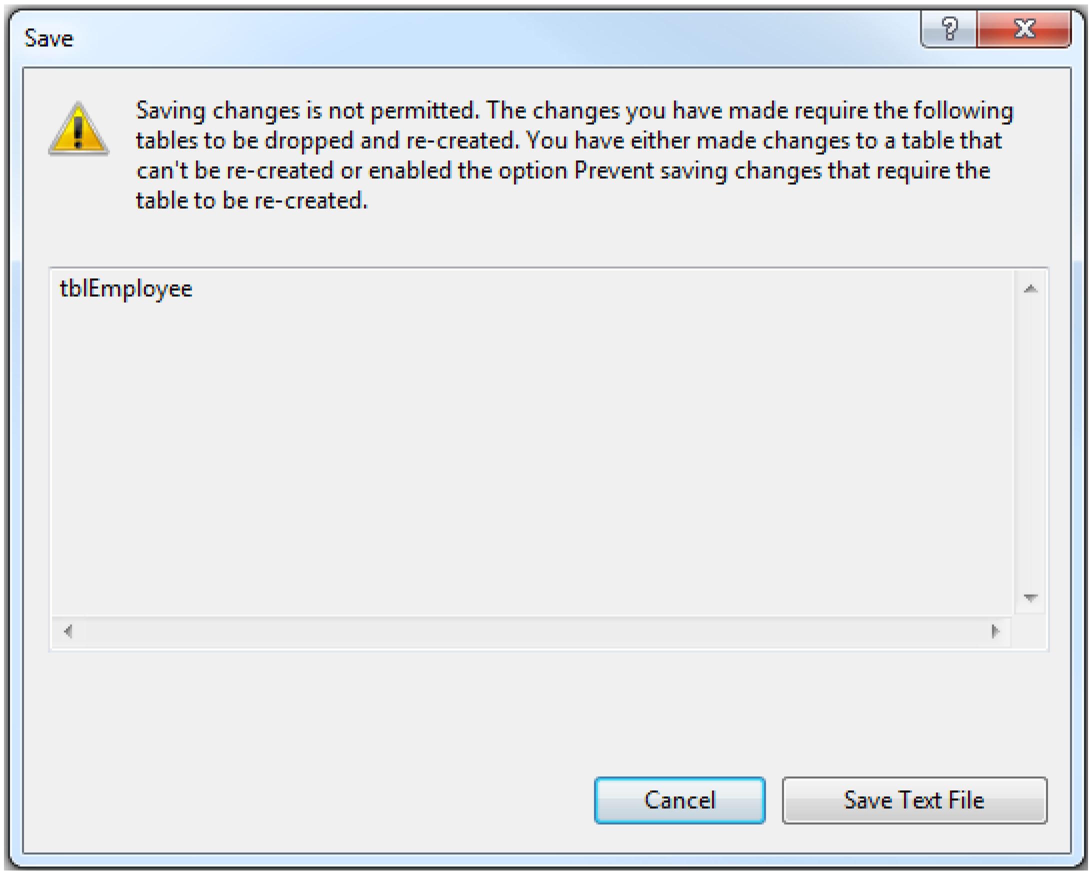
To achieve this we would write a sql query using GROUP BY as shown below.

```
Select Gender, Sum(Salary) as Total
from tblEmployee
Group by Gender
```

When you execute this query, we will get an error - Operand data type nvarchar is invalid for sum operator. This is because, when we created **tblEmployee** table, the **"Salary"** column was created using **nvarchar** datatype. SQL server **Sum**() aggregate function can only be applied on numeric columns. So, let's try to modify **"Salary"** column to use **int** datatype. Let's do it using the designer.

1. Right click on "tblEmployee" table in "Object Explorer" window, and select "Design"
2. Change the datatype from nvarchar(50) to int
3. Save the table

At this point, you will get an error message - Saving changes is not permitted. The changes you have made require the following tables to be dropped and re-created. You have either made changes to a table that can't be re-created or enabled the option Prevent saving changes that require the table to be re-created.



Saving changes is not permitted. The changes you have made require the following tables to be dropped and re-created. You have either made changes to a table that can't be re-created or enabled the option Prevent saving changes that require the table to be re-created.

tblEmployee

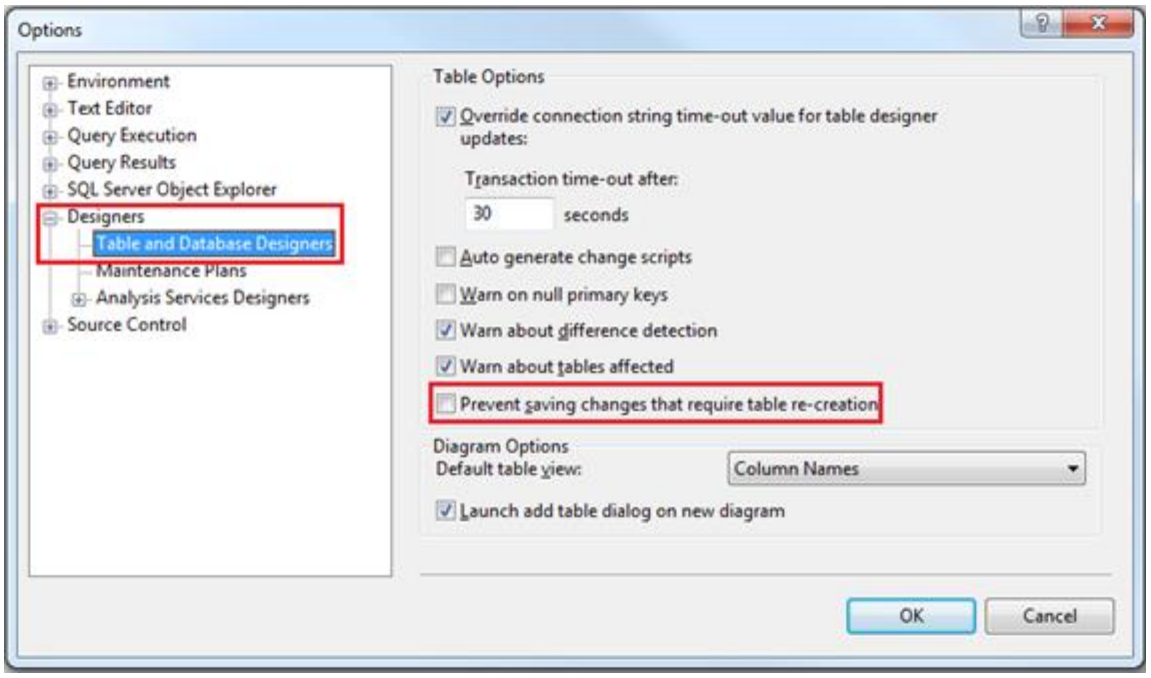
So, the obvious next question is, **how to alter the database table definition without the need to drop, re-create and again populate the table with data?**
There are 2 options

Option 1: Use a sql query to alter the column as shown below.

```
Alter table tblEmployee
Alter column Salary int
```

Option 2: Disable **"Prevent saving changes that require table re-creation"** option in sql server 2008
1. Open Microsoft SQL Server Management Studio 2008
2. Click Tools, select Options
3. Expand Designers, and select "Table and Database Designers"
4. On the right hand side window, uncheck, Prevent saving changes that require table re-creation

5. Click OK


Options

- Environment
- Text Editor
- Query Execution
- Query Results
- SQL Server Object Explorer
- Designers
  - Table and Database Designers
  - Maintenance Plans
  - Analysis Services Designers
- Source Control

Table Options

- [✓] Override connection string time-out value for table designer updates:
  - Transaction time-out after: 30 seconds
- [ ] Auto generate change scripts
- [ ] Warn on null primary keys
- [✓] Warn about difference detection
- [✓] Warn about tables affected
- [ ] Prevent saving changes that require table re-creation

Diagram Options

Default table view: Column Names

- [✓] Launch add table dialog on new diagram

Part 68 - Optional parameters in sql server stored procedures

Parameters of a sql server stored procedure can be made optional by specifying default values.

We wil be using table tblEmployee for this Demo.

```
CREATE TABLE tblEmployee
(
 Id int IDENTITY PRIMARY KEY,
 Name nvarchar(50),
 Email nvarchar(50),
 Age int,
 Gender nvarchar(50),
 HireDate date,
)


Insert into tblEmployee values
('Sara Nan','Sara.Nan@test.com',35,'Female','1999-04-04')
Insert into tblEmployee values
('James Histo','James.Histo@test.com',33,'Male','2008-07-13')
Insert into tblEmployee values
('Mary Jane','Mary.Jane@test.com',28,'Female','2005-11-11')
Insert into tblEmployee values
```

('Paul Sensit','Paul.Sensit@test.com',29,'Male','2007-10-23')

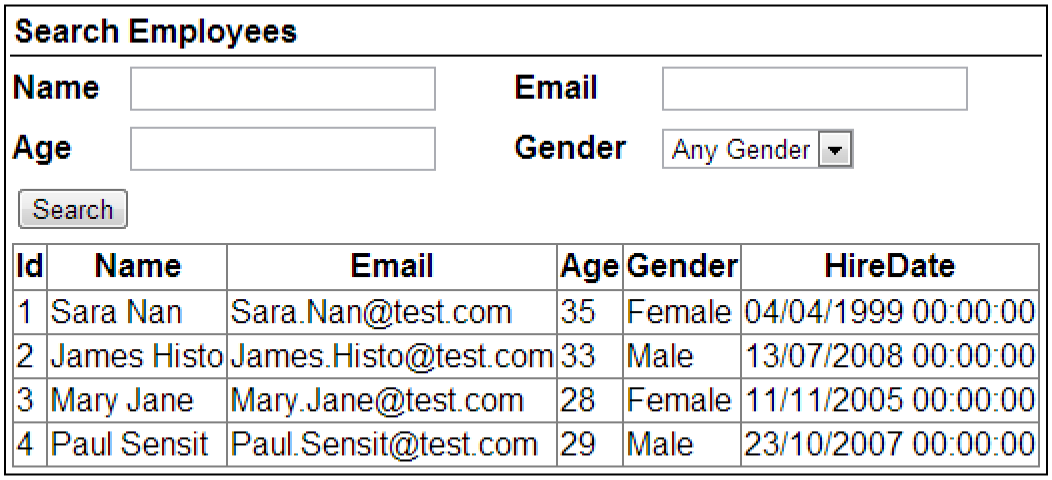
Name, Email, Age and Gender parameters of spSearchEmployees stored procedure are optional. Notice that, we have set defaults for all the parameters, and in the "WHERE" clause we are checking if the respective parameter IS NULL.

```
Create Proc spSearchEmployees
@Name nvarchar(50) = NULL,
@Email nvarchar(50) = NULL,
@Age int = NULL,
@Gender nvarchar(50) = NULL
as
Begin
 Select * from tblEmployee where
 (Name = @Name OR @Name IS NULL) AND
 (Email = @Email OR @Email IS NULL) AND
 (Age = @Age OR @Age IS NULL) AND
 (Gender = @Gender OR @Gender IS NULL)
End
```

Testing the stored procedure

1. Execute spSearchEmployees -- This command will return all the rows
2. Execute spSearchEmployees @Gender = 'Male' -- Retruns only Male employees
3. Execute spSearchEmployees @Gender = 'Male', @Age = 29 -- Retruns Male employees whose age is 29

This stored procedure can be used by a search page that looks as shown below.



Id	Name	Email	Age	Gender	HireDate
1	Sara Nan	Sara.Nan@test.com	35	Female	04/04/1999 00:00:00
2	James Histo	James.Histo@test.com	33	Male	13/07/2008 00:00:00
3	Mary Jane	Mary.Jane@test.com	28	Female	11/11/2005 00:00:00
4	Paul Sensit	Paul.Sensit@test.com	29	Male	23/10/2007 00:00:00

Part 69 - Merge in SQL Server

What is the use of MERGE statement in SQL Server

Merge statement introduced in SQL Server 2008 allows us to perform Inserts, Updates and Deletes in one statement. This means we no longer have to use multiple statements for performing Insert, Update and Delete.

With merge statement we require 2 tables
1. Source Table - Contains the changes that needs to be applied to the target table
2. Target Table - The table that require changes (Inserts, Updates and Deletes)

The merge statement joins the target table to the source table by using a common column in both the tables. Based on how the rows match up as a result of the join, we can then perform insert, update, and delete on the target table.

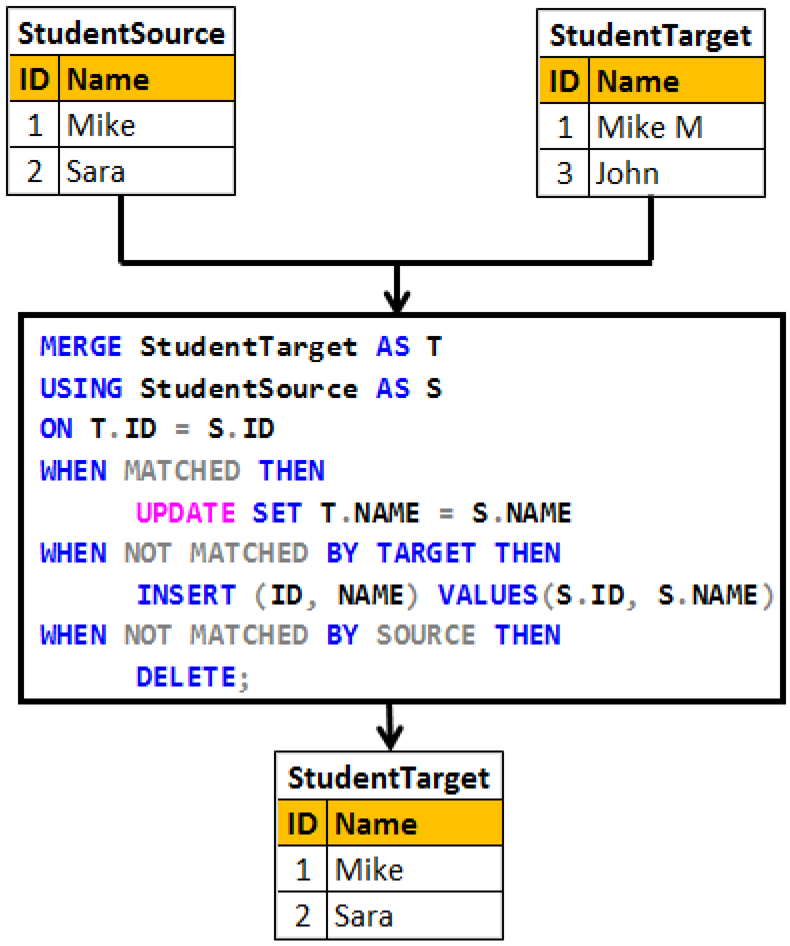
Merge statement syntax

```
MERGE [TARGET] AS T
USING [SOURCE] AS S
  ON [JOIN_CONDITIONS]
 WHEN MATCHED THEN
     [UPDATE STATEMENT]
 WHEN NOT MATCHED BY TARGET THEN
     [INSERT STATEMENT]
 WHEN NOT MATCHED BY SOURCE THEN
     [DELETE STATEMENT]
```

Example 1 : In the example below, INSERT, UPDATE and DELETE are all performed in one statement
1. When matching rows are found, StudentTarget table is UPDATED (i.e WHEN MATCHED)

2. When the rows are present in StudentSource table but not in StudentTarget table those rows are INSERTED into StudentTarget table (i.e WHEN NOT MATCHED BY TARGET)

3. When the rows are present in StudentTarget table but not in StudentSource table those rows are DELETED from StudentTarget table (i.e WHEN NOT MATCHED BY SOURCE)



StudentSource	
ID	Name
1	Mike
2	Sara

StudentTarget	
ID	Name
1	Mike M
3	John

```
MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
      UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
      INSERT (ID, NAME) VALUES(S.ID, S.NAME)
WHEN NOT MATCHED BY SOURCE THEN
      DELETE;
```

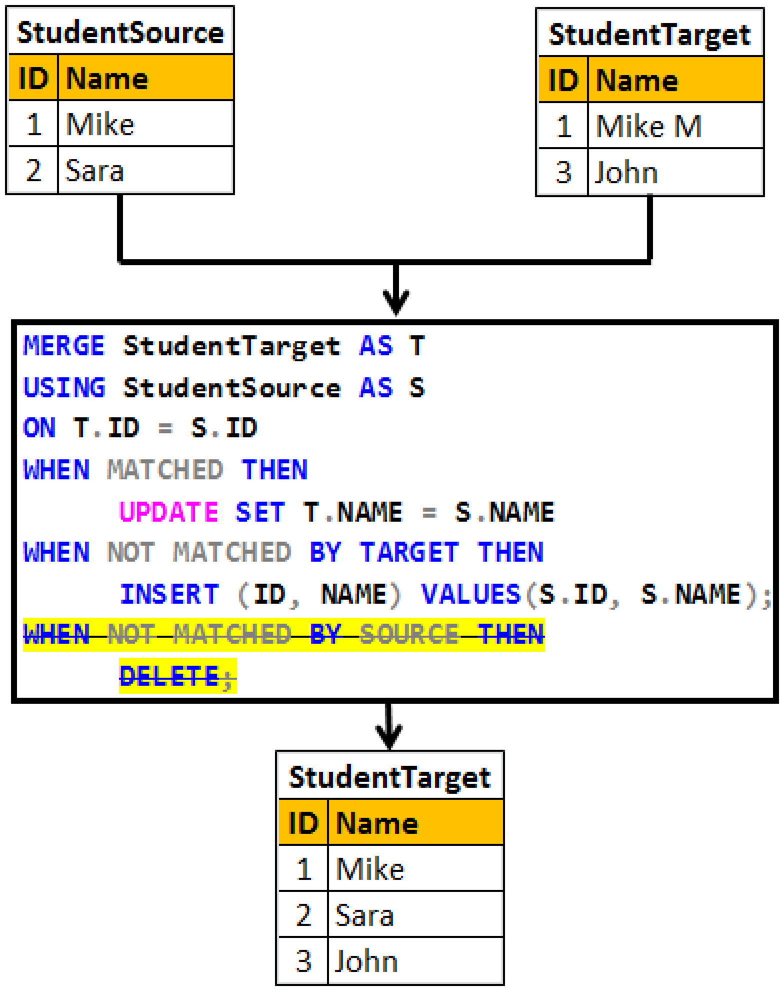
StudentTarget	
ID	Name
1	Mike
2	Sara

```
Create table StudentSource
(
    ID int primary key,
    Name nvarchar(20)
)
GO

Insert into StudentSource values (1, 'Mike')
Insert into StudentSource values (2, 'Sara')
GO

Create table StudentTarget
(
    ID int primary key,
    Name nvarchar(20)
)
GO

Insert into StudentTarget values (1, 'Mike M')
Insert into StudentTarget values (3, 'John')
GO

MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, NAME) VALUES(S.ID, S.NAME)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Please Note : Merge statement should end with a semicolon, otherwise you would get an error stating - A MERGE statement must be terminated by a semi-colon (;)

In real time we mostly perform INSERTS and UPDATES. The rows that are present in target table but not in source table are usually not deleted from the target table.

Example 2 : In the example below, only INSERT and UPDATE is performed. We are not deleting the rows that are present in the target table but not in the source table.



StudentSource	
ID	Name
1	Mike
2	Sara

StudentTarget	
ID	Name
1	Mike M
3	John

```
MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
      UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
      INSERT (ID, NAME) VALUES(S.ID, S.NAME);
```
~~WHEN NOT MATCHED BY SOURCE THEN~~
~~DELETE;~~

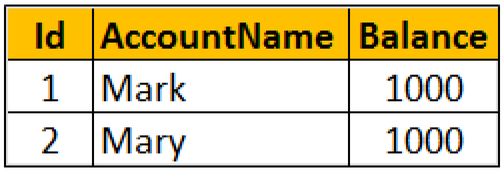
StudentTarget	
ID	Name
1	Mike
2	Sara
3	John

```
Truncate table StudentSource
Truncate table StudentTarget
GO

Insert into StudentSource values (1, 'Mike')
Insert into StudentSource values (2, 'Sara')
GO

Insert into StudentTarget values (1, 'Mike M')
Insert into StudentTarget values (3, 'John')
GO
MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, NAME) VALUES(S.ID, S.NAME);
```

Part 70-sql server concurrent transactions

In this video we will discuss
1. What a transaction is
2. The problems that might arise when tarnsactions are run concurrently
3. The different transaction isolation levels provided by SQL Server to address concurrency side effects

What is a transaction
A transaction is a group of commands that change the data stored in a database. A transaction, is treated as a single unit of work. A transaction ensures that, either all of the commands succeed, or none of them. If one of the commands in the transaction fails, all of the commands fail, and any data that was modified in the database is rolled back. In this way, transactions maintain the integrity of data in a database.



Id	AccountName	Balance
1	Mark	1000
2	Mary	1000

Example : The following transaction ensures that both the UPDATE statements succeed or both of them fail if there is a problem with one UPDATE statement.

```
-- Transfer $100 from Mark to Mary Account
BEGIN TRY
   BEGIN TRANSACTION
       UPDATE Accounts SET Balance = Balance - 100 WHERE Id = 1
       UPDATE Accounts SET Balance = Balance + 100 WHERE Id = 2
   COMMIT TRANSACTION
   PRINT 'Transaction Committed'
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
   PRINT 'Transaction Rolled back'
END CATCH
```

Databases are powerful systems and are potentially used by many users or applications at the same time. Allowing concurrent transactions is essential for performance but may introduce concurrency issues when two or more transactions are working with the same data at the same time.

Some of the common concurrency problems

- Dirty Reads
- Lost Updates

- Nonrepeatable Reads
- Phantom Reads

We will discuss what these problems are in detail with examples in our upcomning videos

One way to solve all these concurrency problems is by allowing only one user to execute, only one transaction at any point in time. Imagine what could happen if you have a large database with several users who want to execute several transactions. All the transactions get queued and they may have to wait a long time before they could get a chance to execute their transactions. So you are getting poor performance and the whole purpose of having a powerful database system is defeated if you serialize access this way.

At this point you might be thinking, for best performance let us allow all transactions to execute concurrently. The problem with this approach is that it may cause all sorts of concurrency problems (i.e Dirty Reads, Lost Updates, Nonrepeatable Reads, Phantom Reads) if two or more transactions work with the same data at the same time.

SQL Server provides different **transaction isolation levels**, to balance concurrency problems and performance depending on our application needs.

- Read Uncommitted
- Read Committed
- Repeatable Read
- Snapshot
- Serializable

The isolation level that you choose for your transaction, defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Depending on the isolation level you have chosen you get varying degrees of performance and concurrency problems. The table here has the list of isoltaion levels along with concurrency side effects.

Isolation Level	**Dirty Reads**	**Lost Update**	**Nonrepeatable 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

If you choose the lowest isolation level (i.e Read Uncommitted), it increases the number of concurrent transactions that can be executed at the same time, but the down side is you have all sorts of concurrency issues. On the other hand if you choose the highest isolation level (i.e Serializable), you will have no concurrency side effects, but the downside is that, this will reduce the number of concurrent transactions that can be executed at the same time if those transactions work with same data.

In our upcoming videos we will discuss the concurrency problems in detail with examples

Part 71-sql server dirty read example

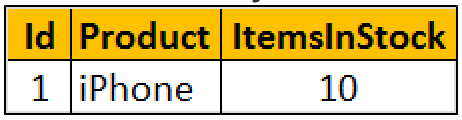
In this video we will discuss, **dirty read concurrency problem** with an example. This is continuation to Part 70. Please watch Part 70 from SQL Server tutorial for beginners.

A dirty read happens when one transaction is permitted to read data that has been modified by another transaction that has not yet been committed. In most cases this would not cause a problem. However, if the first transaction is rolled back after the second reads the data, the second transaction has dirty data that does not exist anymore.

SQL script to create table tblInventory

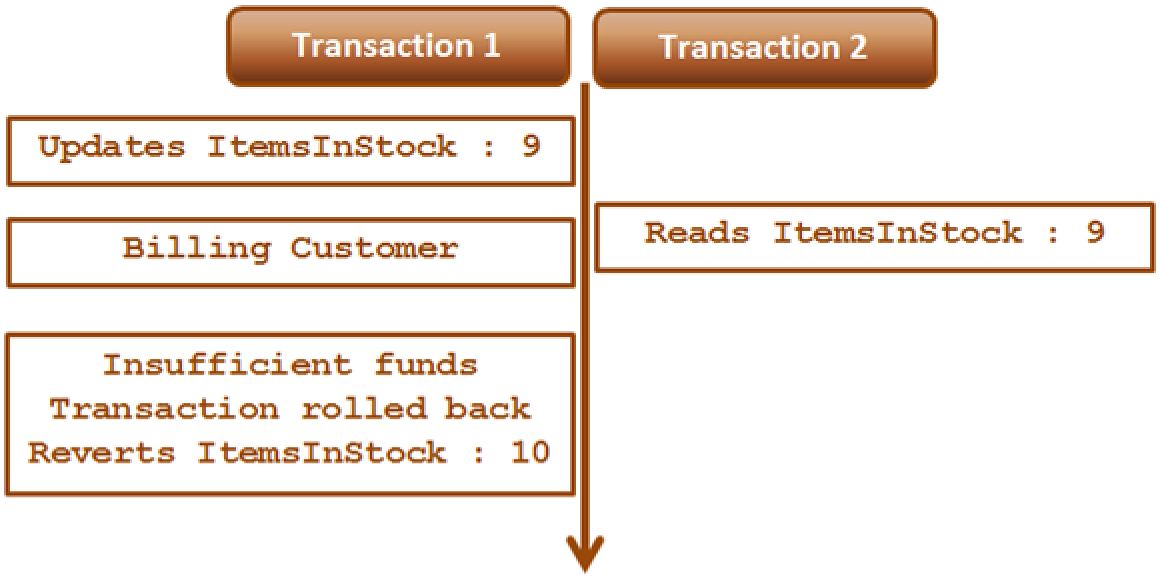
```
Create table tblInventory
(
    Id int identity primary key,

    Product nvarchar(100),

    ItemsInStock int

)

Go


Insert into tblInventory values ('iPhone', 10)
```

Table tblInventory



Id	Product	ItemsInStock
1	iPhone	10

Dirty Read Example : In the example below, Transaction 1, updates the value of ItemsInStock to 9. Then it starts to bill the customer. While Transaction 1 is still in progress, Transaction 2 starts and reads ItemsInStock value which is 9 at the moment. At this point, Transaction 1 fails because of insufficient funds and is rolled back. The ItemsInStock is reverted to the original value of 10, but Transaction 2 is working with a different value (i.e 10).



Transaction 1 :

```
Begin Tran

Update tblInventory set ItemsInStock = 9 where Id=1


-- Billing the customer

Waitfor Delay '00:00:15'

-- Insufficient Funds. Rollback transaction


Rollback Transaction
```

Transaction 2 :

```
Set Transaction Isolation Level Read Uncommitted

Select * from tblInventory where Id=1
```

Read Uncommitted transaction isolation level is the only isolation level that has dirty read side effect. This is the least restrictive of all the isolation levels. When this transaction isolation level is set, it is possible to read uncommitted or dirty data. Another option to read dirty data is by using NOLOCK table hint. The query below is equivalent to the query in Transaction 2.

```
Select * from tblInventory (NOLOCK) where Id=1
```

Part 72-sql server lost update problem

In this video we will discuss, **lost update problem in sql server** with an example

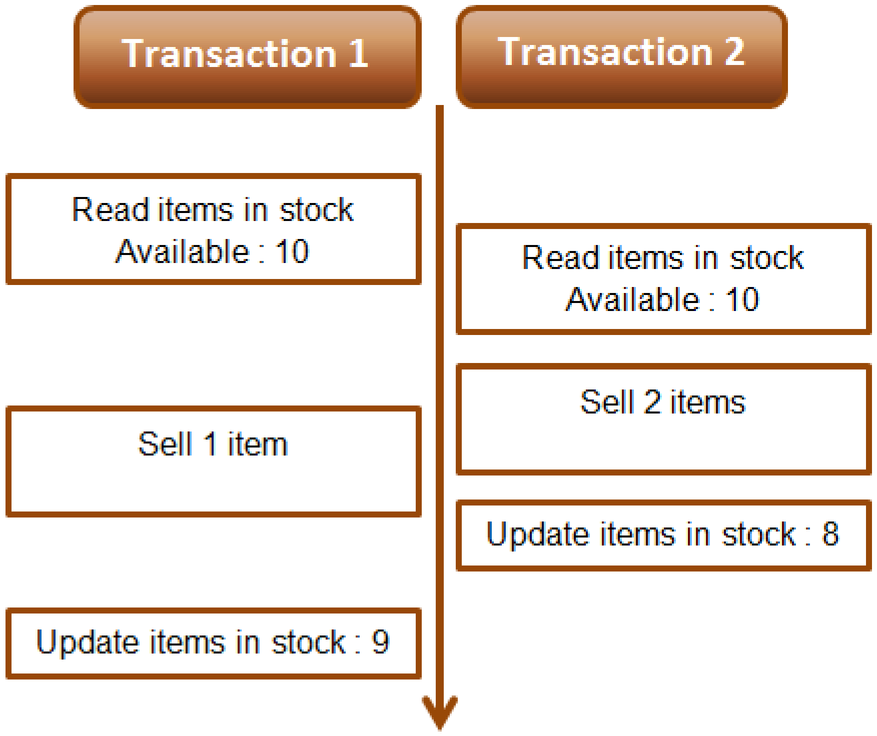
Lost update problem happens when 2 transactions read and update the same data. Let's understand this with an example. We will use the following table **tblInventory** for this example.

Id	Product	ItemsInStock
1	iPhone	10

As you can see in the diagram below there are 2 transactions - Transaction 1 and Transaction 2. Transaction 1 starts first, and it is processing an order for 1 iPhone. It sees ItemsInStock as 10.

At this time Transaction 2 is processing another order for 2 iPhones. It also sees ItemsInStock as 10. Transaction 2 makes the sale first and updates ItemsInStock with a value of 8.

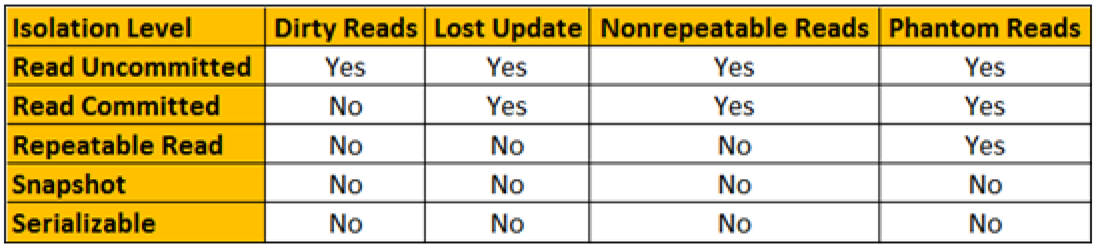
At this point Transaction 1 completes the sale and silently overwrites the update of Transaction 2. As Transaction 1 sold 1 iPhone it has updated ItemsInStock to 9, while it actually should have updated it to 7.


Transaction 1
Transaction 2
Read items in stock Available : 10
Read items in stock Available : 10
Sell 2 items
Update items in stock : 8
Sell 1 item
Update items in stock : 9

Example : The lost update problem example. Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code. Transaction 1 is processing an order for 1 iPhone, while Transaction 2 is processing an order for 2 iPhones. At the end of both the transactions ItemsInStock must be 7, but we have a value of 9. This is because Transaction 1 silently overwrites the update of Transaction 2. This is called the **lost update problem**.

```
-- Transaction 1
Begin Tran
Declare @ItemsInStock int

Select @ItemsInStock = ItemsInStock
from tblInventory where Id=1

-- Transaction takes 10 seconds
Waitfor Delay '00:00:10'
Set @ItemsInStock = @ItemsInStock - 1

Update tblInventory
Set ItemsInStock = @ItemsInStock where Id=1

Print @ItemsInStock

Commit Transaction


-- Transaction 2
Begin Tran
Declare @ItemsInStock int

Select @ItemsInStock = ItemsInStock
from tblInventory where Id=1

-- Transaction takes 1 second
Waitfor Delay '00:00:1'
Set @ItemsInStock = @ItemsInStock - 2

Update tblInventory
Set ItemsInStock = @ItemsInStock where Id=1

Print @ItemsInStock

Commit Transaction
```

Both Read Uncommitted and Read Committed transaction isolation levels have the lost update side effect. Repeatable Read, Snapshot, and Serializable isolation levels does not have this side effect. If you run the above Transactions using any of the higher isolation levels (Repeatable Read, Snapshot, or Serializable) you will not have lost update problem. The repeatable read isolation level uses additional locking on rows that are read by the current transaction, and prevents them from being updated or deleted elsewhere. This solves the lost update problem.



Isolation Level	Dirty Reads	Lost Update	Nonrepeatable 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

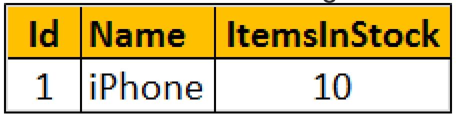
For both the above transactions, set Repeatable Read Isolation Level. Run Transaction 1 first and then a few seconds later run Transaction 2. Transaction 1 completes successfully, but Transaction 2 competes with the following error.
Transaction was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Once you rerun Transaction 2, ItemsInStock will be updated correctly as expected

Part 73-Non repeatable read example in sql server

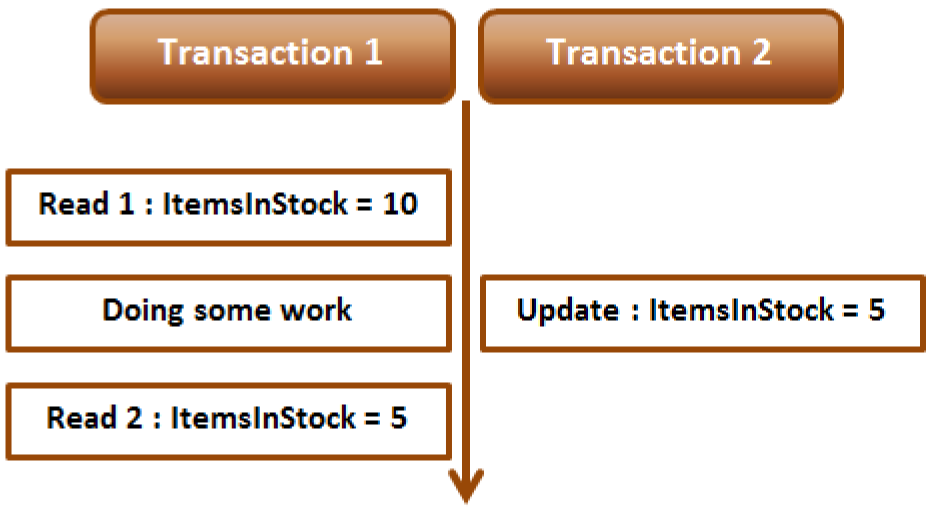
Non repeatable read problem happens when one transaction reads the same data twice and another transaction updates that data in between the first and second read of transaction one.

We will use the following table **tblInventory** in this demo



Id	Name	ItemsInStock
1	iPhone	10

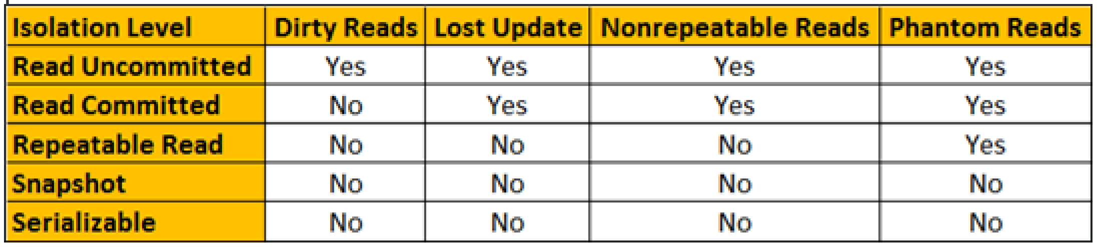
The following diagram explains the problem : Transaction 1 starts first. Reads ItemsInStock. Gets a value of 10 for first read. Transaction 1 is doing some work and at this point Transaction 2 starts and UpdatesItemsInStock to 5. Transaction 1 then makes a second read. At this point Transaction 1 gets a value of 5, reulting in non-repeatable read problem.


Transaction 1
Transaction 2
Read 1 : ItemsInStock = 10
Doing some work
Update : ItemsInStock = 5
Read 2 : ItemsInStock = 5

Non-repeatable read example : Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code. Notice that when Transaction 1 completes, it gets different values for read 1 and read 2, resulting in non-repeatable read.

```
-- Transaction 1
Begin Transaction
Select ItemsInStock from tblInventory where Id = 1

-- Do Some work
waitfor delay '00:00:10'

Select ItemsInStock from tblInventory where Id = 1
Commit Transaction

-- Transaction 2
Update tblInventory set ItemsInStock = 5 where Id = 1
```

Repeatable read or any other higher isolation level should solve the non-repeatable read problem.



Isolation Level	Dirty Reads	Lost Update	Nonrepeatable 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

Fixing non repeatable read concurrency problem : To fix the non-repeatable read problem, set transaction isolation level of Transaction 1 to repeatable read. 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 problem.

When you execute Transaction 1 and 2 from 2 different instances of SQL Server management studio, Transaction 2 is blocked until Transaction 1 completes and at the end of Transaction 1, both the reads get the same value for ItemsInStock.

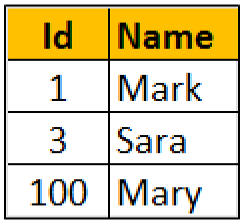
```
-- Transaction 1
Set transaction isolation level repeatable read
Begin Transaction
Select ItemsInStock from tblInventory where Id = 1

-- Do Some work
waitfor delay '00:00:10'

Select ItemsInStock from tblInventory where Id = 1
Commit Transaction

-- Transaction 2
Update tblInventory set ItemsInStock = 5 where Id = 1
```

Part 74-Phantom reads example in sql server

In this video we will discuss **phantom read concurrency problem** with examples.

Phantom read happens when one transaction executes a query twice and it gets a different number of rows in the result set each time. This happens when a second transaction inserts a new row that matches the WHERE clause of the query executed by the first transaction.

We will use the following table tblEmployees in this demo

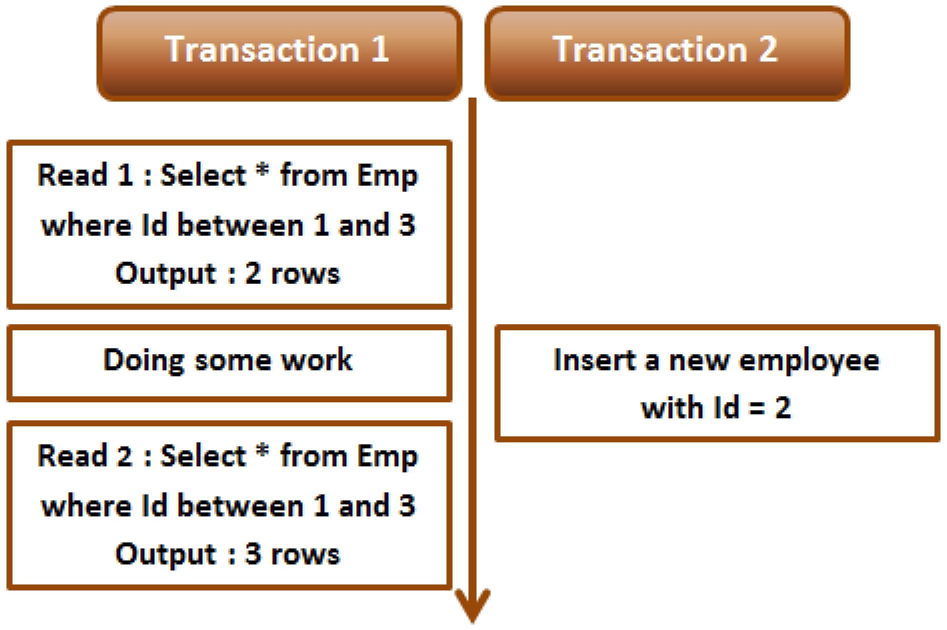


Id	Name
1	Mark
3	Sara
100	Mary

Scrip to create the table tblEmployees

```
Create table tblEmployees
(
   Id int primary key,
   Name nvarchar(50)
)
Go

Insert into tblEmployees values(1,'Mark')
Insert into tblEmployees values(3, 'Sara')
Insert into tblEmployees values(100, 'Mary')
```

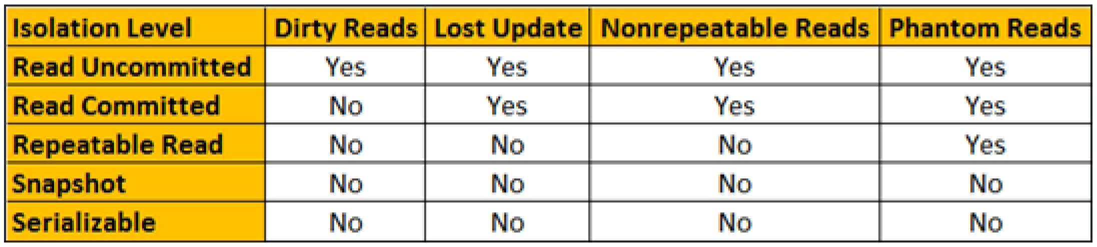
The following diagram explains the problem : Transaction 1 starts first. Reads from Emp table where Id between 1 and 3. 2 rows retrieved for first read. Transaction 1 is doing some work and at this point Transaction 2 starts and inserts a new employee with Id = 2. Transaction 1 then makes a second read. 3 rows retrieved for second read, reulting in phantom read problem.


Transaction 1
Transaction 2
Read 1 : Select * from Emp where Id between 1 and 3
Output : 2 rows
Doing some work
Insert a new employee with Id = 2
Read 2 : Select * from Emp where Id between 1 and 3
Output : 3 rows

Phantom read example : Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code. Notice that when Transaction 1 completes, it gets different number of rows for read 1 and read 2, resulting in phantom read.

```
-- Transaction 1
Begin Transaction
Select * from tblEmployees where Id between 1 and 3
-- Do Some work
waitfor delay '00:00:10'
Select * from tblEmployees where Id between 1 and 3
Commit Transaction

-- Transaction 2
Insert into tblEmployees values(2, 'Marcus')
```

Serializable or any other higher isolation level should solve the phantom read problem.



Isolation Level	Dirty Reads	Lost Update	Nonrepeatable 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

Fixing phantom read concurrency problem : To fix the phantom read problem, set transaction isolation level of Transaction 1 to serializable. This will place a range lock on the rows between 1 and 3, which prevents any other transaction from inserting new rows with in that range. This solves the phantom read problem.

When you execute Transaction 1 and 2 from 2 different instances of SQL Server management studio, Transaction 2 is blocked until Transaction 1 completes and at the end of Transaction 1, both the reads get the same number of rows.

```
-- Transaction 1
Set transaction isolation level serializable
Begin Transaction
Select * from tblEmployees where Id between 1 and 3
-- Do Some work
waitfor delay '00:00:10'
Select * from tblEmployees where Id between 1 and 3
Commit Transaction

-- Transaction 2

Insert into tblEmployees values(2, 'Marcus')
```

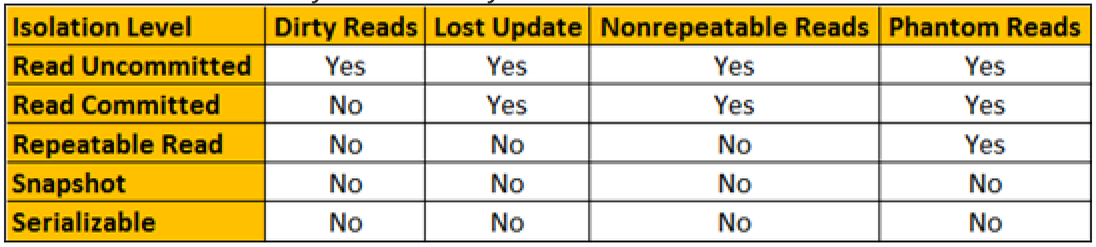
Difference between repeatable read and serializable
Repeatable read prevents only non-repeatable read. Repeatable read isolation level ensures that the data that one transaction has read, will be prevented from being updated or deleted by any other transaction, but it doe not prevent new rows from being inserted by other transactions resulting in phantom read concurrency problem.

Serializable prevents both non-repeatable read and phantom read problems. Serializable isolation level ensures that the data that one transaction has read, will be prevented from being updated or deleted by any other transaction. It also prevents new rows from being inserted by other transactions, so this isolation level prevents both non-repeatable read and phantom read problems

Part 75-Snapshot isolation level in sql server

In this video we will discuss, **snapshot isolation level in sql server** with examples.

As you can see from the table below, just like serializable isolation level, snapshot isolation level does not have any concurrency side effects.

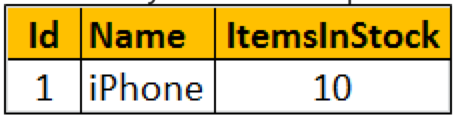


Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

What is the difference between serializable and snapshot isolation levels
Serializable isolation is implemented by acquiring locks which means the resources are locked for the duration of the current transaction. This isolation level does not have any concurrency side effects but at the cost of significant reduction in concurrency.

Snapshot isolation doesn't acquire locks, it maintains versioning in Tempdb. Since, snapshot isolation does not lock resources, it can significantly increase the number of concurrent transactions while providing the same level of data consistency as serializable isolation does.

Let us understand Snapshot isolation with an example. We will be using the following table tblInventory for this example.



Id	Name	ItemsInStock
1	iPhone	10

Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed.

```
--Transaction 1
Set transaction isolation level serializable
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
waitfor delay '00:00:10'
Commit Transaction
```

```
-- Transaction 2
Set transaction isolation level serializable
Select ItemsInStock from tblInventory where Id = 1
```

Now change the isolation level of Transaction 2 to snapshot. To set snapshot isolation level, it must first be enabled at the database level, and then set the transaction isolation level to snapshot.

```
-- Transaction 2
-- Enable snapshot isloation for the database
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
-- Set the transaction isolation level to snapshot
Set transaction isolation level snapshot
Select ItemsInStock from tblInventory where Id = 1
```

From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code. Notice that Transaction 2 is not blocked and returns the data from the database as it was before Transaction 1 has started.

Modifying data with snapshot isolation level : Now let's look at an example of what happens when a transaction that is using snapshot isolation tries to update the same data that another transaction is updating at the same time.

In the following example, Transaction 1 starts first and it is updating ItemsInStock to 5. At the same time, Transaction 2 that is using snapshot isolation level is also updating the same data. Notice that Transaction 2 is blocked until Transaction 1 completes. When Transaction 1 completes, Transaction 2 fails with the following error to prevent concurrency side effect - Lost update. If Transaction 2 was allowed to continue, it would have changed the ItemsInStock value to 8 and when Transaction 1 completes it overwrites ItemsInStock to 5, which means we have lost an update. To complete the work that Transaction 2 is doing we will have to rerun the transaction.

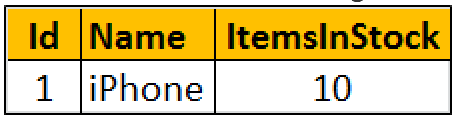
```
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot isolation to access table 'dbo.tblInventory' directly or indirectly in database 'SampleDB' to update, delete, or insert the row that has been modified or deleted by another transaction. Retry the transaction or change the isolation level for the update/delete statement.
```

```
--Transaction 1
Set transaction isolation level serializable
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
waitfor delay '00:00:10'
Commit Transaction

-- Transaction 2
-- Enable snapshot isloation for the database
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
-- Set the transaction isolation level to snapshot
Set transaction isolation level snapshot
Update tblInventory set ItemsInStock = 8 where Id = 1
```

Part 76-Read committed snapshot isolation level in sql server

In this video we will discuss **Read committed snapshot isolation level** in sql server. This is continuation Part 75. Please watch Part 75 from SQL Server tutorial before proceeding.

We will use the following table tblInventory in this demo



Id	Name	ItemsInStock
1	iPhone	10

Read committed snapshot isolation level is not a different isolation level. It is a different way of implementing Read committed isolation level. One problem we have with Read Committed isloation level is that, it blocks the transaction if it is trying to read the data, that another transaction is updating at the same time.

The following example demonstrates the above point. Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed.

```
--Transaction 1
Set transaction isolation level Read Committed

Begin Transaction

Update tblInventory set ItemsInStock = 5 where Id = 1

waitfor delay '00:00:10'

Commit Transaction
```

```
-- Transaction 2

Set transaction isolation level read committed

Begin Transaction

Select ItemsInStock from tblInventory where Id = 1

Commit Transaction
```

We can make Transaction 2 to use row versioning technique instead of locks by enabling Read committed snapshot isolation at the database level. Use the following command to enable READ_COMMITTED_SNAPSHOT isolation

```
Alter database SampleDB SET READ_COMMITTED_SNAPSHOT ON
```

Please note : For the above statement to execute successfully all the other database connections should be closed.

After enabling READ_COMMITTED_SNAPSHOT, execute Transaction 1 first and then

Transaction 2 simultaneously. Notice that the Transaction 2 is not blocked. It immediately returns the committed data that is in the database before Transaction 1 started. This is because Transaction 2 is now using Read committed snapshot isolation level.

Let's see if we can achieve the same thing using snapshot isolation level instead of read committed snapshot isolation level.

Step 1 : Turn off READ_COMMITTED_SNAPSHOT

```
Alter database SampleDB SET READ_COMMITTED_SNAPSHOT OFF
```

Step 2 : Enable snapshot isolation level at the database level

```
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
```

Step 3 : Execute Transaction 1 first and then Transaction 2 simultaneously. Just like in the previous example, notice that the Transaction 2 is not blocked. It immediately returns the committed data that is in the database before Transaction 1 started.

```
--Transaction 1
Set transaction isolation level Read Committed

Begin Transaction

Update tblInventory set ItemsInStock = 5 where Id = 1

waitfor delay '00:00:10'

Commit Transaction


-- Transaction 2

Set transaction isolation level snapshot

Begin Transaction

Select ItemsInStock from tblInventory where Id = 1

Commit Transaction
```

So what is the point in using read committed snapshot isolation level over snapshot isolation level?
There are some differences between read committed snapshot isolation level and snapshot isolation level. We will discuss these in our next video.

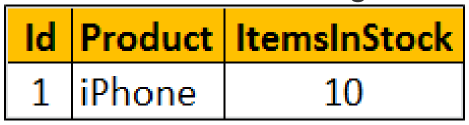
Part 77-Difference between snapshot isolation and read committed snapshot

In this video we will discuss the differences between snapshot isolation and read committed snapshot isolation in sql server. This is continuation to Parts 75 and 76. Please watch Part 75 and 76 from SQL Server tutorial before proceeding.

Read Committed Snapshot Isolation	Snapshot Isolation
No update conflicts	Vulnerable to update conflicts
Works with existing applications without requiring any change to the application	Application change may be required to use with an existing application
Can be used with distributed transactions	Cannot be used with distributed transactions
Provides statement-level read consistency	Provides transaction-level read consistency

Update conflicts : Snapshot isolation is vulnerable to update conflicts where as Read Committed Snapshot Isolation is not. When a transaction running under snapshot isolation triess to update data that an another transaction is already updating at the sametime, an update conflict occurs and the transaction terminates and rolls back with an error.

We will use the following table tblInventory in this demo



Id	Product	ItemsInStock
1	iPhone	10

Enable Snapshot Isolation for the SampleDB database using the following command

```
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
```

Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed. When Transaction 1 completes, Transaction 2 raises an update conflict and the transaction terminates and rolls back with an error.

```
--Transaction 1
Set transaction isolation level snapshot
```

```
Begin Transaction

Update tblInventory set ItemsInStock = 8 where Id = 1

waitfor delay '00:00:10'

Commit Transaction


-- Transaction 2

Set transaction isolation level snapshot

Begin Transaction

Update tblInventory set ItemsInStock = 5 where Id = 1

Commit Transaction
```

Now let's try the same thing using **Read Committed Sanpshot Isolation**

Step 1 : Disable Snapshot Isolation for the SampleDB database using the following command

```
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION OFF
```

Step 2 : Enable Read Committed Sanpshot Isolation at the database level using the following command

```
Alter database SampleDB SET READ_COMMITTED_SNAPSHOT ON
```

Step 3 : Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed. When Transaction 1 completes, Transaction 2 also completes successfully without any update conflict.

```
--Transaction 1
Set transaction isolation level read committed

Begin Transaction

Update tblInventory set ItemsInStock = 8 where Id = 1

waitfor delay '00:00:10'

Commit Transaction


-- Transaction 2
```

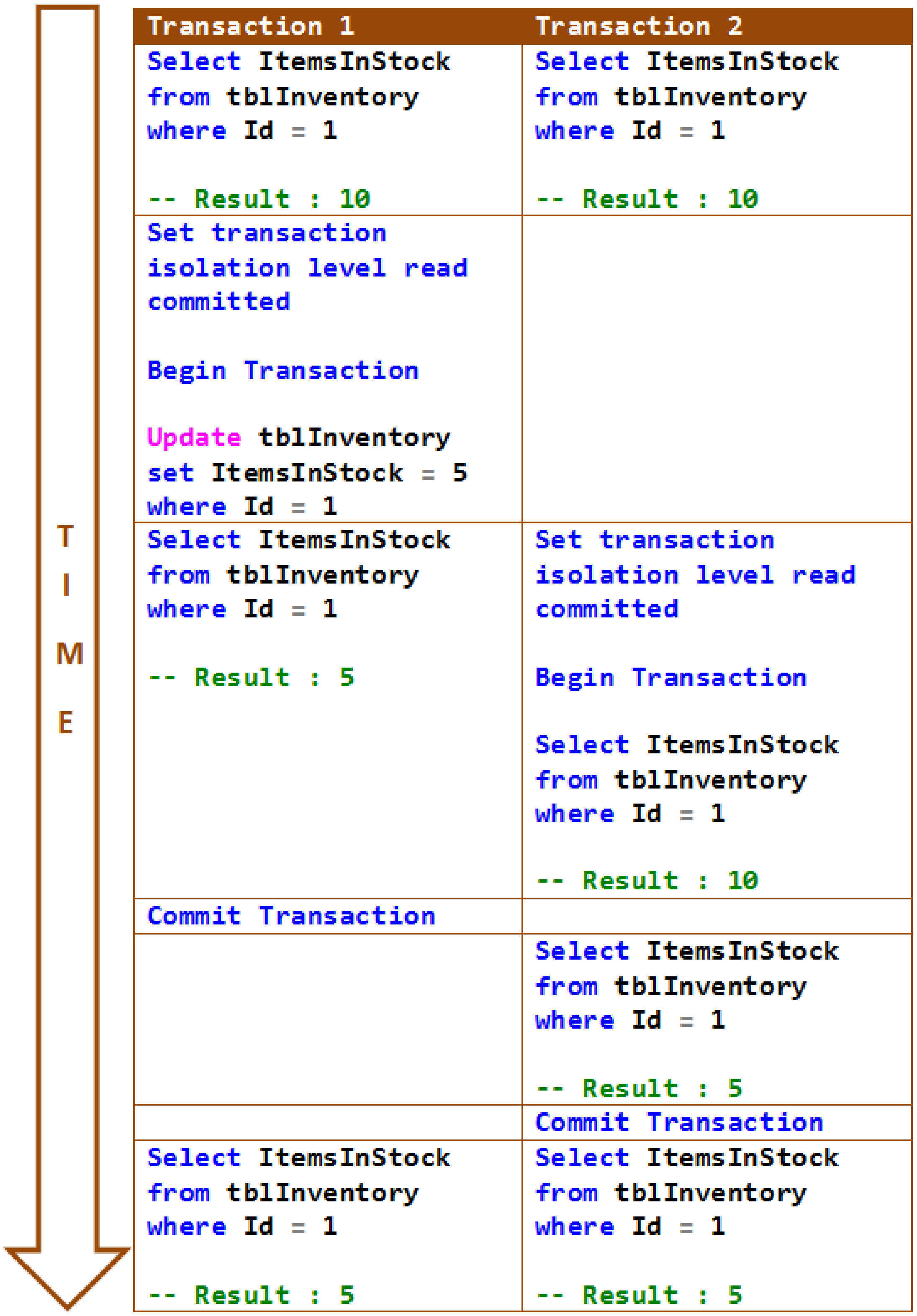
```
Set transaction isolation level read committed

Begin Transaction

Update tblInventory set ItemsInStock = 5 where Id = 1

Commit Transaction
```

Existing application : If your application is using the default Read Committed isolation level, you can very easily make the application to use Read Committed Snapshot Isolation without requiring any change to the application at all. All you need to do is turn on READ_COMMITTED_SNAPSHOT option in the database, which will change read committed isolation to use row versioning when reading the committed data.

Distributed transactions : Read Committed Snapshot Isolation works with distributed transactions, whereas snapshot isolation does not.

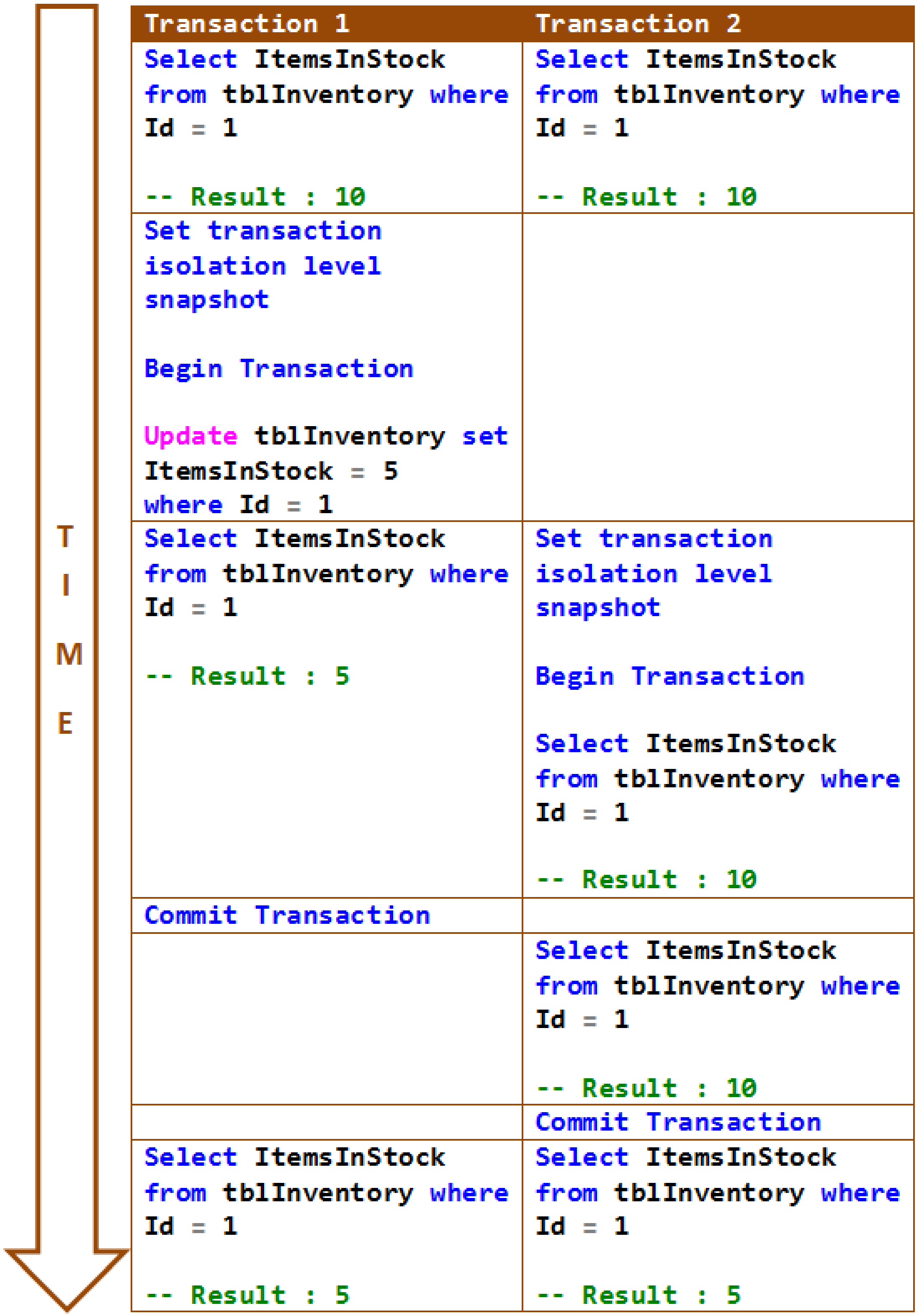
Read consistency : Read Committed Snapshot Isolation provides statement-level read consistency where as Snapshot Isolation provides transaction-level read consistency. The following diagrams explain this.

Transaction 2 has 2 select statements. Notice that both of these select statements return different data. This is because Read Committed Snapshot Isolation returns the last committed data before the select statement began and not the last committed data before the transaction began.



Transaction 1	Transaction 2
Select ItemsInStock from tblInventory where Id = 1 -- Result : 10	Select ItemsInStock from tblInventory where Id = 1 -- Result : 10
Set transaction isolation level read committed Begin Transaction Update tblInventory set ItemsInStock = 5 where Id = 1	
Select ItemsInStock from tblInventory where Id = 1 -- Result : 5	Set transaction isolation level read committed Begin Transaction Select ItemsInStock from tblInventory where Id = 1 -- Result : 10
Commit Transaction	
	Select ItemsInStock from tblInventory where Id = 1 -- Result : 5
	Commit Transaction
Select ItemsInStock from tblInventory where Id = 1 -- Result : 5	Select ItemsInStock from tblInventory where Id = 1 -- Result : 5

In the following example, both the select statements of Transaction 2 return same data. This is because Snapshot Isolation returns the last committed data before the transaction began and not the last committed data before the select statement began.



Transaction 1	Transaction 2

`Select ItemsInStock from tblInventory where Id = 1`
 	
`-- Result : 10`
	`Select ItemsInStock from tblInventory where Id = 1`
	
	`-- Result : 10`

`Set transaction isolation level snapshot`
 	
`Begin Transaction`
 	
`Update tblInventory set ItemsInStock = 5 where Id = 1`	

`Select ItemsInStock from tblInventory where Id = 1`
 	
`-- Result : 5`
	`Set transaction isolation level snapshot`
	
	`Begin Transaction`
	
	`Select ItemsInStock from tblInventory where Id = 1`
	
	`-- Result : 10`

`Commit Transaction`	

	`Select ItemsInStock from tblInventory where Id = 1`
	
	`-- Result : 10`

	`Commit Transaction`

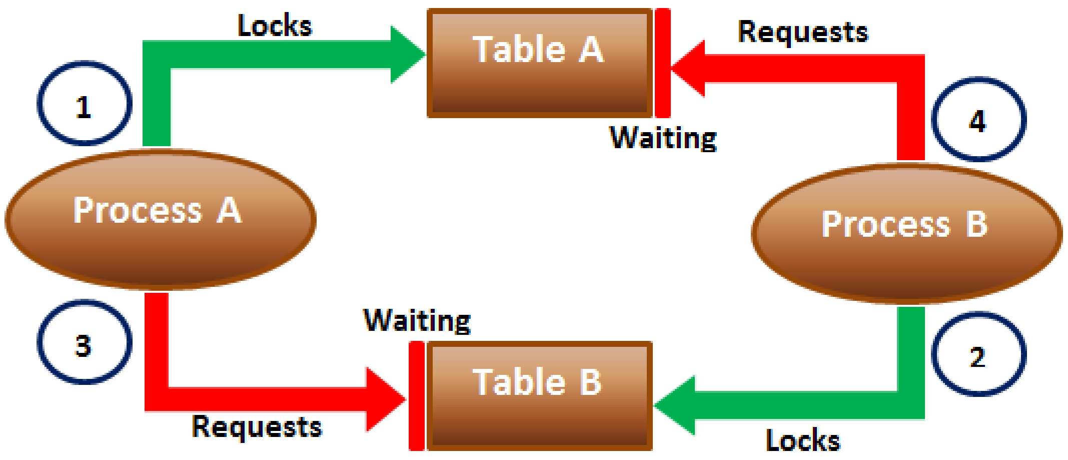
`Select ItemsInStock from tblInventory where Id = 1`
 	
`-- Result : 5`
	`Select ItemsInStock from tblInventory where Id = 1`
	
	`-- Result : 5`

Part 78-SQL Server deadlock example

In this video we will discuss a scenario when a deadlock can occur in SQL Server.

When can a deadlock occur
In a database, a deadlock occurs when two or more processes have a resource locked, and each process requests a lock on the resource that another process has already locked. Neither of the transactions here can move forward, as each one is waiting for the other to release the lock. The following diagram explains this.

Dead Lock Scenario in SQL Server



When deadlocks occur, SQL Server will choose one of processes as the deadlock victim and rollback that process, so the other process can move forward. The transaction that is chosen as the deadlock victim will produce the following error.
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 57) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Let us look at this in action. We will use the following 2 tables for this example.

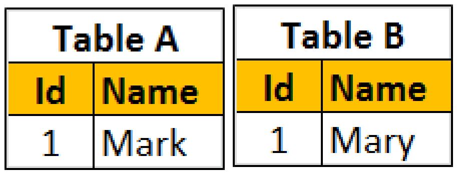


Table A

Id	Name
1	Mark

Table B

Id	Name
1	Mary

SQL script to create the tables and populate them with test data

```
Create table TableA
(
    Id int identity primary key,
    Name nvarchar(50)
)
```

```
Go

Insert into TableA values ('Mark')
Go

Create table TableB
(
    Id int identity primary key,
    Name nvarchar(50)
)
Go

Insert into TableB values ('Mary')

Go
```

The following 2 transactions will result in a dead lock. Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code.

```
-- Transaction 1
Begin Tran
Update TableA Set Name = 'Mark Transaction 1' where Id = 1

-- From Transaction 2 window execute the first update statement

Update TableB Set Name = 'Mary Transaction 1' where Id = 1

-- From Transaction 2 window execute the second update statement
Commit Transaction



-- Transaction 2
Begin Tran
Update TableB Set Name = 'Mark Transaction 2' where Id = 1

-- From Transaction 1 window execute the second update statement

Update TableA Set Name = 'Mary Transaction 2' where Id = 1

-- After a few seconds notice that one of the transactions complete
-- successfully while the other transaction is made the deadlock victim

Commit Transaction
```

Next Video : We will discuss the criteria SQL Server uses to choose a deadlock victim

Part 79-SQL Server deadlock victim selection

In this video we will discuss
1. How SQL Server detects deadlocks
2. What happens when a deadlock is detected
3. What is DEADLOCK_PRIORITY
4. What is the criteria that SQL Server uses to choose a deadlock victim when there is a deadlock

This is continuation to Part 78, please watch Part 78 before proceeding.

How SQL Server detects deadlocks
Lock monitor thread in SQL Server, runs every 5 seconds by default to detect if there are any deadlocks. If the lock monitor thread finds deadlocks, the deadlock detection interval will drop from 5 seconds to as low as 100 milliseconds depending on the frequency of deadlocks. If the lock monitor thread stops finding deadlocks, the Database Engine increases the intervals between searches to 5 seconds.

What happens when a deadlock is detected
When a deadlock is detected, the Database Engine ends the deadlock by choosing one of the threads as the deadlock victim. The deadlock victim's transaction is then rolled back and returns a 1205 error to the application. Rolling back the transaction of the deadlock victim releases all locks held by that transaction. This allows the other transactions to become unblocked and move forward.

What is DEADLOCK_PRIORITY
By default, SQL Server chooses a transaction as the deadlock victim that is least expensive to roll back. However, a user can specify the priority of sessions in a deadlock situation using the SET DEADLOCK_PRIORITY statement. The session with the lowest deadlock priority is chosen as the deadlock victim.

Example : SET DEADLOCK_PRIORITY NORMAL

DEADLOCK_PRIORITY
1. The default is Normal
2. Can be set to LOW, NORMAL, or HIGH
3. Can also be set to a integer value in the range of -10 to 10.
 LOW : -5
 NORMAL : 0
 HIGH : 5

What is the deadlock victim selection criteria
1. If the DEADLOCK_PRIORITY is different, the session with the lowest priority is selected

as the victim
2. If both the sessions have the same priority, the transaction that is least expensive to rollback is selected as the victim
3. If both the sessions have the same deadlock priority and the same cost, a victim is chosen randomly

SQL Script to setup the tables for the examples

```
Create table TableA
(
    Id int identity primary key,
    Name nvarchar(50)
)
Go

Insert into TableA values ('Mark')
Insert into TableA values ('Ben')
Insert into TableA values ('Todd')
Insert into TableA values ('Pam')
Insert into TableA values ('Sara')
Go

Create table TableB
(
    Id int identity primary key,
    Name nvarchar(50)
)
Go

Insert into TableB values ('Mary')
Go
```

Open 2 instances of SQL Server Management studio. From the first window execute

Transaction 1 code and from the second window execute Transaction 2 code. We have not explicitly set DEADLOCK_PRIORITY, so both the sessions have the default DEADLOCK_PRIORITY which is NORMAL. So in this case SQL Server is going to choose Transaction 2 as the deadlock victim as it is the least expensive one to rollback.

```
-- Transaction 1
Begin Tran

Update TableA Set Name = Name + ' Transaction 1' where Id IN (1, 2, 3, 4, 5)


-- From Transaction 2 window execute the first update statement


Update TableB Set Name = Name + ' Transaction 1' where Id = 1


-- From Transaction 2 window execute the second update statement

Commit Transaction



-- Transaction 2

Begin Tran

Update TableB Set Name = Name + ' Transaction 2' where Id = 1


-- From Transaction 1 window execute the second update statement


Update TableA Set Name = Name + ' Transaction 2' where Id IN (1, 2, 3, 4, 5)


-- After a few seconds notice that this transaction will be chosen as the deadlock

-- victim as it is less expensive to rollback this transaction than Transaction 1

Commit Transaction
```

In the following example we have set DEADLOCK_PRIORITY of Transaction 2 to HIGH. Transaction 1 will be chosen as the deadlock victim, because it's DEADLOCK_PRIORITY (Normal) is lower than the DEADLOCK_PRIORITY of Transaction 2.

```
-- Transaction 1
Begin Tran

Update TableA Set Name = Name + ' Transaction 1' where Id IN (1, 2, 3, 4, 5)


-- From Transaction 2 window execute the first update statement


Update TableB Set Name = Name + ' Transaction 1' where Id = 1


-- From Transaction 2 window execute the second update statement

Commit Transaction



-- Transaction 2

SET DEADLOCK_PRIORITY HIGH

GO

Begin Tran

Update TableB Set Name = Name + ' Transaction 2' where Id = 1


-- From Transaction 1 window execute the second update statement


Update TableA Set Name = Name + ' Transaction 2' where Id IN (1, 2, 3, 4, 5)


-- After a few seconds notice that Transaction 2 will be chosen as the

-- deadlock victim as it's DEADLOCK_PRIORITY (Normal) is lower than the

-- DEADLOCK_PRIORITY this transaction (HIGH)

Commit Transaction
```

Part80-Logging deadlocks in sql server

In this video we will discuss **how to write the deadlock information to the SQL Server error log**

When deadlocks occur, SQL Server chooses one of the transactions as the deadlock victim and rolls it back. There are several ways in SQL Server to track down the queries that are causing deadlocks. One of the options is to use SQL Server trace flag 1222 to write the deadlock information to the SQL Server error log.

Enable Trace flag : To enable trace flags use DBCC command. -1 parameter indicates that the trace flag must be set at the global level. If you omit -1 parameter the trace flag will be set only at the session level.

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

To check the status of the trace flag

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

To turn off the trace flag

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

The following SQL code generates a dead lock. This is the same code we discussed in Part 78 of SQL Server Tutorial.

```
--SQL script to create the tables and populate them with test data
Create table TableA

(

   Id int identity primary key,

   Name nvarchar(50)

)

Go


Insert into TableA values ('Mark')

Go


Create table TableB

(

   Id int identity primary key,

   Name nvarchar(50)
)
Go

Insert into TableB values ('Mary')
Go

--SQL Script to create stored procedures
Create procedure spTransaction1
as
Begin
   Begin Tran
   Update TableA Set Name = 'Mark Transaction 1' where Id = 1
   Waitfor delay '00:00:05'
   Update TableB Set Name = 'Mary Transaction 1' where Id = 1
   Commit Transaction
End

Create procedure spTransaction2
as
Begin
   Begin Tran
   Update TableB Set Name = 'Mark Transaction 2' where Id = 1
   Waitfor delay '00:00:05'
   Update TableA Set Name = 'Mary Transaction 2' where Id = 1
   Commit Transaction
End
```

Open 2 instances of SQL Server Management studio. From the first window execute **spTransaction1** and from the second window execute **spTransaction2**.

After a few seconds notice that one of the transactions complete successfully while the other transaction is made the deadlock victim and rollback.

The information about this deadlock should now have been logged in sql server error log.

To read the error log

```
execute sp_readerrorlog
```

Next video : How to read and understand the deadlock information that is logged in the sql server error log

Part 81-SQL Server deadlock analysis and prevention

In this video we will discuss **how to read and analyze sql server deadlock information captured in the error log**, so we can understand what's causing the deadlocks and take appropriate actions to prevent or minimize the occurrence of deadlocks. This is continuation to Part 80. Please watch Part 80 from SQL Server tutorial before proceeding.

The deadlock information in the error log has three sections

Section	Description
Deadlock Victim	Contains the ID of the process that was selected as the deadlock victim and killed by SQL Server.
Process List	Contains the list of the processes that participated in the deadlock.
Resource List	Contains the list of the resources (database objects) owned by the processes involved in the deadlock

Process List : The process list has lot of items. Here are some of them that are particularly useful in understanding what caused the deadlock.

Node	Description
loginname	The loginname associated with the process
isolationlevel	What isolation level is used
procname	The stored procedure name
Inputbuf	The code the process is executing when the deadlock occured

Resource List : Some of the items in the resource list that are particularly useful in understanding what caused the deadlock.

Node	Description
objectname	Fully qualified name of the resource involved in the deadlock
owner-list	Contains (owner id) the id of the owning process and the lock mode it has acquired on the resource. lock mode determines how the resource can be accessed by concurrent transactions. S for Shared lock, U for Update lock, X for Exclusive lock etc
waiter-list	Contains (waiter id) the id of the process that wants to acquire a lock on the resource and the lock mode it is requesting

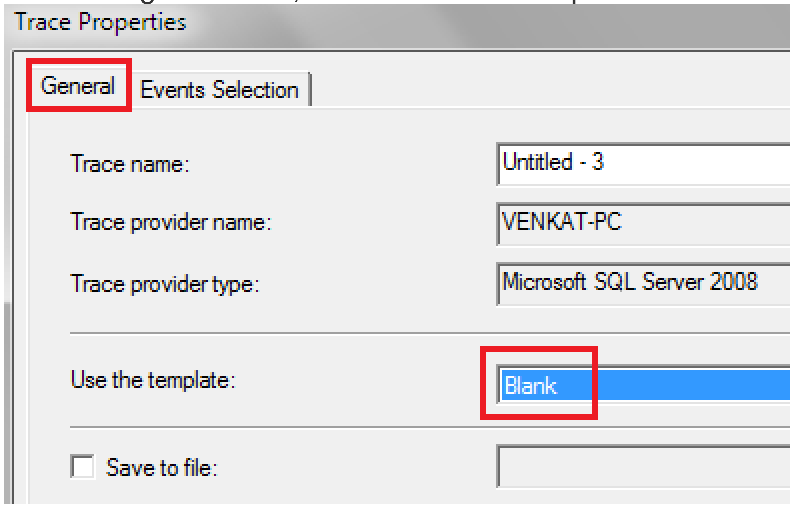
To prevent the deadlock that we have in our case, we need to ensure that database objects (Table A & Table B) are accessed in the same order every time

Part 82-Capturing deadlocks in sql profiler

In this video we will discuss **how to capture deadlock graph using SQL profiler.**

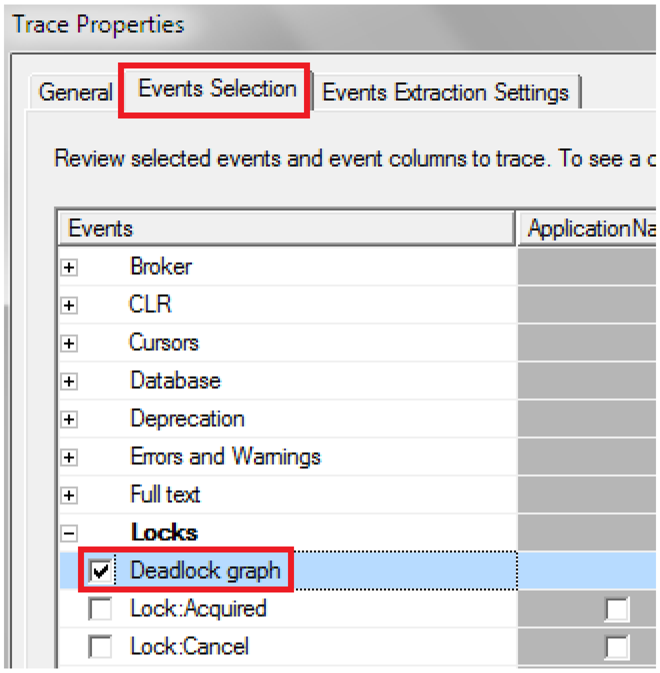
To capture deadlock graph, all you need to do is add Deadlock graph event to the trace in SQL profiler.

Here are the steps :
1. Open SQL Profiler
2. Click **File - New Trace**. Provide the credentials and connect to the server
3. On the general tab, select **"Blank"** template from **"Use the template"** dropdownlist



Trace name: Untitled - 3
Trace provider name: VENKAT-PC
Trace provider type: Microsoft SQL Server 2008
Use the template: Blank
Save to file:

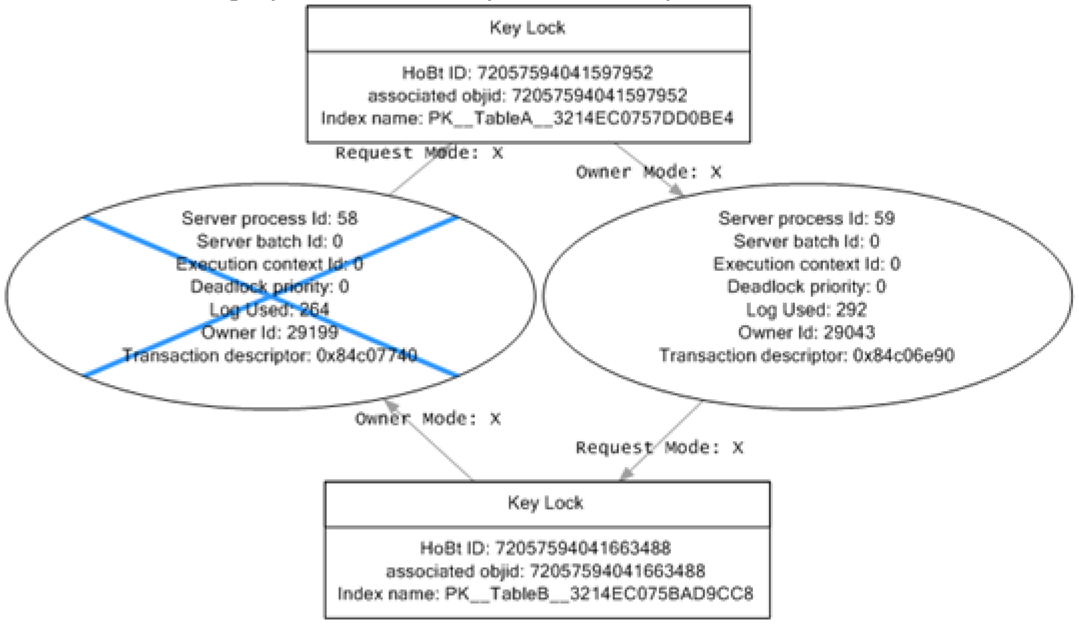
4. On the **"Events Selection"** tab, expand **"Locks"** section and select **"Deadlock graph"** event



Review selected events and event columns to trace. To see a c

Events	ApplicationNa
+ Broker	
+ CLR	
+ Cursors	
+ Database	
+ Deprecation	
+ Errors and Warnings	
+ Full text	
− **Locks**	
☑ Deadlock graph	
☐ Lock:Acquired	☐
☐ Lock:Cancel	☐

5. Finally click the **Run** button to start the trace
6. At this point execute the code that causes deadlock
7. The deadlock graph should be captured in the profiler as shown below.



The deadlock graph data is captured in XML format. If you want to extract this XML data to a physical file for later analysis, you can do so by following the steps below.
1. In SQL profiler, click on **"File - Export - Extract SQL Server Events - Extract Deadlock Events"**
2. Provide a name for the file
3. The extension for the deadlock xml file is **.xdl**
4. Finally choose if you want to export all events in a single file or each event in a separate

file

The deadlock information in the XML file is similar to what we have captured using the trace flag 1222.

Analyzing the deadlock graph
1. The oval on the graph, with the blue cross, represents the transaction that was chosen as the deadlock victim by SQL Server.
2. The oval on the graph represents the transaction that completed successfully.
3. When you move the mouse pointer over the oval, you can see the SQL code that was running that caused the deadlock.
4. The oval symbols represent the process nodes

- **Server Process Id :** If you are using SQL Server Management Studio you can see the server process id on information bar at the bottom.
- **Deadlock Priority :** If you have not set DEADLOCK PRIORITY explicitly using SET DEADLOCK PRIORITY statement, then both the processes should have the same default deadlock priority NORMAL (0).
- **Log Used :** The transaction log space used. If a transaction has used a lot of log space then the cost to roll it back is also more. So the transaction that has used the least log space is killed and rolled back.

5. The rectangles represent the resource nodes.

- **HoBt ID :** Heap Or Binary Tree ID. Using this ID query **sys.partitions** view to find the database objects involved in the deadlock.

```
SELECT object_name([object_id])

FROM sys.partitions

WHERE hobt_id = 72057594041663488
```

6. The arrows represent types of locks each process has on each resource node

Part 83-SQL Server deadlock error handling

In this video we will discuss **how to catch deadlock error using try/catch in SQL Server**.

Modify the stored procedure as shown below to catch the deadlock error. The code is commented and is self-explanatory.

```
Alter procedure spTransaction1
as
Begin
   Begin Tran
```

```
    Begin Try
        Update TableA Set Name = 'Mark Transaction 1' where Id = 1
        Waitfor delay '00:00:05'
        Update TableB Set Name = 'Mary Transaction 1' where Id = 1
        -- If both the update statements succeeded.
        -- No Deadlock occurred. So commit the transaction.
        Commit Transaction
        Select 'Transaction Successful'
    End Try
    Begin Catch
        -- Check if the error is deadlock error
        If(ERROR_NUMBER() = 1205)
        Begin
           Select 'Deadlock. Transaction failed. Please retry'
        End
        -- Rollback the transaction
        Rollback
    End Catch
End

Alter procedure spTransaction2
as
Begin
    Begin Tran
    Begin Try
        Update TableB Set Name = 'Mary Transaction 2' where Id = 1
        Waitfor delay '00:00:05'
        Update TableA Set Name = 'Mark Transaction 2' where Id = 1
        Commit Transaction
        Select 'Transaction Successful'
    End Try
    Begin Catch
        If(ERROR_NUMBER() = 1205)
        Begin
           Select 'Deadlock. Transaction failed. Please retry'
       End
        Rollback
    End Catch
End
```

After modifying the stored procedures, execute both the procedures from 2 different windows simultaneously. Notice that the deadlock error is handled by the catch block.

In our next video, we will discuss **how applications using ADO.NET can handle deadlock errors**.

Part 84-Handling deadlocks in ado.net

Part 85-Retry logic for deadlock exceptions

For these both part we need to know C#,ASP.net SQL Server and Jquery so I Skipped both part.

http://csharp-video-tutorials.blogspot.in/2015/08/retry-logic-for-deadlock-exceptions.html

Part 86-How to find blocking queries in sql server

In this video we will discuss, **how to find blocking queries in sql server**.

Blocking occurs if there are open transactions. Let us understand this with an example.

Execute the following 2 sql statements

```
Begin Tran
Update TableA set Name='Mark Transaction 1' where Id = 1
```

Now from a different window, execute any of the following commands. Notice that all the queries are blocked.

```
Select Count(*) from TableA

Delete from TableA where Id = 1

Truncate table TableA

Drop table TableA
```

This is because there is an open transaction. Once the open transaction completes, you will be able to execute the above queries.

So the obvious next question is - **How to identify all the active transactions**.

One way to do this is by using DBCC OpenTran. DBCC OpenTran will display only the oldest active transaction. It is not going to show you all the open transactions.

```
DBCC OpenTran
```

The following link has the SQL script that you can use to identify all the active transactions.
http://www.sqlskills.com/blogs/paul/script-open-transactions-with-text-and-plans

The beauty about this script is that it has a lot more useful information about the open transactions
Session Id
Login Name
Database Name
Transaction Begin Time
The actual query that is executed

You can now use this information and ask the respective developer to either commit or rollback the transactions that they have left open unintentionally.

For some reason if the person who initiated the transaction is not available, you also have the option to KILL the associated process. However, this may have unintended consequences, so use it with extreme caution.

There are 2 ways to kill the process are described below

Killing the process using SQL Server Activity Monitor :
1. Right Click on the Server Name in Object explorer and select **"Activity Monitor"**
2. In the **"Activity Monitor"** window expand Processes section
3. Right click on the associated **"Session ID"** and select **"Kill Process"** from the context menu

Killing the process using SQL command :
```
KILL Process_ID
```

What happens when you kill a session
All the work that the transaction has done will be rolled back. The database must be put back in the state it was in, before the transaction started.

Part 87-SQL Server except operator

In this video we will discuss **SQL Server except operator with examples.**

EXCEPT operator returns unique rows from the left query that aren’t in the right query’s results.

- Introduced in SQL Server 2005
- The number and the order of the columns must be the same in all queries
- The data types must be same or compatible
- This is similar to minus operator in oracle

Let us understand this with an example. We will use the following 2 tables for this example.

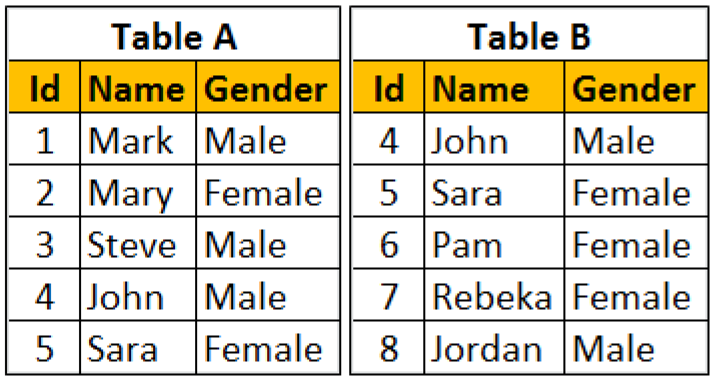


Table A		
Id	**Name**	**Gender**
1	Mark	Male
2	Mary	Female
3	Steve	Male
4	John	Male
5	Sara	Female

Table B		
Id	**Name**	**Gender**
4	John	Male
5	Sara	Female
6	Pam	Female
7	Rebeka	Female
8	Jordan	Male

SQL Script to create the tables

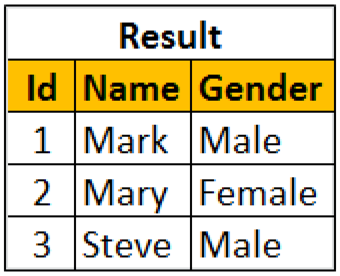
```
Create Table TableA
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go

Insert into TableA values (1, 'Mark', 'Male')
Insert into TableA values (2, 'Mary', 'Female')
Insert into TableA values (3, 'Steve', 'Male')
Insert into TableA values (4, 'John', 'Male')
Insert into TableA values (5, 'Sara', 'Female')
Go

Create Table TableB
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go

Insert into TableB values (4, 'John', 'Male')
Insert into TableB values (5, 'Sara', 'Female')
Insert into TableB values (6, 'Pam', 'Female')
Insert into TableB values (7, 'Rebeka', 'Female')
Insert into TableB values (8, 'Jordan', 'Male')
Go
```

Notice that the following query returns the unique rows from the left query that aren’t in the right query’s results.

```
Select Id, Name, Gender
From TableA
Except
Select Id, Name, Gender
```

```
From TableB
```

Result :

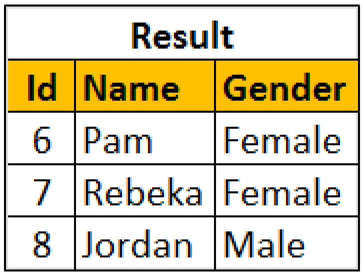


Result		
Id	**Name**	**Gender**
1	Mark	Male
2	Mary	Female
3	Steve	Male

To retrieve all of the rows from Table B that does not exist in Table A, reverse the two queries as shown below.

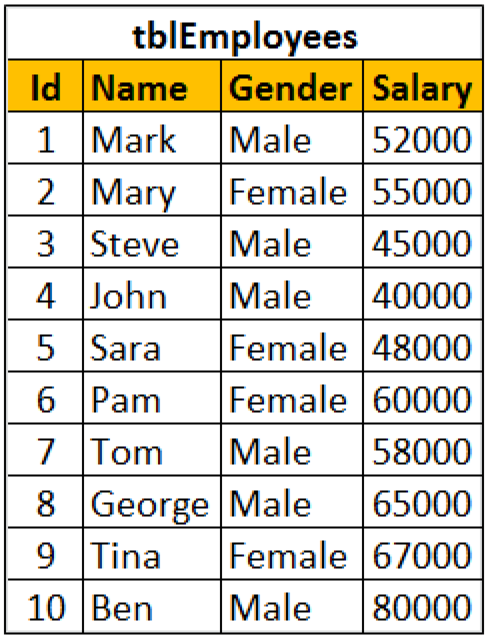
```
Select Id, Name, Gender
From TableB
Except
Select Id, Name, Gender
From TableA
```

Result :



Result		
Id	**Name**	**Gender**
6	Pam	Female
7	Rebeka	Female
8	Jordan	Male

You can also use Except operator on a single table. Let's use the following tblEmployees table for this example.

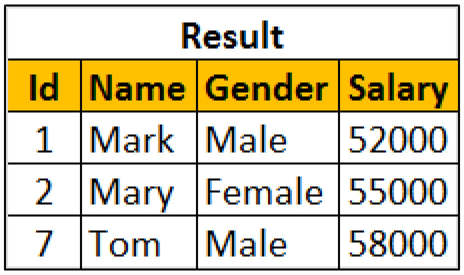


tblEmployees			
Id	Name	Gender	Salary
1	Mark	Male	52000
2	Mary	Female	55000
3	Steve	Male	45000
4	John	Male	40000
5	Sara	Female	48000
6	Pam	Female	60000
7	Tom	Male	58000
8	George	Male	65000
9	Tina	Female	67000
10	Ben	Male	80000

SQL script to create tblEmployees table

```
Create table tblEmployees
(
   Id int identity primary key,
   Name nvarchar(100),
   Gender nvarchar(10),
   Salary int
)
Go

Insert into tblEmployees values ('Mark', 'Male', 52000)
Insert into tblEmployees values ('Mary', 'Female', 55000)
Insert into tblEmployees values ('Steve', 'Male', 45000)
Insert into tblEmployees values ('John', 'Male', 40000)
Insert into tblEmployees values ('Sara', 'Female', 48000)
Insert into tblEmployees values ('Pam', 'Female', 60000)
Insert into tblEmployees values ('Tom', 'Male', 58000)
Insert into tblEmployees values ('George', 'Male', 65000)
Insert into tblEmployees values ('Tina', 'Female', 67000)
Insert into tblEmployees values ('Ben', 'Male', 80000)
Go
```

Result :



Result			
Id	Name	Gender	Salary
1	Mark	Male	52000
2	Mary	Female	55000
7	Tom	Male	58000

Order By clause should be used only once after the right query

```
Select Id, Name, Gender, Salary
From tblEmployees
Where Salary >= 50000
Except
Select Id, Name, Gender, Salary
From tblEmployees
Where Salary >= 60000
order By Name
```

Part 88-Difference between except and not in sql server

In this video we will discuss the **difference between EXCEPT and NOT IN operators in SQL Server**.

We will use the following 2 tables for this example.

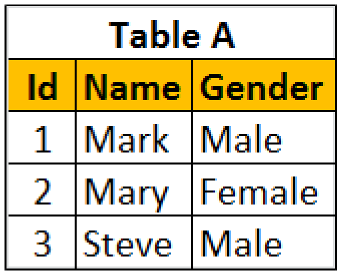


Table A		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male

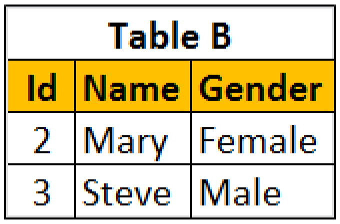
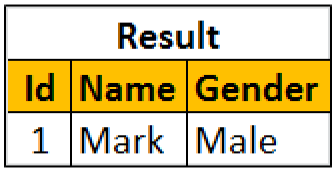


Table B		
Id	Name	Gender
2	Mary	Female
3	Steve	Male

The following query returns the rows from the left query that aren't in the right query's results.

```
Select Id, Name, Gender From TableA
Except
Select Id, Name, Gender From TableB
```

Result :



Result		
Id	Name	Gender
1	Mark	Male

The same result can also be achieved using NOT IN operator.

```
Select Id, Name, Gender From TableA
Where Id NOT IN (Select Id from TableB)
```

So, what is the difference between EXCEPT and NOT IN operators

1. Except filters duplicates and returns only DISTINCT rows from the left query that aren't in the right query's results, where as NOT IN does not filter the duplicates.

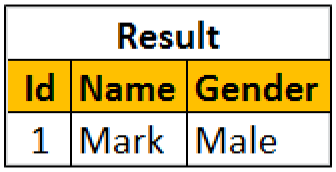
Insert the following row into TableA

```
Insert into TableA values (1, 'Mark', 'Male')
```

Now execute the following EXCEPT query. Notice that we get only the DISTINCT rows

```
Select Id, Name, Gender From TableA
Except
Select Id, Name, Gender From TableB
```

Result:

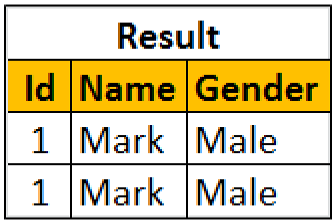


Result		
Id	Name	Gender
1	Mark	Male

Now execute the following query. Notice that the duplicate rows are not filtered.

```
Select Id, Name, Gender From TableA
Where Id NOT IN (Select Id from TableB)
```

Result:



Result		
Id	Name	Gender
1	Mark	Male
1	Mark	Male

2. EXCEPT operator expects the same number of columns in both the queries, where as NOT IN, compares a single column from the outer query with a single column from the subquery.

In the following example, the number of columns are different.

```
Select Id, Name, Gender From TableA
Except
Select Id, Name From TableB
```

The above query would produce the following error.

```
Msg 205, Level 16, State 1, Line 1
All queries combined using a UNION, INTERSECT or EXCEPT operator must have an
equal number of expressions in their target lists.
```

NOT IN, compares a single column from the outer query with a single column from subquery.

In the following example, the subquery returns multiple columns

```
Select Id, Name, Gender From TableA
Where Id NOT IN (Select Id, Name from TableB)
```

```
Msg 116, Level 16, State 1, Line 2
Only one expression can be specified in the select list when the subquery is not introduced with EXISTS
```

Part 89-Intersect operator in sql server

In this video we will discuss
1. Intersect operator in sql server
2. Difference between intersect and inner join

Intersect operator retrieves the common records from both the left and the right query of the Intersect operator.

- Introduced in SQL Server 2005
- The number and the order of the columns must be same in both the queries
- The data types must be same or at least compatible

Let us understand INTERSECT operator with an example.

We will use the following 2 tables for this example.

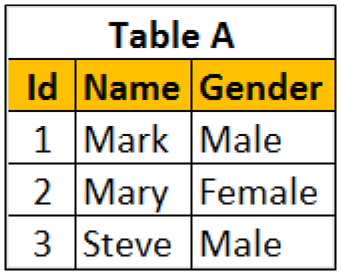


Table A

Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male

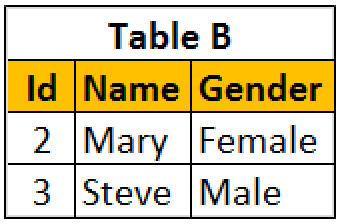


Table B

Id	Name	Gender
2	Mary	Female
3	Steve	Male

SQL Script to create the tables and populate with test data

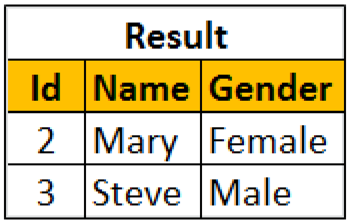
```
Create Table TableA
(
   Id int,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go
```

```
Insert into TableA values (1, 'Mark', 'Male')
Insert into TableA values (2, 'Mary', 'Female')
Insert into TableA values (3, 'Steve', 'Male')
Go

Create Table TableB
(
    Id int,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go

Insert into TableB values (2, 'Mary', 'Female')
Insert into TableB values (3, 'Steve', 'Male')
Go
```

The following query retrieves the common records from both the left and the right query of the Intersect operator.

```
Select Id, Name, Gender from TableA
Intersect
Select Id, Name, Gender from TableB
```

Result :



Result		
Id	Name	Gender
2	Mary	Female
3	Steve	Male

We can also achieve the same thinkg using INNER join. The following INNER join query would produce the exact same result.

```
Select TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

What is the difference between INTERSECT and INNER JOIN
1. INTERSECT filters duplicates and returns only DISTINCT rows that are common between the LEFT and Right Query, where as INNER JOIN does not filter the duplicates.

To understand this difference, insert the following row into TableA

```
Insert into TableA values (2, 'Mary', 'Female')
```

Now execute the following INTERSECT query. Notice that we get only the DISTINCT rows

```
Select Id, Name, Gender from TableA
Intersect
Select Id, Name, Gender from TableB
```

Result :

Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male

Now execute the following INNER JOIN query. Notice that the duplicate rows are not filtered.

```
Select TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

Result :

Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male
2	Mary	Female

You can make the INNER JOIN behave like INTERSECT operator by using the DISTINCT operator

```
Select DISTINCT TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

Result :

Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male

2. INNER JOIN treats two NULLS as two different values. So if you are joining two tables based on a nullable column and if both tables have NULLs in that joining column then, INNER JOIN will not include those rows in the result-set, where as INTERSECT treats two NULLs as a same value and it returns all matching rows.

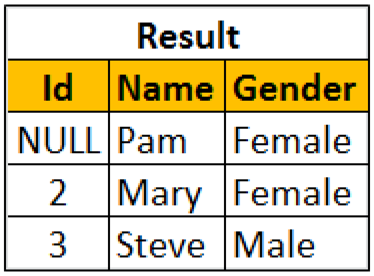
To understand this difference, execute the following 2 insert statements

```
Insert into TableA values(NULL, 'Pam', 'Female')
Insert into TableB values(NULL, 'Pam', 'Female')
```

INTERSECT query

```
Select Id, Name, Gender from TableA
Intersect
Select Id, Name, Gender from TableB
```

Result :

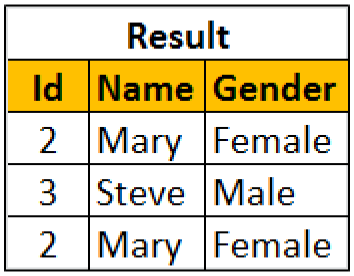


Result		
Id	**Name**	**Gender**
NULL	Pam	Female
2	Mary	Female
3	Steve	Male

INNER JOIN query

```
Select TableA.Id, TableA.Name, TableA.Gender
From TableA Inner Join TableB
On TableA.Id = TableB.Id
```

Result :

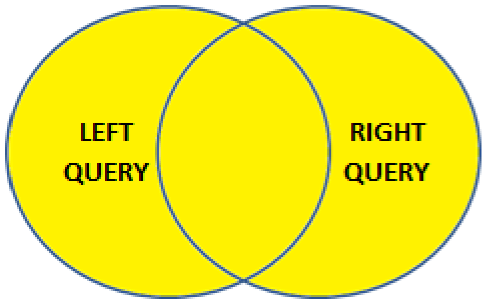


Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male
2	Mary	Female

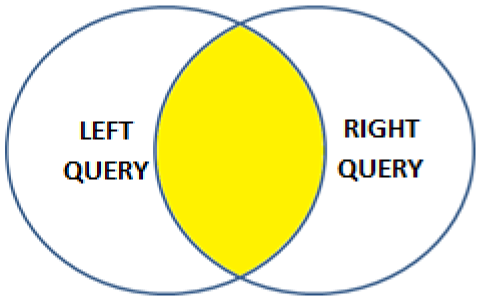
Part 90-Difference between union intersect and except in sql server

In this video we will discuss the **difference between union intersect and except in sql server with examples**.

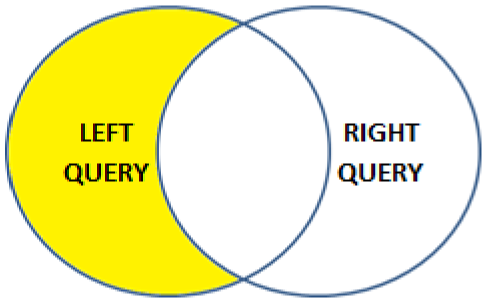
The following diagram explains the difference graphically



UNION operator returns all the unique rows from both the left and the right query. UNION ALL includes the duplicates as well



INTERSECT operator retrieves the common unique rows from both the left and the right query



EXCEPT operator returns unique rows from the left query that aren’t in the right query’s results

UNION operator returns all the unique rows from both the left and the right query. UNION ALL included the duplicates as well.

INTERSECT operator retrieves the common unique rows from both the left and the right query.

EXCEPT operator returns unique rows from the left query that aren’t in the right query’s results.

Let us understand these differences with examples. We will use the following 2 tables for the examples.

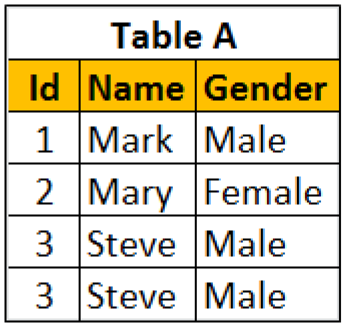


Table A		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
3	Steve	Male

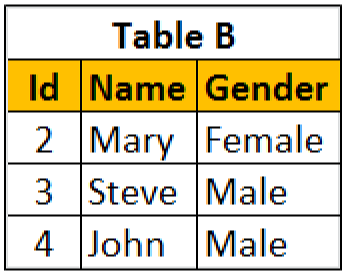


Table B		
Id	Name	Gender
2	Mary	Female
3	Steve	Male
4	John	Male

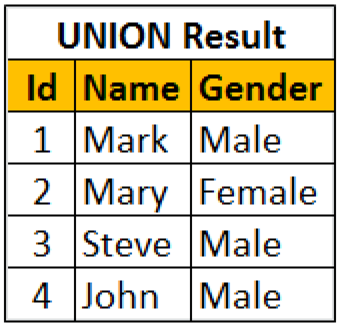
SQL Script to create the tables

```
Create Table TableA
(
    Id int,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go

Insert into TableA values (1, 'Mark', 'Male')
Insert into TableA values (2, 'Mary', 'Female')
Insert into TableA values (3, 'Steve', 'Male')
Insert into TableA values (3, 'Steve', 'Male')
Go

Create Table TableB
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go

Insert into TableB values (2, 'Mary', 'Female')
Insert into TableB values (3, 'Steve', 'Male')
Insert into TableB values (4, 'John', 'Male')
Go
```

UNION operator returns all the unique rows from both the queries. Notice the duplicates are removed.

```
Select Id, Name, Gender from TableA
UNION
Select Id, Name, Gender from TableB
```

Result :



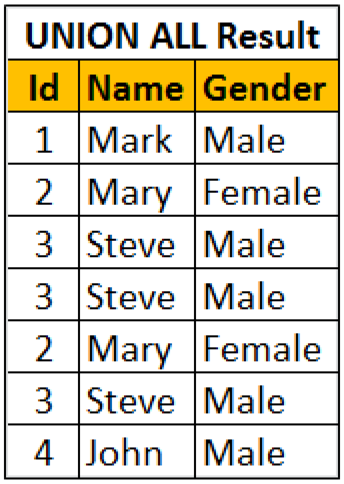
UNION Result

Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
4	John	Male

UNION ALL operator returns all the rows from both the queries, including the duplicates.

```
Select Id, Name, Gender from TableA
UNION ALL
Select Id, Name, Gender from TableB
```

Result :



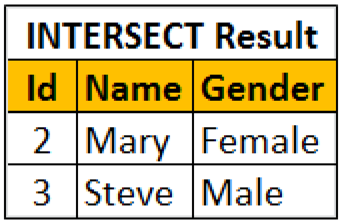
UNION ALL Result

Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
3	Steve	Male
2	Mary	Female
3	Steve	Male
4	John	Male

INTERSECT operator retrieves the common unique rows from both the left and the right query. Notice the duplicates are removed.

```
Select Id, Name, Gender from TableA
INTERSECT
Select Id, Name, Gender from TableB
```

Result :



INTERSECT Result

Id	Name	Gender
2	Mary	Female
3	Steve	Male

EXCEPT operator returns unique rows from the left query that aren’t in the right query’s results.

```
Select Id, Name, Gender from TableA
EXCEPT
Select Id, Name, Gender from TableB
```

Result :

EXCEPT Result		
Id	**Name**	**Gender**
1	Mark	Male

If you wnat the rows that are present in Table B but not in Table A, reverse the queries.

```
Select Id, Name, Gender from TableB
EXCEPT
Select Id, Name, Gender from TableA
```

Result :

EXCEPT Result		
Id	**Name**	**Gender**
4	John	Male

For all these 3 operators to work the following 2 conditions must be met

- The number and the order of the columns must be same in both the queries
- The data types must be same or at least compatible

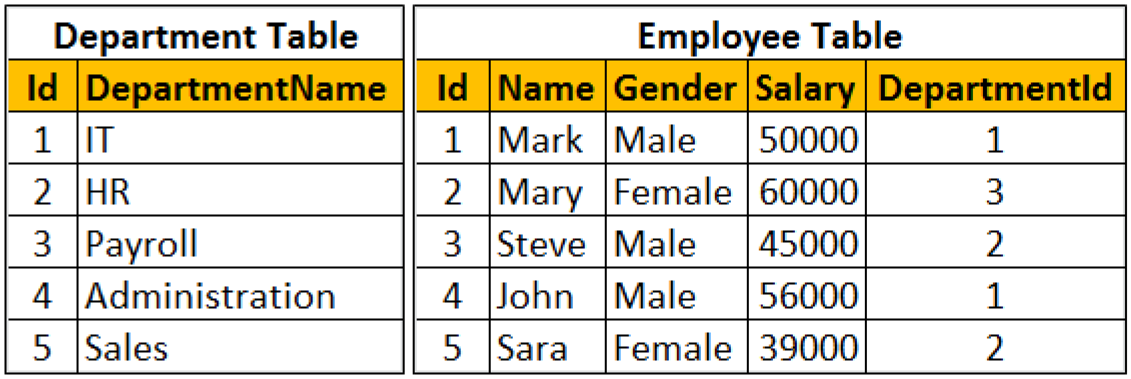
For example, if the number of columns are different, you will get the following error

```
Msg 205, Level 16, State 1, Line 1
All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists
```

Part 91-Cross apply and outer apply in sql server

In this video we will discuss **cross apply and outer apply in sql server** with examples.

We will use the following 2 tables for examples in this demo



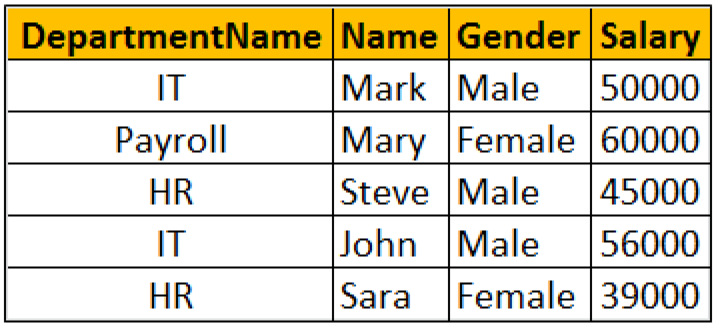
Department Table	
Id	DepartmentName
1	IT
2	HR
3	Payroll
4	Administration
5	Sales

Employee Table				
Id	Name	Gender	Salary	DepartmentId
1	Mark	Male	50000	1
2	Mary	Female	60000	3
3	Steve	Male	45000	2
4	John	Male	56000	1
5	Sara	Female	39000	2

SQL Script to create the tables and populate with test data

```
Create table Department
(
   Id int primary key,
   DepartmentName nvarchar(50)
)
Go

Insert into Department values (1, 'IT')
Insert into Department values (2, 'HR')
Insert into Department values (3, 'Payroll')
Insert into Department values (4, 'Administration')
Insert into Department values (5, 'Sales')
Go

Create table Employee
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10),
   Salary int,
   DepartmentId int foreign key references Department(Id)
)
Go

Insert into Employee values (1, 'Mark', 'Male', 50000, 1)
Insert into Employee values (2, 'Mary', 'Female', 60000, 3)
Insert into Employee values (3, 'Steve', 'Male', 45000, 2)
Insert into Employee values (4, 'John', 'Male', 56000, 1)
Insert into Employee values (5, 'Sara', 'Female', 39000, 2)
Go
```

We want to retrieve all the matching rows between **Department** and **Employee** tables.

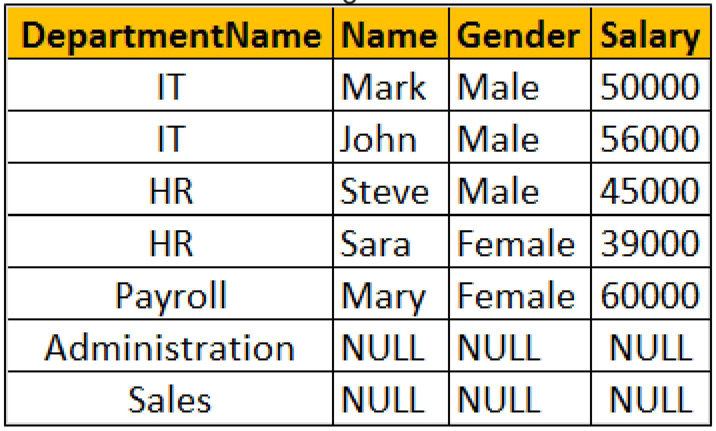


DepartmentName	Name	Gender	Salary
IT	Mark	Male	50000
Payroll	Mary	Female	60000
HR	Steve	Male	45000
IT	John	Male	56000
HR	Sara	Female	39000

This can be very easily achieved using an Inner Join as shown below.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Inner Join Employee E
On D.Id = E.DepartmentId
```

Now if we want to retrieve all the matching rows between **Department** and **Employee** tables + the non-matching rows from the LEFT table (**Department**)



DepartmentName	Name	Gender	Salary
IT	Mark	Male	50000
IT	John	Male	56000
HR	Steve	Male	45000
HR	Sara	Female	39000
Payroll	Mary	Female	60000
Administration	NULL	NULL	NULL
Sales	NULL	NULL	NULL

This can be very easily achieved using a Left Join as shown below.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Left Join Employee E
On D.Id = E.DepartmentId
```

Now let's assume we do not have access to the Employee table. Instead we have access to the following Table Valued function, that returns all employees belonging to a department by Department Id.

```
Create function fn_GetEmployeesByDepartmentId(@DepartmentId int)
Returns Table
as
Return
(
   Select Id, Name, Gender, Salary, DepartmentId
   from Employee where DepartmentId = @DepartmentId
)
Go
```

The following query returns the employees of the department with Id =1.

```
Select * from fn_GetEmployeesByDepartmentId(1)
```

Now if you try to perform an Inner or Left join between **Department** table and **fn_GetEmployeesByDepartmentId**() function you will get an error.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Inner Join fn_GetEmployeesByDepartmentId(D.Id) E
On D.Id = E.DepartmentId
```

If you execute the above query you will get the following error

```
Msg 4104, Level 16, State 1, Line 3
The multi-part identifier "D.Id" could not be bound.
```

This is where we use **Cross Apply** and **Outer Apply** operators. **Cross Apply** is semantically equivalent to **Inner Join** and **Outer Apply** is semantically equivalent to **Left Outer Join**.

Just like Inner Join, Cross Apply retrieves only the matching rows from the Department table and fn_GetEmployeesByDepartmentId() table valued function.

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Cross Apply fn_GetEmployeesByDepartmentId(D.Id) E
```

Just like Left Outer Join, Outer Apply retrieves all matching rows from the Department table and fn_GetEmployeesByDepartmentId() table valued function + non-matching rows from the left table (Department)

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
Outer Apply fn_GetEmployeesByDepartmentId(D.Id) E
```

How does Cross Apply and Outer Apply work

- The APPLY operator introduced in SQL Server 2005, is used to join a table to a table-valued function.
- The Table Valued Function on the right hand side of the APPLY operator gets called for each row from the left (also called outer table) table.
- Cross Apply returns only matching rows (semantically equivalent to Inner Join)
- Outer Apply returns matching + non-matching rows (semantically equivalent to Left Outer Join). The unmatched columns of the table valued function will be set to NULL

Part 92-DDL Triggers in sql server

In this video we will discuss **DDL Triggers in sql server**.

In SQL Server there are 4 types of triggers
1. DML Triggers - Data Manipulation Language. Discussed in Parts 43 to 47 of SQL Server Tutorial.
2. DDL Triggers - Data Definition Language
3. CLR triggers - Common Language Runtime
4. Logon triggers

What are DDL triggers
DDL triggers fire in response to DDL events - CREATE, ALTER, and DROP (Table, Function, Index, Stored Procedure etc...). For the list of all DDL events please visit https://msdn.microsoft.com/en-us/library/bb522542.aspx

Certain system stored procedures that perform DDL-like operations can also fire DDL triggers. Example - sp_rename system stored procedure

What is the use of DDL triggers

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

Syntax for creating DDL trigger
```
CREATE TRIGGER [Trigger_Name]

ON [Scope (Server|Database)]

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

AS

BEGIN

  -- Trigger Body

END
```

DDL triggers scope : DDL triggers can be created in a specific database or at the server level.

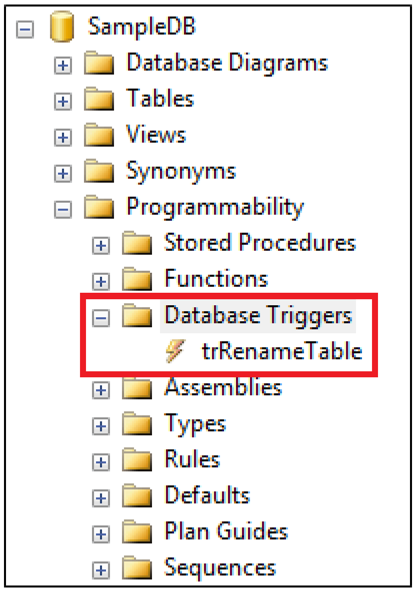
The following trigger will fire in response to CREATE_TABLE DDL event.

```
CREATE TRIGGER trMyFirstTrigger
```

```
ON Database
FOR CREATE_TABLE
AS
BEGIN
   Print 'New table created'
END
```

To check if the trigger has been created

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



Please note : If you can't find the trigger that you just created, make sure to refresh the Database Triggers folder.

When you execute the following code to create the table, the trigger will automatically fire and will print the message - New table created

```
Create Table Test (Id int)
```

The above trigger will be fired only for one DDL event CREATE_TABLE. If you want this trigger to be fired for multiple events, for example when you alter or drop a table, then separate the events using a comma as shown below.

```
ALTER TRIGGER trMyFirstTrigger
```

```
ON Database

FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE

AS

BEGIN

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

END
```

Now if you create, alter or drop a table, the trigger will fire automatically and you will get the message - A table has just been created, modified or deleted.

The 2 DDL triggers above execute some code in response to DDL events

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

```
ALTER TRIGGER trMyFirstTrigger
ON Database

FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE

AS

BEGIN

  Rollback

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

END
```

To be able to create, alter or drop a table, you either have to disable or delete the trigger.

To disable trigger
1. Right click on the trigger in object explorer and select **"Disable"** from the context menu
2. You can also disable the trigger using the following T-SQL command

```
DISABLE TRIGGER trMyFirstTrigger ON DATABASE
```

To enable trigger
1. Right click on the trigger in object explorer and select "Enable" from the context menu
2. You can also enable the trigger using the following T-SQL command

```
ENABLE TRIGGER trMyFirstTrigger ON DATABASE
```

To drop trigger

1. Right click on the trigger in object explorer and select "Delete" from the context menu
2. You can also drop the trigger using the following T-SQL command

```
DROP TRIGGER trMyFirstTrigger ON DATABASE
```

Certain system stored procedures that perform DDL-like operations can also fire DDL triggers. The following trigger will be fired when ever you rename a database object using sp_rename system stored procedure.

```
CREATE TRIGGER trRenameTable
ON DATABASE

FOR RENAME

AS

BEGIN

   Print 'You just renamed something'

END
```

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

```
sp_rename 'TestTable', 'NewTestTable'
```

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

```
sp_rename 'NewTestTable.Id' , 'NewId', 'column'
```

Part 93-Server-scoped ddl triggers

In this video we will discuss **server-scoped ddl triggers**

The following trigger is a database scoped trigger. This will prevent users from creating, altering or dropping tables only from the database in which it is created.

```
CREATE TRIGGER tr_DatabaseScopeTrigger
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   ROLLBACK
   Print 'You cannot create, alter or drop a table in the current database'
END
```

If you have another database on the server, they will be able to create, alter or drop tables in that database. If you want to prevent users from doing this you may create the trigger again in this database.

But, what if you have 100 different databases on your SQL Server, and you want to prevent users from creating, altering or dropping tables from all these 100 databases. Creating the same trigger for all the 100 different databases is not a good approach for 2 reasons.
1. It is tedious and error prone
2. Maintainability is a night mare. If for some reason you have to change the trigger, you will have to do it in 100 different databases, which again is tedious and error prone.

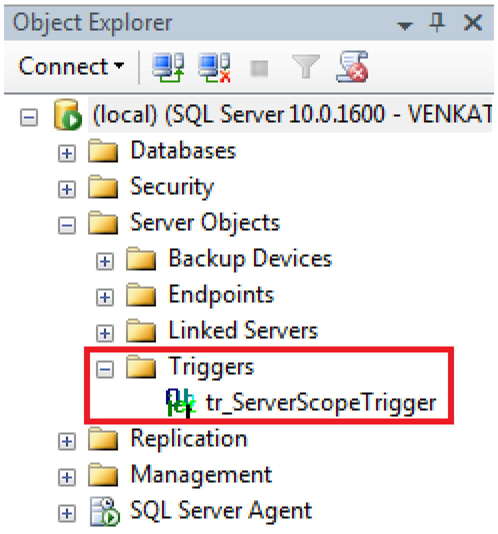
This is where server-scoped DDL triggers come in handy. When you create a server scoped DDL trigger, it will fire in response to the DDL events happening in all of the databases on that server.

Creating a Server-scoped DDL trigger : Similar to creating a database scoped trigger, except that you will have to change the scope to ALL Server as shown below.

```
CREATE TRIGGER tr_ServerScopeTrigger
ON ALL SERVER
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   ROLLBACK
   Print 'You cannot create, alter or drop a table in any database on the server'
END
```

Now if you try to create, alter or drop a table in any of the databases on the server, the trigger will be fired.

Where can I find the Server-scoped DDL triggers
1. In the Object Explorer window, expand "Server Objects" folder
2. Expand Triggers folder



To disable Server-scoped ddl trigger
1. Right click on the trigger in object explorer and select "Disable" from the context menu
2. You can also disable the trigger using the following T-SQL command
DISABLE TRIGGER tr_ServerScopeTrigger ON ALL SERVER

To enable Server-scoped ddl trigger
1. Right click on the trigger in object explorer and select "Enable" from the context menu
2. You can also enable the trigger using the following T-SQL command
ENABLE TRIGGER tr_ServerScopeTrigger ON ALL SERVER

To drop Server-scoped ddl trigger
1. Right click on the trigger in object explorer and select "Delete" from the context menu
2. You can also drop the trigger using the following T-SQL command
DROP TRIGGER tr_ServerScopeTrigger ON ALL SERVER

Part 94-sql server trigger execution order

In this video we will discuss **how to set the execution order of triggers** using **sp_settriggerorder** stored procedure.

Server scoped triggers will always fire before any of the database scoped triggers. This execution order cannot be changed.

In the example below, we have a database-scoped and a server-scoped trigger handling the same event (CREATE_TABLE). When you create a table, notice that server-scoped trigger is always fired before the database-scoped trigger.

CREATE TRIGGER tr_DatabaseScopeTrigger

```
ON DATABASE
FOR CREATE_TABLE
AS
BEGIN
   Print 'Database Scope Trigger'
END
GO

CREATE TRIGGER tr_ServerScopeTrigger
ON ALL SERVER
FOR CREATE_TABLE
AS
BEGIN
   Print 'Server Scope Trigger'
END
GO
```

Using the **sp_settriggerorder** stored procedure, you can set the execution order of server-scoped or database-scoped triggers.

sp_settriggerorder stored procedure has 4 parameters

Parameter	Description
@triggername	Name of the trigger
@order	Value can be First, Last or None. When set to None, trigger is fired in random order
@stmttype	SQL statement that fires the trigger. Can be INSERT, UPDATE, DELETE or any DDL event
@namespace	Scope of the trigger. Value can be DATABASE, SERVER, or NULL

```
EXEC sp_settriggerorder
@triggername = 'tr_DatabaseScopeTrigger1',
@order = 'none',
@stmttype = 'CREATE_TABLE',
@namespace = 'DATABASE'
GO
```

If you have a database-scoped and a server-scoped trigger handling the same event, and if you have set the execution order at both the levels. Here is the execution order of the triggers.
1. The server-scope trigger marked First
2. Other server-scope triggers
3. The server-scope trigger marked Last
4. The database-scope trigger marked First
5. Other database-scope triggers
6. The database-scope trigger marked Last

Part 95-Audit table changes in sql server

In this video we will discuss, **how to audit table changes in SQL Server using a DDL trigger**.

Table to store the audit data

```
Create table TableChanges
(
   DatabaseName nvarchar(250),
   TableName nvarchar(250),
   EventType nvarchar(250),
   LoginName nvarchar(250),
   SQLCommand nvarchar(2500),
   AuditDateTime datetime
)
Go
```

The following trigger audits all table changes in all databases on a SQL Server

```
CREATE TRIGGER tr_AuditTableChanges
ON ALL SERVER
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   DECLARE @EventData XML
   SELECT @EventData = EVENTDATA()

   INSERT INTO SampleDB.dbo.TableChanges
   (DatabaseName, TableName, EventType, LoginName,
    SQLCommand, AuditDateTime)
   VALUES
   (
       @EventData.value('(/EVENT_INSTANCE/DatabaseName)[1]', 'varchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/ObjectName)[1]', 'varchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/EventType)[1]', 'nvarchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/LoginName)[1]', 'varchar(250)'),
       @EventData.value('(/EVENT_INSTANCE/TSQLCommand)[1]', 'nvarchar(2500)'),
       GetDate()
   )
END
```

In the above example we are using **EventData**() function which returns event data in XML format. The following XML is returned by the **EventData**() function when I created a table with name = **MyTable** in **SampleDB** database.

```
<EVENT_INSTANCE>
  <EventType>CREATE_TABLE</EventType>
  <PostTime>2015-09-11T16:12:49.417</PostTime>
  <SPID>58</SPID>
  <ServerName>VENKAT-PC</ServerName>
  <LoginName>VENKAT-PC\Tan</LoginName>
  <UserName>dbo</UserName>
  <DatabaseName>SampleDB</DatabaseName>
  <SchemaName>dbo</SchemaName>
  <ObjectName>MyTable</ObjectName>
  <ObjectType>TABLE</ObjectType>
  <TSQLCommand>
    <SetOptions ANSI_NULLS="ON" ANSI_NULL_DEFAULT="ON"
             ANSI_PADDING="ON" QUOTED_IDENTIFIER="ON"
             ENCRYPTED="FALSE" />
    <CommandText>
      Create Table MyTable
      (
         Id int,
         Name nvarchar(50),
         Gender nvarchar(50)
      )
    </CommandText>
  </TSQLCommand>
</EVENT_INSTANCE>
```

Part 96-Logon triggers in sql server

In this video we will discuss **Logon triggers in SQL Server**.

As the name implies **Logon triggers fire in response to a LOGON event**. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established.

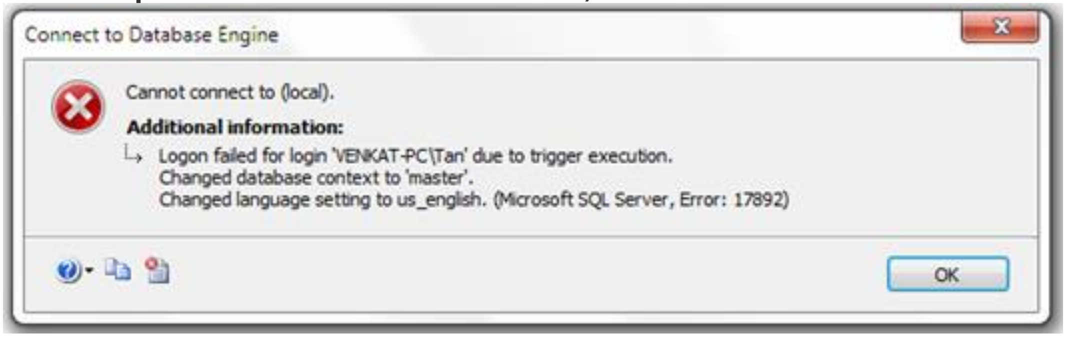
Logon triggers can be used for
1. Tracking login activity
2. Restricting logins to SQL Server
3. Limiting the number of sessions for a specific login

Logon trigger example : The following trigger limits the maximum number of open connections for a user to 3.

```
CREATE TRIGGER tr_LogonAuditTriggers
ON ALL SERVER
FOR LOGON
AS
BEGIN
   DECLARE @LoginName NVARCHAR(100)
```

```
    Set @LoginName = ORIGINAL_LOGIN()

    IF (SELECT COUNT(*) FROM sys.dm_exec_sessions
        WHERE is_user_process = 1
        AND original_login_name = @LoginName) > 3
    BEGIN
        Print 'Fourth connection of ' + @LoginName + ' blocked'
        ROLLBACK
    END
END
```

An attempt to make a fourth connection, will be blocked.



Connect to Database Engine

Cannot connect to (local).

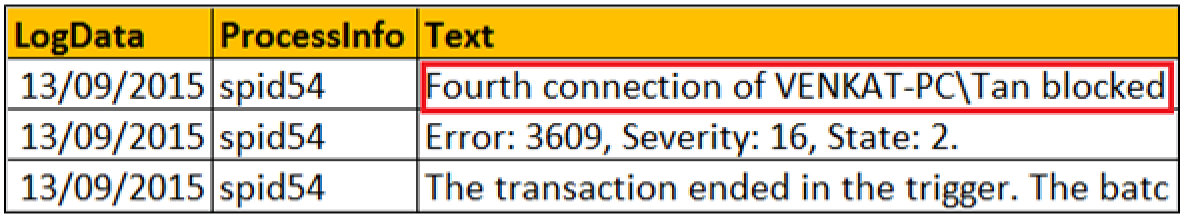
Additional information:

Logon failed for login 'VENKAT-PC\Tan' due to trigger execution.
Changed database context to 'master'.
Changed language setting to us_english. (Microsoft SQL Server, Error: 17892)

OK

The trigger error message will be written to the error log. Execute the following command to read the error log.

```
Execute sp_readerrorlog
```

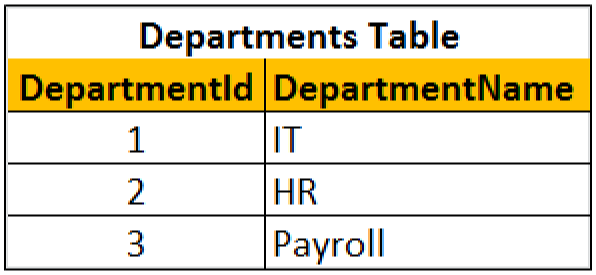


LogData	ProcessInfo	Text
13/09/2015	spid54	Fourth connection of VENKAT-PC\Tan blocked
13/09/2015	spid54	Error: 3609, Severity: 16, State: 2.
13/09/2015	spid54	The transaction ended in the trigger. The batc

Part 97-Select into in sql server

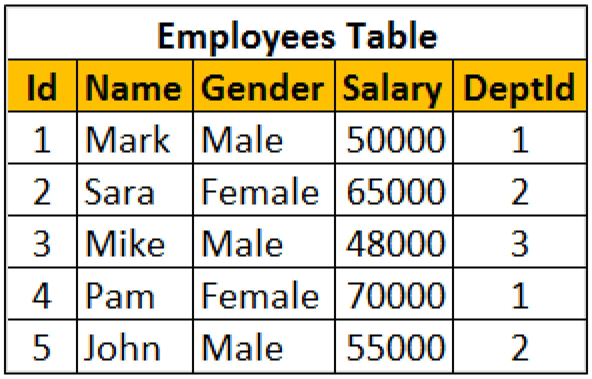
In this video we will discuss the power and use of **SELECT INTO statement in SQL Server**.

We will be using the following 2 tables for the examples.



Departments Table

DepartmentId	DepartmentName
1	IT
2	HR
3	Payroll



Employees Table

Id	Name	Gender	Salary	DeptId
1	Mark	Male	50000	1
2	Sara	Female	65000	2
3	Mike	Male	48000	3
4	Pam	Female	70000	1
5	John	Male	55000	2

SQL Script to create Departments and Employees tables

```
Create table Departments
(
   DepartmentId int primary key,
   DepartmentName nvarchar(50)
)
Go

Insert into Departments values (1, 'IT')
Insert into Departments values (2, 'HR')
Insert into Departments values (3, 'Payroll')
Go

Create table Employees
(
   Id int primary key,
   Name nvarchar(100),
   Gender nvarchar(10),
   Salary int,
   DeptId int foreign key references Departments(DepartmentId)
)
Go

Insert into Employees values (1, 'Mark', 'Male', 50000, 1)
Insert into Employees values (2, 'Sara', 'Female', 65000, 2)
Insert into Employees values (3, 'Mike', 'Male', 48000, 3)
Insert into Employees values (4, 'Pam', 'Female', 70000, 1)
Insert into Employees values (5, 'John', 'Male', 55000, 2)
Go
```

The **SELECT INTO statement in SQL Server**, selects data from one table and inserts it into a new table.

SELECT INTO statement in SQL Server can do the following

1. Copy all rows and columns from an existing table into a new table. This is extremely useful when you want to make a backup copy of the existing table.

```
SELECT * INTO EmployeesBackup FROM Employees
```

2. Copy all rows and columns from an existing table into a new table in an external database.

```
SELECT * INTO HRDB.dbo.EmployeesBackup FROM Employees
```

3. Copy only selected columns into a new table

```
SELECT Id, Name, Gender INTO EmployeesBackup FROM Employees
```

4. Copy only selected rows into a new table

```
SELECT * INTO EmployeesBackup FROM Employees WHERE DeptId = 1
```

5. Copy columns from 2 or more table into a new table

```
SELECT * INTO EmployeesBackup
FROM Employees
INNER JOIN Departments
ON Employees.DeptId = Departments.DepartmentId
```

6. Create a new table whose columns and datatypes match with an existing table.

```
SELECT * INTO EmployeesBackup FROM Employees WHERE 1 <> 1
```

7. Copy all rows and columns from an existing table into a new table on a different SQL Server instance. For this, create a linked server and use the 4 part naming convention

```
SELECT * INTO TargetTable
FROM [SourceServer].[SourceDB].[dbo].[SourceTable]
```

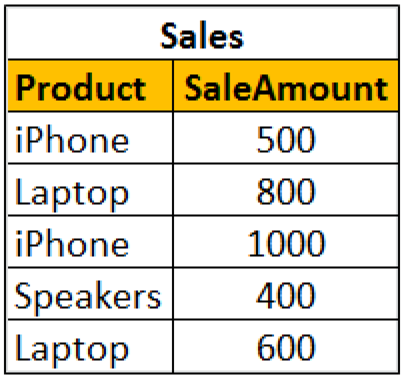
Please note : You cannot use SELECT INTO statement to select data into an existing table. For this you will have to use INSERT INTO statement.

```
INSERT INTO ExistingTable (ColumnList)
SELECT ColumnList FROM SourceTable
```

Part 98-Difference between where and having in sql server

In this video we will discuss the **difference between where and having** clauses in SQL Server.

Let us understand the difference with an example. For the examples in this video we will use the following Sales table.



Sales

Product	SaleAmount
iPhone	500
Laptop	800
iPhone	1000
Speakers	400
Laptop	600

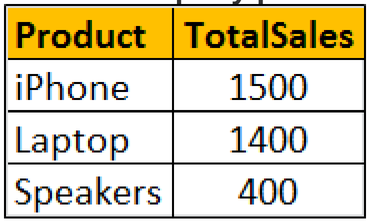
SQL Script to create and populate Sales table with test data

```
Create table Sales
(
    Product nvarchar(50),
    SaleAmount int
)
Go

Insert into Sales values ('iPhone', 500)
Insert into Sales values ('Laptop', 800)
Insert into Sales values ('iPhone', 1000)
Insert into Sales values ('Speakers', 400)
Insert into Sales values ('Laptop', 600)
Go
```

To calculate total sales by product, we would write a GROUP BY query as shown below

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
```

The above query produces the following result



Product	TotalSales
iPhone	1500
Laptop	1400
Speakers	400

Now if we want to find only those **products where the total sales amount is greater than $1000**, we will use HAVING clause to filter products

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
HAVING SUM(SaleAmount) > 1000
```

Result :

Product	TotalSales
iPhone	1500
Laptop	1400

If we use WHERE clause instead of HAVING clause, we will get a syntax error. This is because the WHERE clause doesn't work with aggregate functions like sum, min, max, avg, etc.

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
WHERE SUM(SaleAmount) > 1000
```

So in short, the difference is **WHERE clause cannot be used with aggregates where as HAVING can.**

However, there are other differences as well that we need to keep in mind when using WHERE and HAVING clauses. WHERE clause filters rows before aggregate calculations are performed where as HAVING clause filters rows after aggregate calculations are performed. Let us understand this with an example.

Total sales of iPhone and Speakers can be calculated by using either WHERE or HAVING clause

Calculate Total sales of iPhone and Speakers using WHERE clause : In this example the WHERE clause retrieves only iPhone and Speaker products and then performs the sum.

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
WHERE Product in ('iPhone', 'Speakers')
GROUP BY Product
```

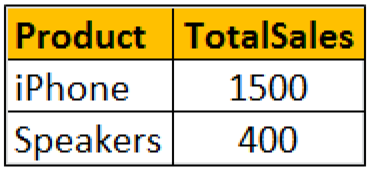
Result :

Product	TotalSales
iPhone	1500
Speakers	400

Calculate Total sales of iPhone and Speakers using HAVING clause : This example retrieves all rows from Sales table, performs the sum and then removes all products except iPhone and Speakers.

```
SELECT Product, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY Product
HAVING Product in ('iPhone', 'Speakers')
```

Result :



Product	TotalSales
iPhone	1500
Speakers	400

So from a performance standpoint, HAVING is slower than WHERE and should be avoided when possible.

Another difference is WHERE comes before GROUP BY and HAVING comes after GROUP BY.

Difference between WHERE and Having
1. WHERE clause cannot be used with aggregates where as HAVING can. This means WHERE clause is used for filtering individual rows where as HAVING clause is used to filter groups.

2. WHERE comes before GROUP BY. This means WHERE clause filters rows before aggregate calculations are performed. HAVING comes after GROUP BY. This means HAVING clause filters rows after aggregate calculations are performed. So from a performance standpoint, HAVING is slower than WHERE and should be avoided when possible.

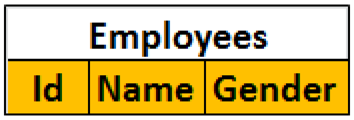
3. WHERE and HAVING can be used together in a SELECT query. In this case WHERE clause is applied first to filter individual rows. The rows are then grouped and aggregate calculations are performed, and then the HAVING clause filters the groups

Part 99-Table valued parameters in SQL Server

In this video we will discuss **table valued parameters in SQL Server**.

Table Valued Parameter is a new feature introduced in SQL SERVER 2008. Table Valued Parameter allows a table (i.e multiple rows of data) to be passed as a parameter to a stored procedure from T-SQL code or from an application. Prior to SQL SERVER 2008, it is not possible to pass a table variable as a parameter to a stored procedure.

Let us understand how to pass multiple rows to a stored procedure using Table Valued Parameter with an example. We want to insert multiple rows into the following Employees table. At the moment this table does not have any rows.



Employees		
Id	Name	Gender

SQL Script to create the Employees table
```
Create Table Employees
(
   Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10)
)
Go
```

Step 1 : Create User-defined Table Type

```
CREATE TYPE EmpTableType AS TABLE
(
    Id INT PRIMARY KEY,
    Name NVARCHAR(50),
    Gender NVARCHAR(10)
)
Go
```

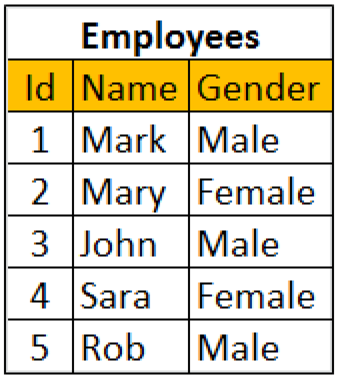
Step 2 : Use the User-defined Table Type as a parameter in the stored procedure. Table valued parameters must be passed as read-only to stored procedures, functions etc. This means you cannot perform DML operations like INSERT, UPDATE or DELETE on a table-valued parameter in the body of a function, stored procedure etc.

```
CREATE PROCEDURE spInsertEmployees
@EmpTableType EmpTableType READONLY
AS
BEGIN
    INSERT INTO Employees
    SELECT * FROM @EmpTableType
END
```

Step 3 : Declare a table variable, insert the data and then pass the table variable as a parameter to the stored procedure.

```
DECLARE @EmployeeTableType EmpTableType

INSERT INTO @EmployeeTableType VALUES (1, 'Mark', 'Male')
INSERT INTO @EmployeeTableType VALUES (2, 'Mary', 'Female')
INSERT INTO @EmployeeTableType VALUES (3, 'John', 'Male')
INSERT INTO @EmployeeTableType VALUES (4, 'Sara', 'Female')
INSERT INTO @EmployeeTableType VALUES (5, 'Rob', 'Male')

EXECUTE spInsertEmployees @EmployeeTableType
```

That's it. Now select the data from Employees table and notice that all the rows of the table variable are inserted into the Employees table.



Employees		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	John	Male
4	Sara	Female
5	Rob	Male

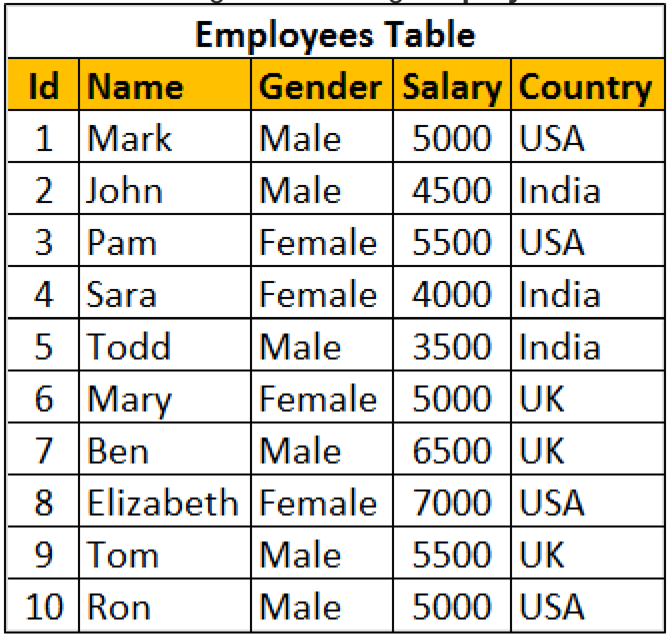
In our next video, we will discuss **how to pass table as a parameter to the stored procedure from an ADO.NET application**

Part 100 is also related to Ado.net and Angular JS so I skipped this.

Part 101-Grouping Sets in SQL Server

Grouping sets is a new feature introduced in SQL Server 2008. Let us understand Grouping sets with an example.

We will be using the following **Employees table** for the examples in this video.

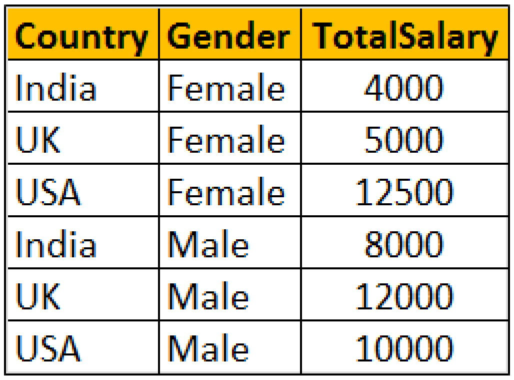


Employees Table				
Id	Name	Gender	Salary	Country
1	Mark	Male	5000	USA
2	John	Male	4500	India
3	Pam	Female	5500	USA
4	Sara	Female	4000	India
5	Todd	Male	3500	India
6	Mary	Female	5000	UK
7	Ben	Male	6500	UK
8	Elizabeth	Female	7000	USA
9	Tom	Male	5500	UK
10	Ron	Male	5000	USA

SQL Script to create and populate Employees table

```
Create Table Employees
(
   Id int primary key,
   Name nvarchar(50),
    Gender nvarchar(10),
    Salary int,
    Country nvarchar(10)
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 5000, 'USA')
Insert Into Employees Values (2, 'John', 'Male', 4500, 'India')
Insert Into Employees Values (3, 'Pam', 'Female', 5500, 'USA')
Insert Into Employees Values (4, 'Sara', 'Female', 4000, 'India')
Insert Into Employees Values (5, 'Todd', 'Male', 3500, 'India')
Insert Into Employees Values (6, 'Mary', 'Female', 5000, 'UK')
Insert Into Employees Values (7, 'Ben', 'Male', 6500, 'UK')
Insert Into Employees Values (8, 'Elizabeth', 'Female', 7000, 'USA')
Insert Into Employees Values (9, 'Tom', 'Male', 5500, 'UK')
Insert Into Employees Values (10, 'Ron', 'Male', 5000, 'USA')
Go
```

We want to calculate **Sum of Salary by Country and Gender**. The result should be as shown below.

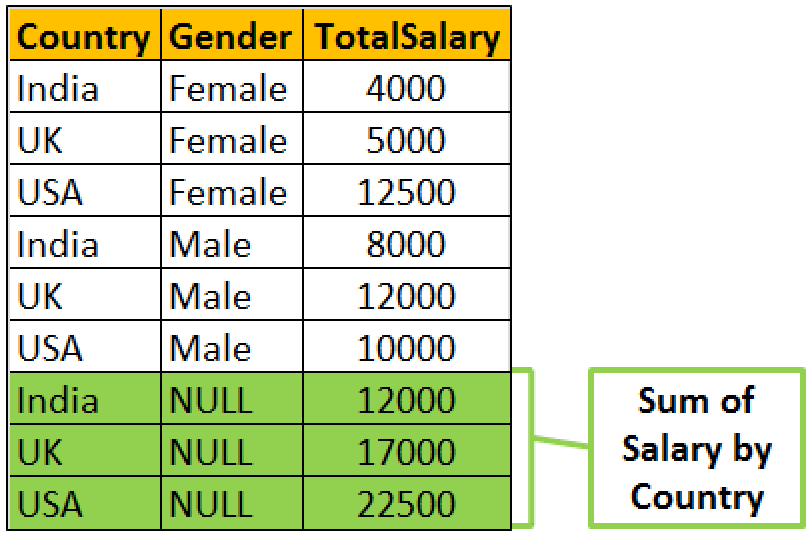


Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000

We can very easily achieve this using a Group By query as shown below

```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender
```

Within the same result set we also want Sum of Salary just by Country. The Result should be as shown below. Notice that Gender column within the resultset is NULL as we are grouping only by Country column



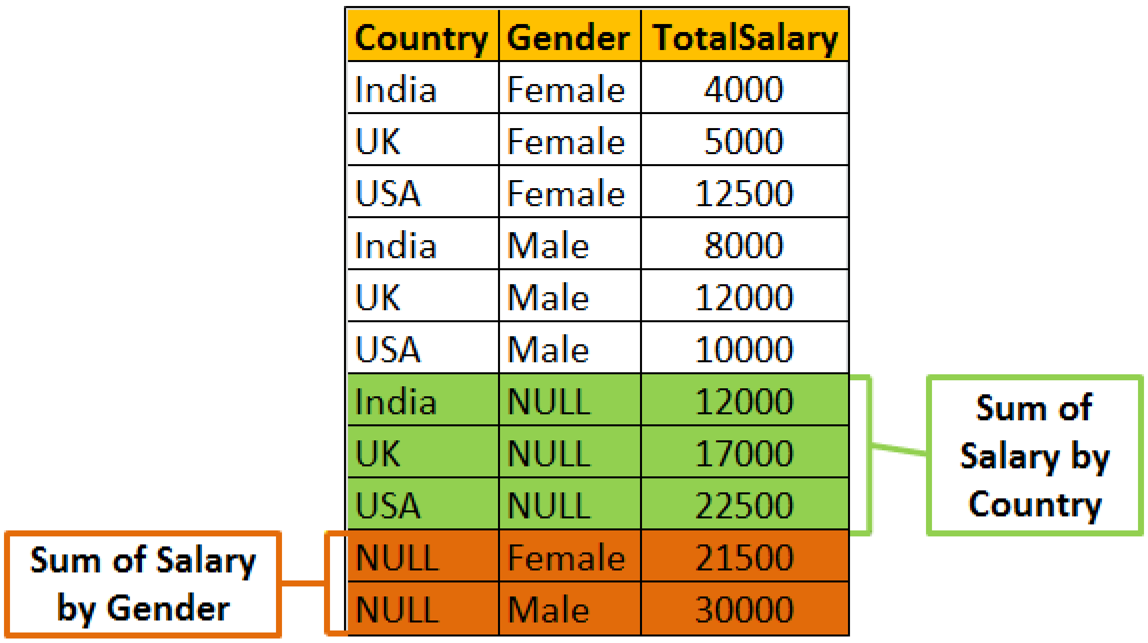
Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000
India	NULL	12000
UK	NULL	17000
USA	NULL	22500

Sum of Salary by Country

To achieve the above result we could combine 2 Group By queries using UNION ALL as shown below.

```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender

UNION ALL

Select Country, NULL, Sum(Salary) as TotalSalary
From Employees
Group By Country
```

Within the same result set we also want Sum of Salary just by Gender. The Result should be as shown below. Notice that the Country column within the resultset is NULL as we are grouping only by Gender column.



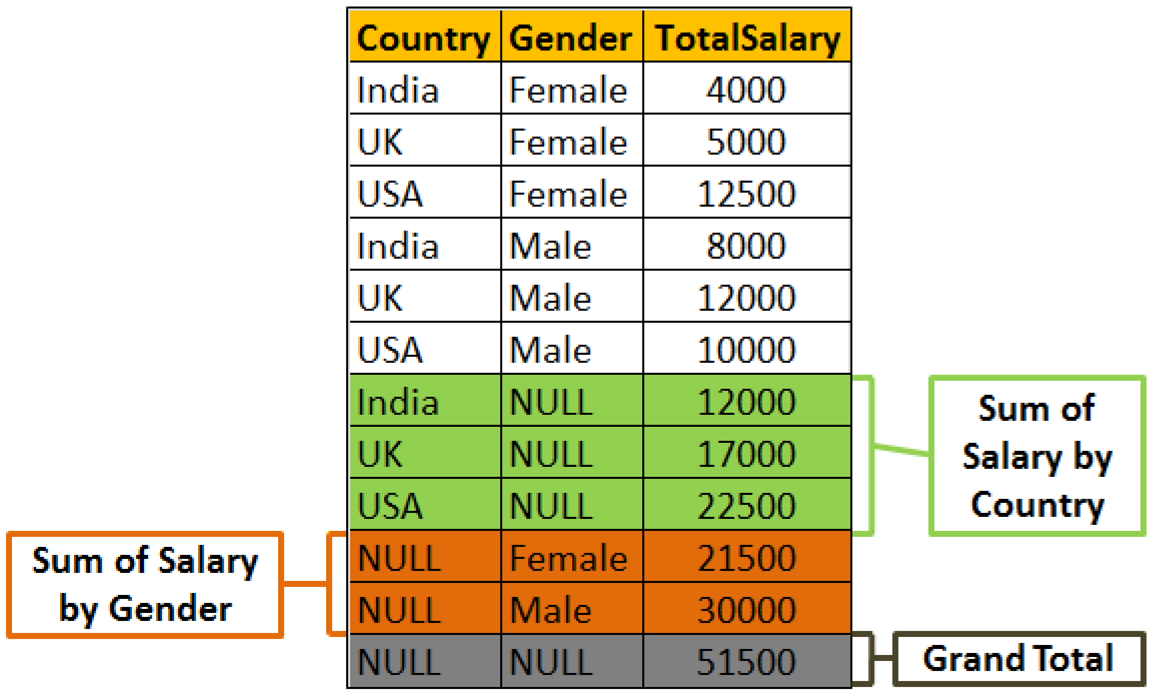
Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000
India	NULL	12000
UK	NULL	17000
USA	NULL	22500
NULL	Female	21500
NULL	Male	30000

Sum of Salary by Country
Sum of Salary by Gender

We can achieve this by combining 3 Group By queries using UNION ALL as shown below

```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender

UNION ALL

Select Country, NULL, Sum(Salary) as TotalSalary
From Employees
Group By Country

UNION ALL

Select NULL, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Gender
```

Finally we also want the grand total of Salary. In this case we are not grouping on any particular column. So both Country and Gender columns will be NULL in the resultset.



Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000
India	NULL	12000
UK	NULL	17000
USA	NULL	22500
NULL	Female	21500
NULL	Male	30000
NULL	NULL	51500

Sum of Salary by Country
Sum of Salary by Gender
Grand Total

To achieve this we will have to combine the fourth query using UNION ALL as shown below.

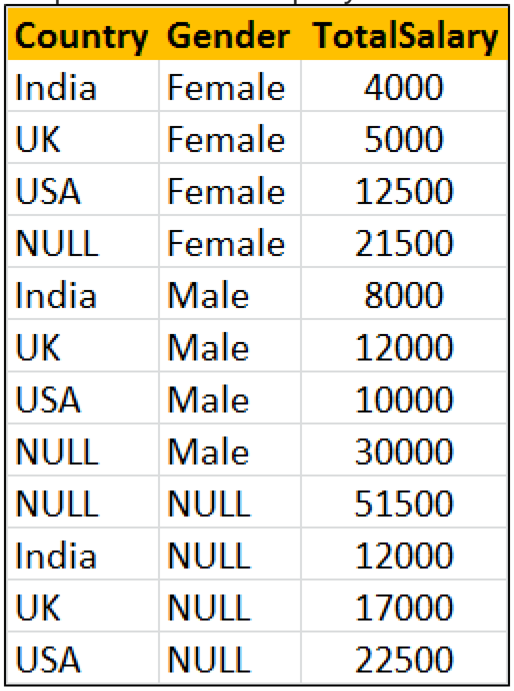
```
Select Country, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Country, Gender

UNION ALL

Select Country, NULL, Sum(Salary) as TotalSalary
From Employees
Group By Country

UNION ALL

Select NULL, Gender, Sum(Salary) as TotalSalary
From Employees
Group By Gender

UNION ALL

Select NULL, NULL, Sum(Salary) as TotalSalary
From Employees
```

There are 2 problems with the above approach.
1. The query is huge as we have combined different Group By queries using UNION ALL operator. This can grow even more if we start to add more groups
2. The Employees table has to be accessed 4 times, once for every query.

If we use **Grouping Sets** feature introduced in SQL Server 2008, the amount of T-SQL code that you have to write will be greatly reduced. The following Grouping Sets query produce the same result as the above UNION ALL query.

```
Select Country, Gender, Sum(Salary) TotalSalary
From Employees
Group BY
     GROUPING SETS
     (
          (Country, Gender), -- Sum of Salary by Country and Gender
          (Country),         -- Sum of Salary by Country
          (Gender) ,          -- Sum of Salary by Gender
          ()                  -- Grand Total
     )
```

Output of the above query

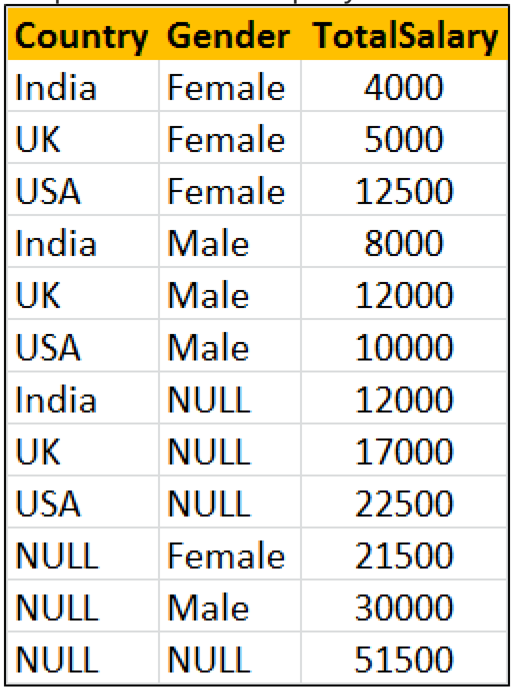


Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
NULL	Female	21500
India	Male	8000
UK	Male	12000
USA	Male	10000
NULL	Male	30000
NULL	NULL	51500
India	NULL	12000
UK	NULL	17000
USA	NULL	22500

The order of the rows in the result set is not the same as in the case of UNION ALL query. To control the order use order by as shown below.

```
Select Country, Gender, Sum(Salary) TotalSalary
From Employees
Group BY
     GROUPING SETS
     (
          (Country, Gender), -- Sum of Salary by Country and Gender
          (Country),         -- Sum of Salary by Country
          (Gender) ,          -- Sum of Salary by Gender
          ()                          -- Grand Total
     )
Order By Grouping(Country), Grouping(Gender), Gender
```

Output of the above query

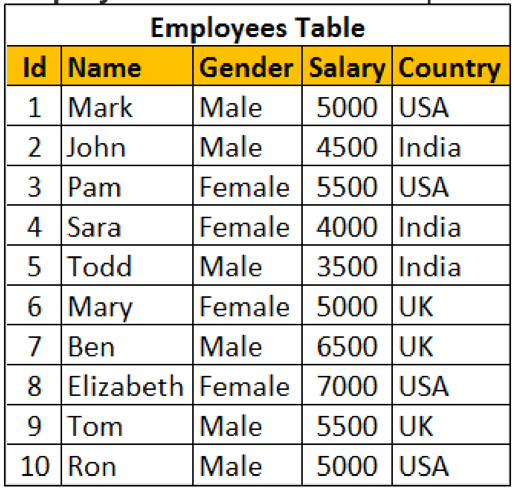


Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000
India	NULL	12000
UK	NULL	17000
USA	NULL	22500
NULL	Female	21500
NULL	Male	30000
NULL	NULL	51500

Part 102-Roll up in SQL Server

ROLLUP in SQL Server is used to do aggregate operation on multiple levels in hierarchy.

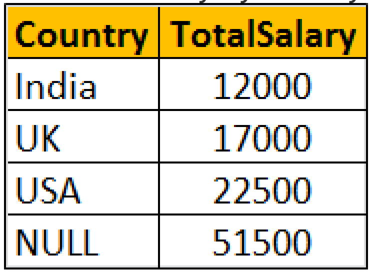
Let us understand Rollup in SQL Server with examples. We will use the following **Employees table** for the examples in this video.



Employees Table

Id	Name	Gender	Salary	Country
1	Mark	Male	5000	USA
2	John	Male	4500	India
3	Pam	Female	5500	USA
4	Sara	Female	4000	India
5	Todd	Male	3500	India
6	Mary	Female	5000	UK
7	Ben	Male	6500	UK
8	Elizabeth	Female	7000	USA
9	Tom	Male	5500	UK
10	Ron	Male	5000	USA

Retrieve Salary by country along with grand total



Country	TotalSalary
India	12000
UK	17000
USA	22500
NULL	51500

There are several ways to achieve this. The easiest way is by using Rollup with Group By.

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees

GROUP BY ROLLUP(Country)
```

The above query can also be rewritten as shown below

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country WITH ROLLUP
```

We can also use UNION ALL operator along with GROUP BY

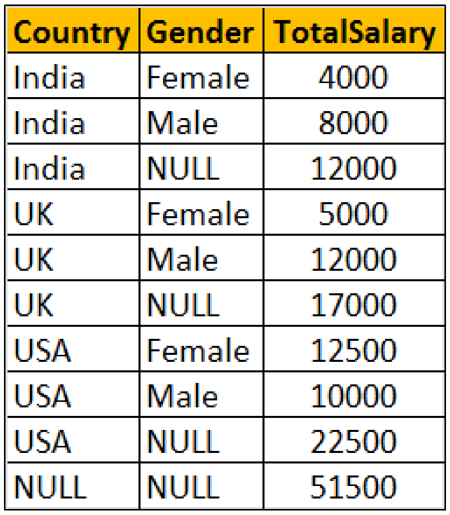
```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country

UNION ALL

SELECT NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

We can also use Grouping Sets to achieve the same result

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY GROUPING SETS
(
   (Country),
   ()
)
```

Let's look at another example.

Group Salary by Country and Gender. Also compute the Subtotal for Country level and Grand Total as shown below.



Country	Gender	TotalSalary
India	Female	4000
India	Male	8000
India	NULL	12000
UK	Female	5000
UK	Male	12000
UK	NULL	17000
USA	Female	12500
USA	Male	10000
USA	NULL	22500
NULL	NULL	51500

Using ROLLUP with GROUP BY

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY ROLLUP(Country, Gender)

--OR
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country, Gender WITH ROLLUP
```

Using UNION ALL with GROUP BY

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country, Gender

UNION ALL
SELECT Country, NULL, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country

UNION ALL
SELECT NULL, NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

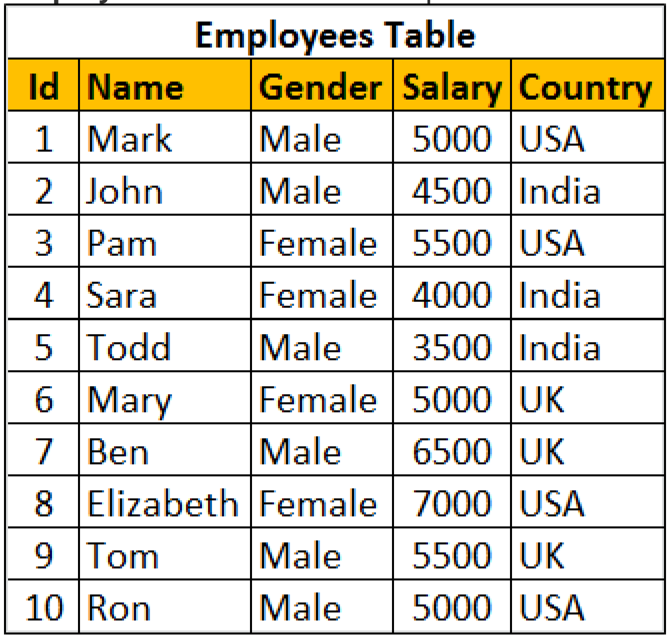
Using GROUPING SETS

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY GROUPING SETS
(
   (Country, Gender),
   (Country),
   ()
)
```

Part 103-Cube in SQL Server

Cube() in SQL Server produces the result set by generating all combinations of columns specified in GROUP BY CUBE().

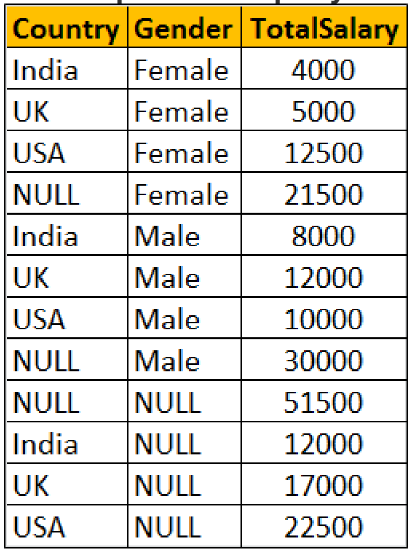
Let us understand Cube() in SQL Server with examples. We will use the following **Employees table** for the examples in this video.



Employees Table				
Id	**Name**	**Gender**	**Salary**	**Country**
1	Mark	Male	5000	USA
2	John	Male	4500	India
3	Pam	Female	5500	USA
4	Sara	Female	4000	India
5	Todd	Male	3500	India
6	Mary	Female	5000	UK
7	Ben	Male	6500	UK
8	Elizabeth	Female	7000	USA
9	Tom	Male	5500	UK
10	Ron	Male	5000	USA

Write a query to retrieve Sum of Salary grouped by all combinations of the following 2 columns as well as Grand Total.
Country,
Gender

The output of the query should be as shown below



Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
NULL	Female	21500
India	Male	8000
UK	Male	12000
USA	Male	10000
NULL	Male	30000
NULL	NULL	51500
India	NULL	12000
UK	NULL	17000
USA	NULL	22500

Using Cube with Group By

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Cube(Country, Gender)

--OR

SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country, Gender with Cube
```

The above query is equivalent to the following Grouping Sets query

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY
   GROUPING SETS
   (
       (Country, Gender),
       (Country),
       (Gender),
       ()
   )
```

The above query is equivalent to the following UNION ALL query. While the data in the result set is the same, the ordering is not. Use ORDER BY to control the ordering of rows in the result set.

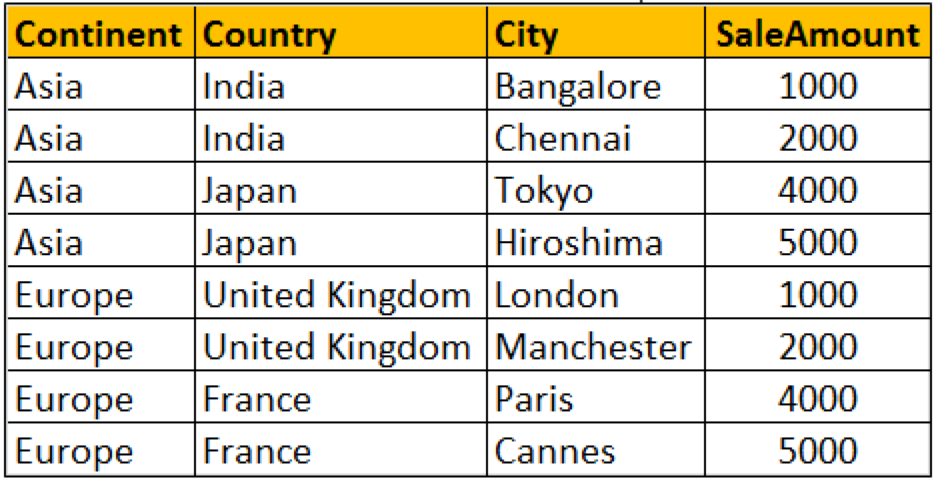
```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country, Gender

UNION ALL

SELECT Country, NULL, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country

UNION ALL

SELECT NULL, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Gender

UNION ALL

SELECT NULL, NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

Part104-Difference between cube and rollup in SQL Server

In this video we will discuss the **difference between cube and rollup in SQL Server.**

CUBE generates a result set that shows aggregates for all combinations of values in the selected columns, where as ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

Let us understand this difference with an example. Consider the following **Sales** table.



Continent	Country	City	SaleAmount
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	Japan	Tokyo	4000
Asia	Japan	Hiroshima	5000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	France	Paris	4000
Europe	France	Cannes	5000

SQL Script to create and populate Sales table

```
Create table Sales
(
   Id int primary key identity,
   Continent nvarchar(50),
   Country nvarchar(50),
   City nvarchar(50),
   SaleAmount int
)
Go

Insert into Sales values('Asia','India','Bangalore',1000)
Insert into Sales values('Asia','India','Chennai',2000)
Insert into Sales values('Asia','Japan','Tokyo',4000)
Insert into Sales values('Asia','Japan','Hiroshima',5000)
Insert into Sales values('Europe','United Kingdom','London',1000)
Insert into Sales values('Europe','United Kingdom','Manchester',2000)
Insert into Sales values('Europe','France','Paris',4000)
Insert into Sales values('Europe','France','Cannes',5000)
Go
```

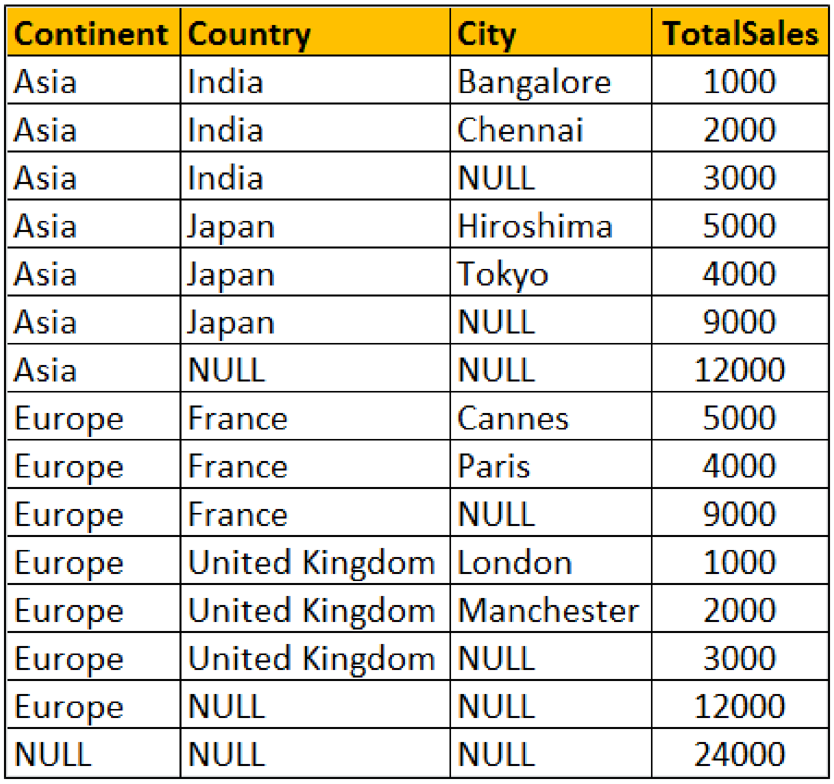
ROLLUP(Continent, Country, City) produces Sum of Salary for the following hierarchy

```
Continent, Country, City
Continent, Country,
Continent
()
```

CUBE(Continent, Country, City) produces Sum of Salary for all the following column combinations

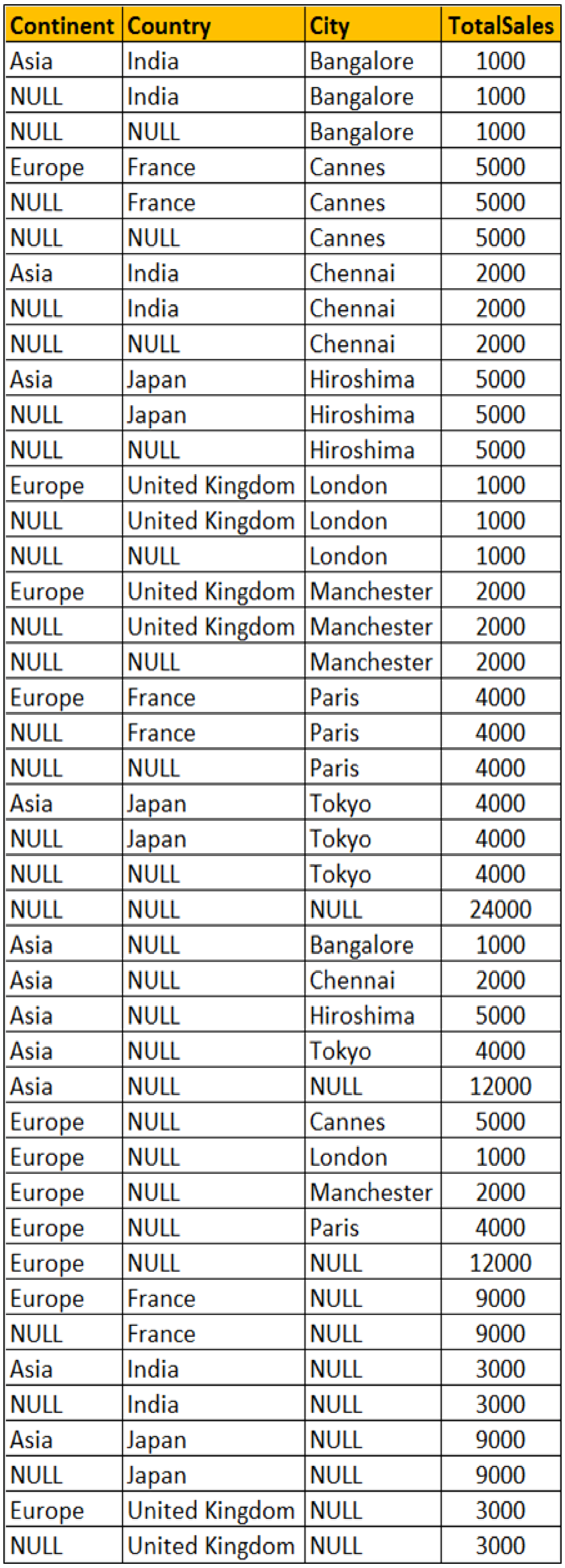
Continent, Country, City
Continent, Country,
Continent, City
Continent
Country, City
Country,
City
()

```
SELECT Continent, Country, City, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```



Continent	Country	City	TotalSales
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	India	NULL	3000
Asia	Japan	Hiroshima	5000
Asia	Japan	Tokyo	4000
Asia	Japan	NULL	9000
Asia	NULL	NULL	12000
Europe	France	Cannes	5000
Europe	France	Paris	4000
Europe	France	NULL	9000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	United Kingdom	NULL	3000
Europe	NULL	NULL	12000
NULL	NULL	NULL	24000

```
SELECT Continent, Country, City, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY CUBE(Continent, Country, City)
```



Continent	Country	City	TotalSales
Asia	India	Bangalore	1000
NULL	India	Bangalore	1000
NULL	NULL	Bangalore	1000
Europe	France	Cannes	5000
NULL	France	Cannes	5000
NULL	NULL	Cannes	5000
Asia	India	Chennai	2000
NULL	India	Chennai	2000
NULL	NULL	Chennai	2000
Asia	Japan	Hiroshima	5000
NULL	Japan	Hiroshima	5000
NULL	NULL	Hiroshima	5000
Europe	United Kingdom	London	1000
NULL	United Kingdom	London	1000
NULL	NULL	London	1000
Europe	United Kingdom	Manchester	2000
NULL	United Kingdom	Manchester	2000
NULL	NULL	Manchester	2000
Europe	France	Paris	4000
NULL	France	Paris	4000
NULL	NULL	Paris	4000
Asia	Japan	Tokyo	4000
NULL	Japan	Tokyo	4000
NULL	NULL	Tokyo	4000
NULL	NULL	NULL	24000
Asia	NULL	Bangalore	1000
Asia	NULL	Chennai	2000
Asia	NULL	Hiroshima	5000
Asia	NULL	Tokyo	4000
Asia	NULL	NULL	12000
Europe	NULL	Cannes	5000
Europe	NULL	London	1000
Europe	NULL	Manchester	2000
Europe	NULL	Paris	4000
Europe	NULL	NULL	12000
Europe	France	NULL	9000
NULL	France	NULL	9000
Asia	India	NULL	3000
NULL	India	NULL	3000
Asia	Japan	NULL	9000
NULL	Japan	NULL	9000
Europe	United Kingdom	NULL	3000
NULL	United Kingdom	NULL	3000

You won't see any difference when you use ROLLUP and CUBE on a single column. Both the following queries produces the same output.

```
SELECT Continent, Sum(SaleAmount) AS TotalSales
FROM Sales
GROUP BY ROLLUP(Continent)

-- OR

SELECT Continent, SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY CUBE(Continent)
```

Continent	TotalSales
Asia	12000
Europe	12000
NULL	24000

Part-105 Grouping function in SQL Server

In this video we will discuss the use of **Grouping function in SQL Server**.

This is continuation to Part 104. Please watch Part 104 from SQL Server tutorial before proceeding. We will use the following Sales table for this example.

Continent	Country	City	SaleAmount
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	Japan	Tokyo	4000
Asia	Japan	Hiroshima	5000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	France	Paris	4000
Europe	France	Cannes	5000

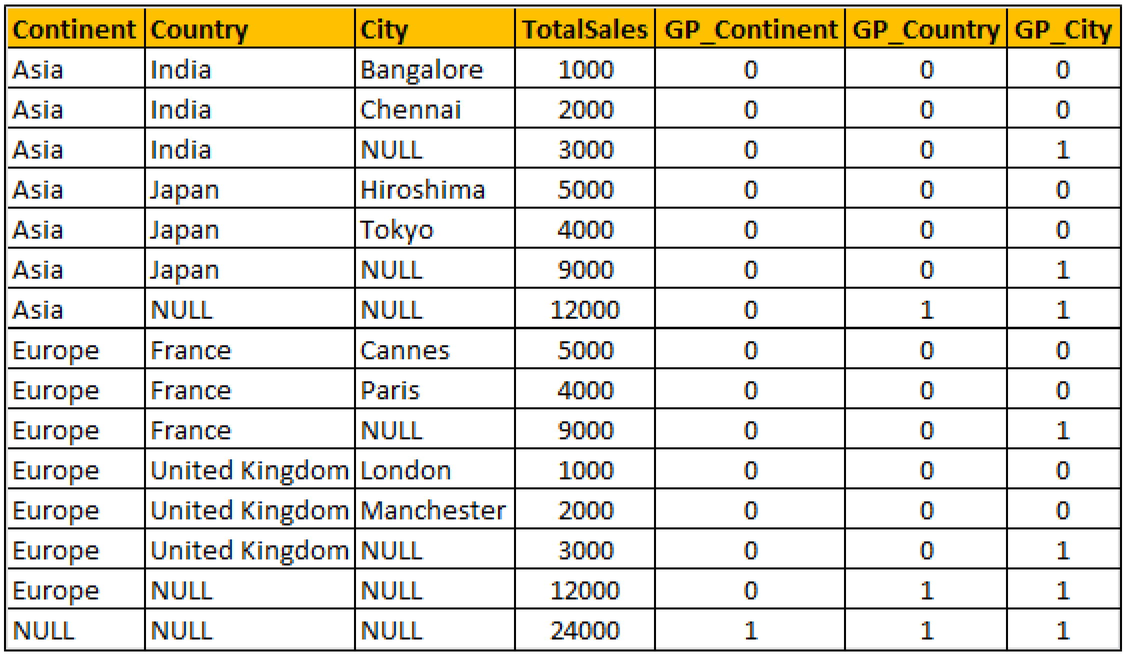
What is Grouping function
Grouping(Column) indicates whether the column in a GROUP BY list is aggregated or not.

Grouping returns 1 for aggregated or 0 for not aggregated in the result set.

The following query returns 1 for aggregated or 0 for not aggregated in the result set

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
       GROUPING(Continent) AS GP_Continent,

       GROUPING(Country) AS GP_Country,

       GROUPING(City) AS GP_City

FROM Sales

GROUP BY ROLLUP(Continent, Country, City)
```

Result :



Continent	Country	City	TotalSales	GP_Continent	GP_Country	GP_City
Asia	India	Bangalore	1000	0	0	0
Asia	India	Chennai	2000	0	0	0
Asia	India	NULL	3000	0	0	1
Asia	Japan	Hiroshima	5000	0	0	0
Asia	Japan	Tokyo	4000	0	0	0
Asia	Japan	NULL	9000	0	0	1
Asia	NULL	NULL	12000	0	1	1
Europe	France	Cannes	5000	0	0	0
Europe	France	Paris	4000	0	0	0
Europe	France	NULL	9000	0	0	1
Europe	United Kingdom	London	1000	0	0	0
Europe	United Kingdom	Manchester	2000	0	0	0
Europe	United Kingdom	NULL	3000	0	0	1
Europe	NULL	NULL	12000	0	1	1
NULL	NULL	NULL	24000	1	1	1

What is the use of Grouping function in real world
When a column is aggregated in the result set, the column will have a NULL value. If you want to replace NULL with All then this GROUPING function is very handy.

```
SELECT
   CASE WHEN

       GROUPING(Continent) = 1 THEN 'All' ELSE ISNULL(Continent, 'Unknown')

   END AS Continent,

   CASE WHEN

       GROUPING(Country) = 1 THEN 'All' ELSE ISNULL(Country, 'Unknown')
```

```
    END AS Country,
    CASE
        WHEN GROUPING(City) = 1 THEN 'All' ELSE ISNULL(City, 'Unknown')
    END AS City,
    SUM(SaleAmount) AS TotalSales
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

Result :

Continent	Country	City	TotalSales
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	India	All	3000
Asia	Japan	Hiroshima	5000
Asia	Japan	Tokyo	4000
Asia	Japan	All	9000
Asia	All	All	12000
Europe	France	Cannes	5000
Europe	France	Paris	4000
Europe	France	All	9000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	United Kingdom	All	3000
Europe	All	All	12000
All	All	All	24000

Can't I use ISNULL function instead as shown below

```
SELECT   ISNULL(Continent, 'All') AS Continent,
        ISNULL(Country, 'All') AS Country,

        ISNULL(City, 'All') AS City,

        SUM(SaleAmount) AS TotalSales

FROM Sales


GROUP BY ROLLUP(Continent, Country, City)
```

Well, you can, but only if your data does not contain NULL values. Let me explain what I mean.

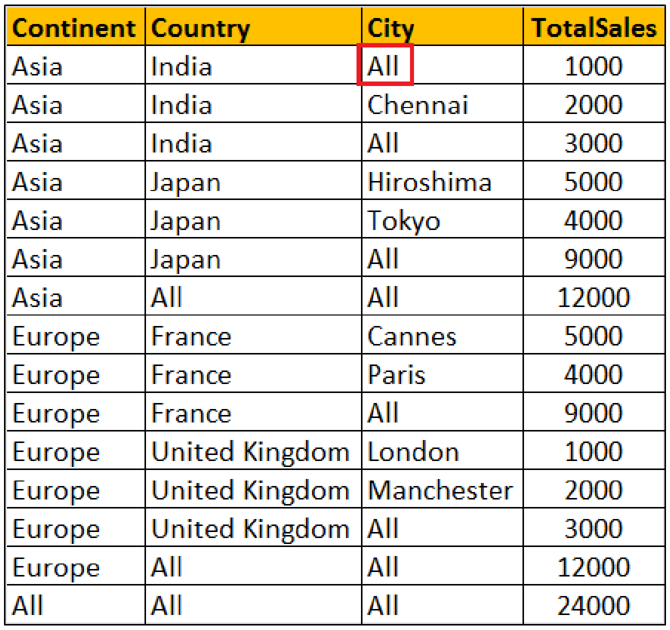
At the moment the raw data in our Sales has no NULL values. Let's introduce a NULL value in the City column of the row where Id = 1

```
Update Sales Set City = NULL where Id = 1
```

Now execute the following query with ISNULL function

```
SELECT   ISNULL(Continent, 'All') AS Continent,
        ISNULL(Country, 'All') AS Country,

        ISNULL(City, 'All') AS City,

        SUM(SaleAmount) AS TotalSales

FROM Sales


GROUP BY ROLLUP(Continent, Country, City)
```

Result : Notice that the actuall NULL value in the raw data is also replaced with the word 'All', which is incorrect. Hence the need for Grouping function.



Continent	Country	City	TotalSales
Asia	India	All	1000
Asia	India	Chennai	2000
Asia	India	All	3000
Asia	Japan	Hiroshima	5000
Asia	Japan	Tokyo	4000
Asia	Japan	All	9000
Asia	All	All	12000
Europe	France	Cannes	5000
Europe	France	Paris	4000
Europe	France	All	9000
Europe	United Kingdom	London	1000
Europe	United Kingdom	Manchester	2000
Europe	United Kingdom	All	3000
Europe	All	All	12000
All	All	All	24000

Please note : Grouping function can be used with Rollup, Cube and Grouping Sets

Part106-GROUPING_ID function in SQL Server

In this video we will discuss
1. GROUPING_ID function in SQL Server
2. Difference between GROUPING and GROUPING_ID functions
3. Use of GROUPING_ID function

GROUPING_ID function computes the level of grouping.

Difference between GROUPING and GROUPING_ID

Syntax : GROUPING function is used on single column, where as the column list for GROUPING_ID function must match with GROUP BY column list.

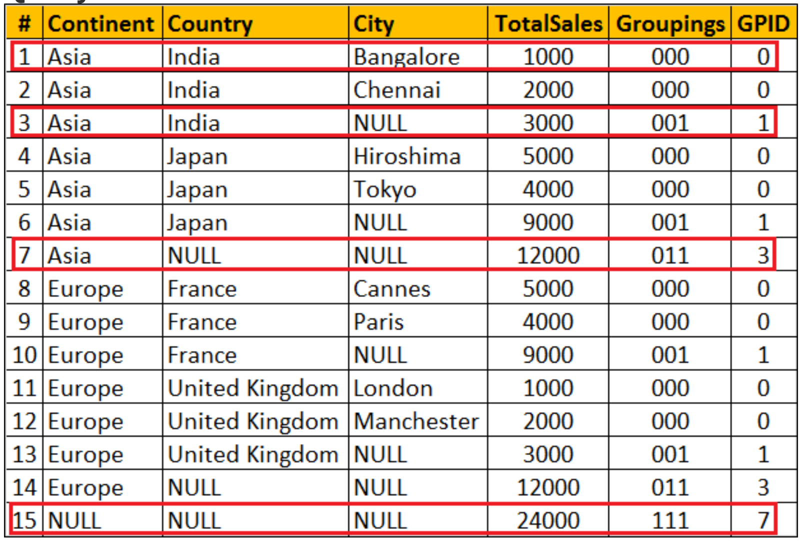
```
GROUPING(Col1)
GROUPING_ID(Col1, Col2, Col3,...)
```

GROUPING indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set.

GROUPING_ID() function concatenates all the GOUPING() functions, perform the binary to decimal conversion, and returns the equivalent integer. In short
GROUPING_ID(A, B, C) = GROUPING(A) + GROUPING(B) + GROUPING(C)
Let us understand this with an example.

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
         CAST(GROUPING(Continent) AS NVARCHAR(1)) +
         CAST(GROUPING(Country) AS NVARCHAR(1)) +
         CAST(GROUPING(City) AS NVARCHAR(1)) AS Groupings,
         GROUPING_ID(Continent, Country, City) AS GPID
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

Query result :



#	Continent	Country	City	TotalSales	Groupings	GPID
1	Asia	India	Bangalore	1000	000	0
2	Asia	India	Chennai	2000	000	0
3	Asia	India	NULL	3000	001	1
4	Asia	Japan	Hiroshima	5000	000	0
5	Asia	Japan	Tokyo	4000	000	0
6	Asia	Japan	NULL	9000	001	1
7	Asia	NULL	NULL	12000	011	3
8	Europe	France	Cannes	5000	000	0
9	Europe	France	Paris	4000	000	0
10	Europe	France	NULL	9000	001	1
11	Europe	United Kingdom	London	1000	000	0
12	Europe	United Kingdom	Manchester	2000	000	0
13	Europe	United Kingdom	NULL	3000	001	1
14	Europe	NULL	NULL	12000	011	3
15	NULL	NULL	NULL	24000	111	7

Row Number 1 : Since the data is not aggregated by any column GROUPING(Continent), GROUPING(Country) and GROUPING(City) return 0 and as result we get a binar string with all ZEROS (000). When this converted to decimal we get 0 which is displayed in GPID column.

Row Number 7 : The data is aggregated for Country and City columns, so GROUPING(Country) and GROUPING(City) return 1 where as GROUPING(Continent) return 0. As result we get a binar string (011). When this converted to decimal we get 10 which is displayed in GPID column.

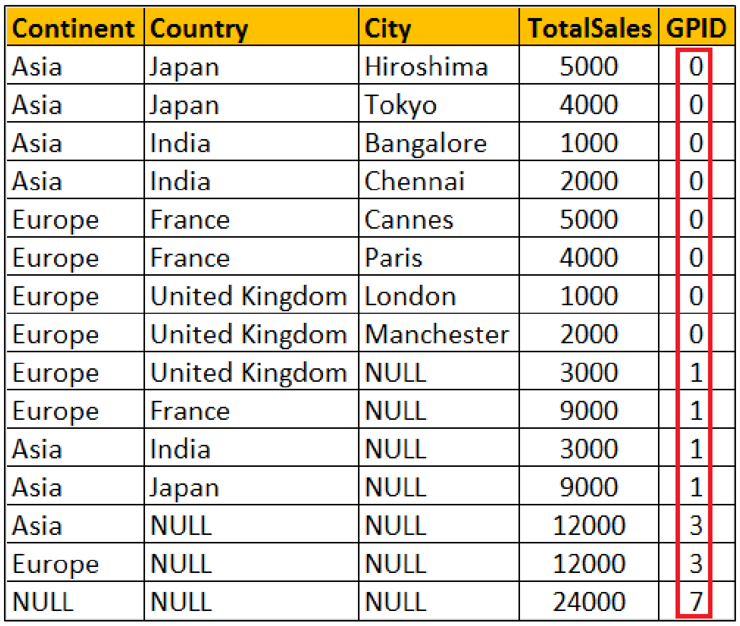
Row Number 15 : This is the Grand total row. Notice in this row the data is aggregated by all the 3 columns. Hence all the 3 GROUPING functions return 1. So we get a binary string with all ONES (111). When this converted to decimal we get 7 which is displayed in GPID column.

Use of GROUPING_ID function : GROUPING_ID function is very handy if you want to sort and filter by level of grouping.

Sorting by level of grouping :

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
       GROUPING_ID(Continent, Country, City) AS GPID
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
ORDER BY GPID
```

Result :



Continent	Country	City	TotalSales	GPID
Asia	Japan	Hiroshima	5000	0
Asia	Japan	Tokyo	4000	0
Asia	India	Bangalore	1000	0
Asia	India	Chennai	2000	0
Europe	France	Cannes	5000	0
Europe	France	Paris	4000	0
Europe	United Kingdom	London	1000	0
Europe	United Kingdom	Manchester	2000	0
Europe	United Kingdom	NULL	3000	1
Europe	France	NULL	9000	1
Asia	India	NULL	3000	1
Asia	Japan	NULL	9000	1
Asia	NULL	NULL	12000	3
Europe	NULL	NULL	12000	3
NULL	NULL	NULL	24000	7

Filter by level of grouping : The following query retrieves only continent level aggregated data

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
       GROUPING_ID(Continent, Country, City) AS GPID
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
HAVING GROUPING_ID(Continent, Country, City) = 3
```

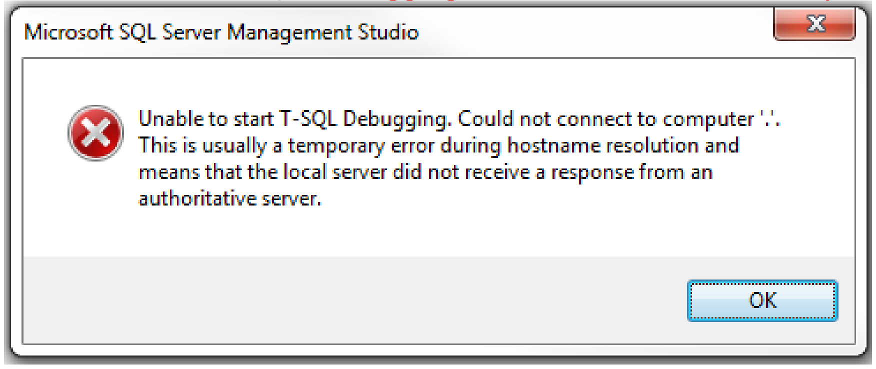
Result :

Continent	Country	City	TotalSales	GPID
Asia	NULL	NULL	12000	3
Europe	NULL	NULL	12000	3

Part107-Debugging sql server stored procedures

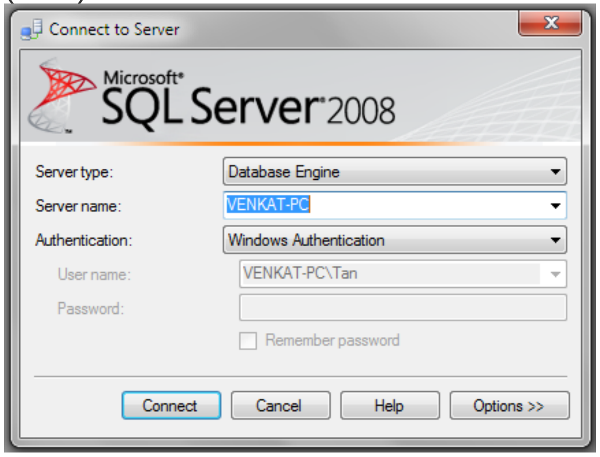
In this video we will discuss **how to debug stored procedures in SQL Server**.

Setting up the Debugger in SSMS : If you have connected to SQL Server using (local) or . (period), and when you start the debugger you will get the following error
Unable to start T-SQL Debugging. Could not connect to computer.


Microsoft SQL Server Management Studio

Unable to start T-SQL Debugging. Could not connect to computer '.'. This is usually a temporary error during hostname resolution and means that the local server did not receive a response from an authoritative server.

To fix this error, use the computer name to connect to the SQL Server instead of using (local) or .


Connect to Server

Server type: Database Engine
Server name: VENKAT-PC
Authentication: Windows Authentication
User name: VENKAT-PC\Tan
Password:
Remember password

For the examples in this video we will be using the following stored procedure.

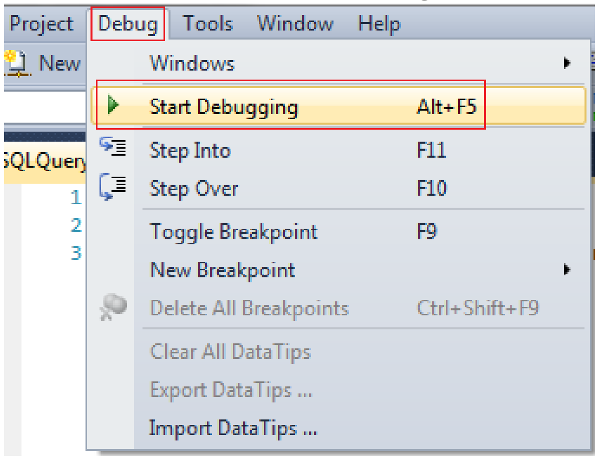
```
Create procedure spPrintEvenNumbers
@Target int
as
Begin
    Declare @StartNumber int
    Set @StartNumber = 1

    while(@StartNumber < @Target)
    Begin
        If(@StartNumber%2 = 0)
        Begin
            Print @StartNumber
        End
        Set @StartNumber = @StartNumber + 1
    End
    Print 'Finished printing even numbers till ' + RTRIM(@Target)
End
```

Connect to SQL Server using your computer name, and then execute the above code to create the stored procedure. At this point, open a New Query window. Copy and paste the following T-SQL code to execute the stored procedure.

```
DECLARE @TargetNumber INT
SET @TargetNumber = 10
EXECUTE spPrintEvenNumbers @TargetNumber
Print 'Done'
```

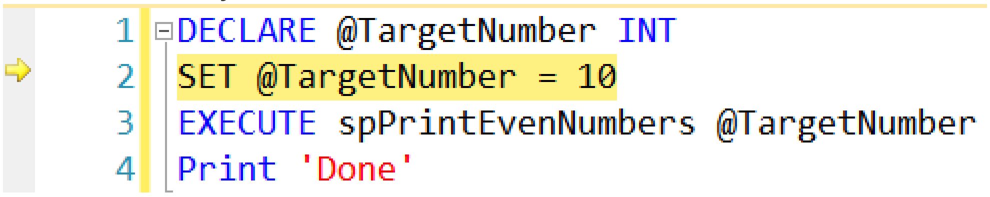
Starting the Debugger in SSMS : There are 2 ways to start the debugger

1. In SSMS, click on the **Debug** Menu and select **Start Debugging**



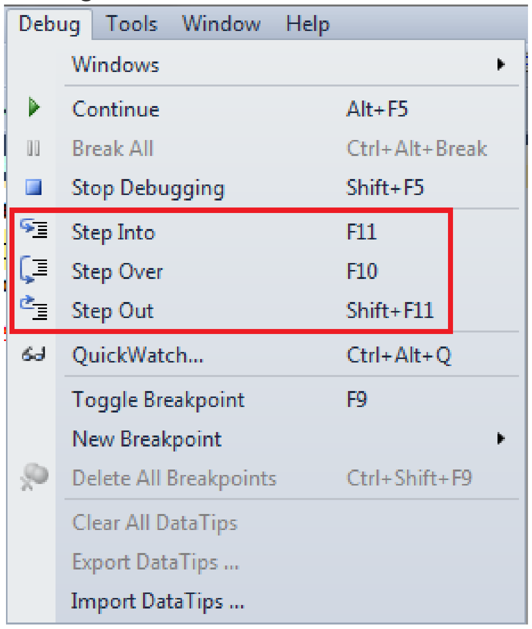
2. Use the keyboard shortcut **ALT + F5**

At this point you should have the debugger running. The line that is about to be executed is marked with an yellow arrow

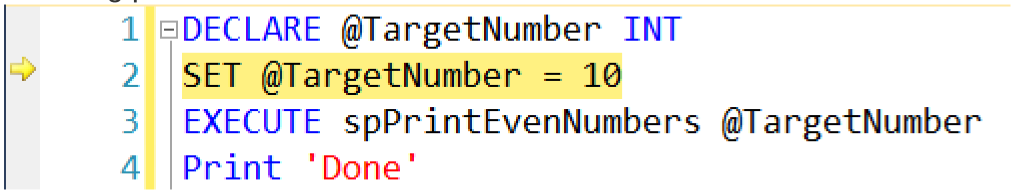


```
DECLARE @TargetNumber INT
SET @TargetNumber = 10
EXECUTE spPrintEvenNumbers @TargetNumber
Print 'Done'
```

Step Over, Step into and Step Out in SSMS : You can find the keyboard shortcuts in the Debug menu in SSMS.



Let us understand what Step Over, Step into and Step Out does when debugging the following piece of code



```
1 DECLARE @TargetNumber INT
2 SET @TargetNumber = 10
3 EXECUTE spPrintEvenNumbers @TargetNumber
4 Print 'Done'
```

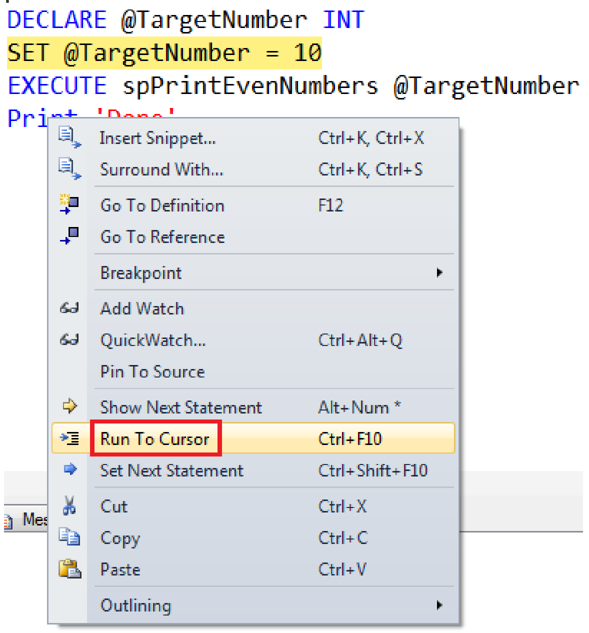
1. There is no difference when you STEP INTO (F11) or STEP OVER (F10) the code on LINE 2

2. On LINE 3, we are calling a Stored Procedure. On this statement if we press F10 (STEP OVER), it won't give us the opportunity to debug the stored procedure code. To be able to debug the stored procedure code you will have to STEP INTO it by pressing F11.

3. If the debugger is in the stored procedure, and you don't want to debug line by line with in that stored procedure, you can STEP OUT of it by pressing SHIFT + F11. When you do this, the debugger completes the execution of the stored procedure and waits on the next line in the main query, i.e on LINE 4 in this example.

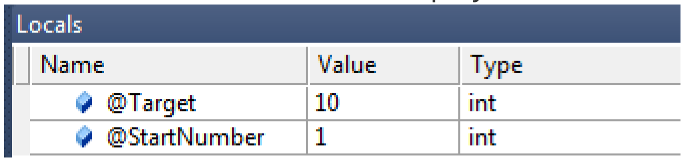
To stop debugging : There are 2 ways to stop debugging
1. In SSMS, click on the Debug Menu and select Stop Debugging
2. Use the keyboard shortcut SHIFT + F5

Show Next Statement shows the next statement that the debugger is about to execute.
Run to Cursor command executes all the statements in a batch up to the current cursor position



```
DECLARE @TargetNumber INT
SET @TargetNumber = 10
EXECUTE spPrintEvenNumbers @TargetNumber
Print 'Done'
```

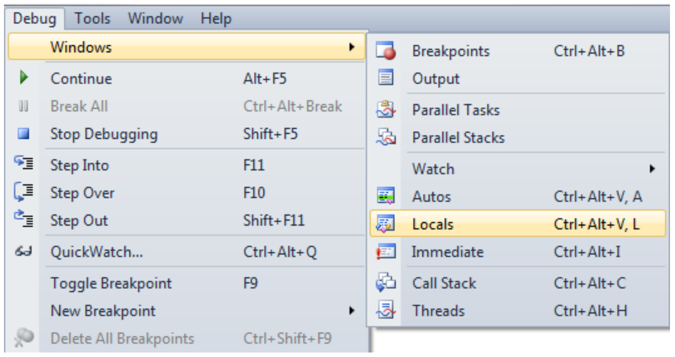
Locals Window in SSMS : Displays the current values of variables and parameters



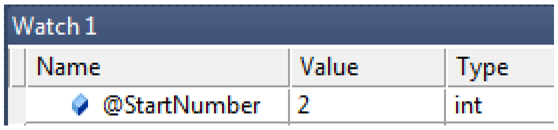
Locals

Name	Value	Type
@Target	10	int
@StartNumber	1	int

If you cannot see the locals window or if you have closed it and if you want to open it, you can do so using the following menu option. Locals window is only available if you are in DEBUG mode.

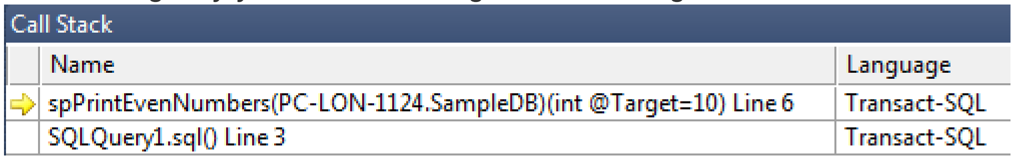


Watch Window in SSMS : Just like Locals window, Watch window is used to watch the values of variables. You can add and remove variables from the watch window. To add a variable to the Watch Window, right click on the variable and select "Add Watch" option from the context menu.


Watch 1

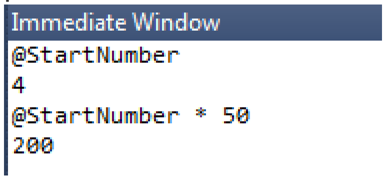
Name	Value	Type
@StartNumber	2	int

Call Stack Window in SSMS : Allows you to navigate up and down the call stack to see what values your application is storing at different levels. It's an invaluable tool for determining why your code is doing what it's doing.


Call Stack

Name	Language
spPrintEvenNumbers(PC-LON-1124.SampleDB)(int @Target=10) Line 6	Transact-SQL
SQLQuery1.sql() Line 3	Transact-SQL

Immediate Window in SSMS : Very helpful during debugging to evaluate expressions, and print variable values. To clear immediate window type **>cls** and press enter.

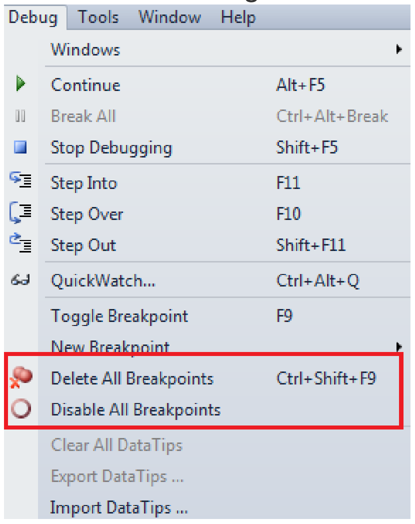

Immediate Window

```
@StartNumber
4
@StartNumber * 50
200
```

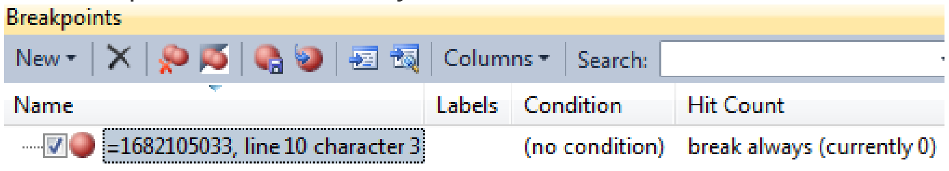
Breakpoints in SSMS : There are 2 ways to set a breakpoint in SSMS.
1. By clicking on the grey margin on the left hand side in SSMS (to remove click again)
2. By pressing F9 (to remove press F9 again)

Enable, Disable or Delete all breakpoints : There are 2 ways to Enable, Disable or Delete all breakpoints

1. From the Debug menu



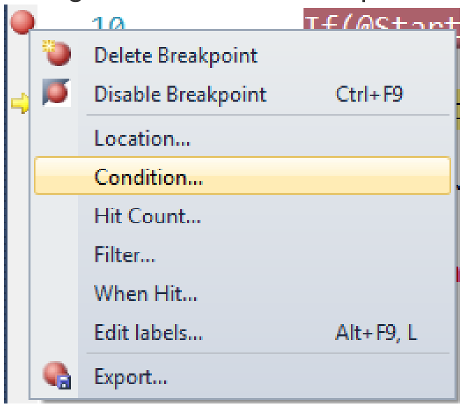
2. From the Breakpoints window. To view Breakpoints window select Debug => Windows => Breakpoints or use the keyboard shortcut ALT + CTRL + B



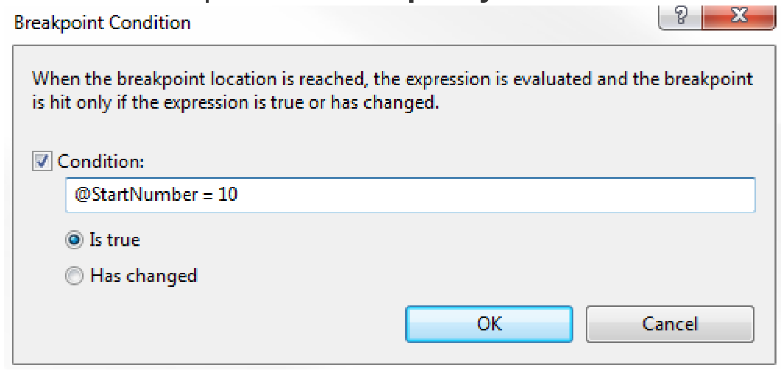
Name	Labels	Condition	Hit Count
=1682105033, line 10 character 3		(no condition)	break always (currently 0)

Conditional Breakpoint : Conditional Breakpoints are hit only when the specified condition is met. These are extremely useful when you have some kind of a loop and you want to break, only when the loop variable has a specific value (For example loop varible = 100).

How to set a conditional break point in SSMS :
1. Right click on the Breakpoint and select **Condition** from the context menu



2. In the Breakpoint window **specify the condition**



When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.

[✓] Condition: @StartNumber = 10

[● Is true] ○ Has changed

Part108-Over clause in SQL Server

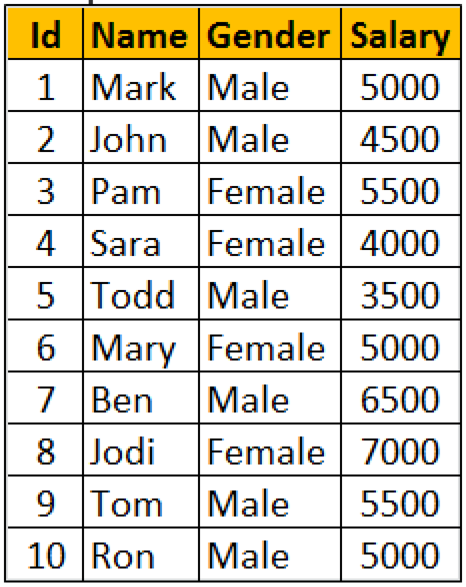
The **OVER** clause combined with **PARTITION BY** is used to break up data into partitions.
Syntax : function (...) OVER (PARTITION BY col1, Col2, ...)

The specified function operates for each partition.

For example :
COUNT(Gender) OVER (PARTITION BY Gender) will partition the data by **GENDER** i.e there will 2 partitions (Male and Female) and then the COUNT() function is applied over each partition.

Any of the following functions can be used. Please note this is not the complete list.
COUNT(), AVG(), SUM(), MIN(), MAX(), ROW_NUMBER(), RANK(), DENSE_RANK() etc.

Example : We will use the following **Employees table** for the examples in this video.



Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

SQl Script to create Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 5000)
Insert Into Employees Values (2, 'John', 'Male', 4500)
Insert Into Employees Values (3, 'Pam', 'Female', 5500)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
Insert Into Employees Values (6, 'Mary', 'Female', 5000)
Insert Into Employees Values (7, 'Ben', 'Male', 6500)
Insert Into Employees Values (8, 'Jodi', 'Female', 7000)
Insert Into Employees Values (9, 'Tom', 'Male', 5500)
Insert Into Employees Values (10, 'Ron', 'Male', 5000)
Go
```

Write a query to retrieve total count of employees by Gender. Also in the result we want Average, Minimum and Maximum salary by Gender. The result of the query should be as shown below.

Gender	GenderTotal	AvgSal	MinSal	MaxSal
Female	4	5375	4000	7000
Male	6	5000	3500	6500

This can be very easily achieved using a simple **GROUP BY** query as show below.

```
SELECT Gender, COUNT(*) AS GenderTotal, AVG(Salary) AS AvgSal,
       MIN(Salary) AS MinSal, MAX(Salary) AS MaxSal
FROM Employees
GROUP BY Gender
```

What if we want **non-aggregated values** (like employee Name and Salary) in result set along with aggregated values

Name	Salary	Gender	GenderTotals	AvgSal	MinSal	MaxSal
Pam	5500	Female	4	5375	4000	7000
Sara	4000	Female	4	5375	4000	7000
Mary	5000	Female	4	5375	4000	7000
Jodi	7000	Female	4	5375	4000	7000
Tom	5500	Male	6	5000	3500	6500
Ron	5000	Male	6	5000	3500	6500
Ben	6500	Male	6	5000	3500	6500
Todd	3500	Male	6	5000	3500	6500
Mark	5000	Male	6	5000	3500	6500
John	4500	Male	6	5000	3500	6500

You cannot include **non-aggregated** columns in the **GROUP BY** query.

```
SELECT Name, Salary, Gender, COUNT(*) AS GenderTotal, AVG(Salary) AS AvgSal,
       MIN(Salary) AS MinSal, MAX(Salary) AS MaxSal
FROM Employees
GROUP BY Gender
```

The above query will result in the following error

```
Column 'Employees.Name' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause
```

One way to achieve this is by including the aggregations in a subquery and then **JOINING** it with the main query as shown in the example below. Look at the amount of T-SQL code we have to write.

```
SELECT Name, Salary, Employees.Gender, Genders.GenderTotals,
       Genders.AvgSal, Genders.MinSal, Genders.MaxSal
```

```
FROM Employees
INNER JOIN
(SELECT Gender, COUNT(*) AS GenderTotals,
         AVG(Salary) AS AvgSal,
        MIN(Salary) AS MinSal, MAX(Salary) AS MaxSal
FROM Employees
GROUP BY Gender) AS Genders
ON Genders.Gender = Employees.Gender
```

Better way of doing this is by using the **OVER** clause combined with **PARTITION BY**

```
SELECT Name, Salary, Gender,
        COUNT(Gender) OVER(PARTITION BY Gender) AS GenderTotals,
        AVG(Salary) OVER(PARTITION BY Gender) AS AvgSal,
        MIN(Salary) OVER(PARTITION BY Gender) AS MinSal,
        MAX(Salary) OVER(PARTITION BY Gender) AS MaxSal
FROM Employees
```

Part109-Row_Number function in SQL Server

In this video we will discuss Row_Number function in SQL Server. This is continuation to Part 108. Please watch Part 108 from SQL Server tutorial before proceeding.

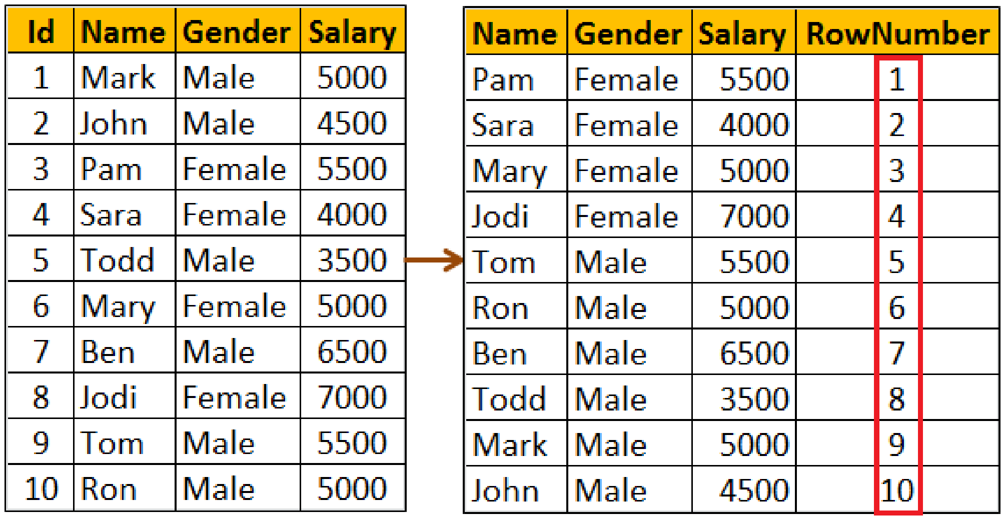
Row_Number function

- Introduced in SQL Server 2005
- Returns the sequential number of a row starting at 1
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, row number is reset to 1 when the partition changes

Syntax : ROW_NUMBER() OVER (ORDER BY Col1, Col2)

Row_Number function without PARTITION BY : In this example, data is not partitioned, so ROW_NUMBER will provide a consecutive numbering for all the rows in the table based on the order of rows imposed by the ORDER BY clause.

```
SELECT Name, Gender, Salary,
        ROW_NUMBER() OVER (ORDER BY Gender) AS RowNumber

FROM Employees
```



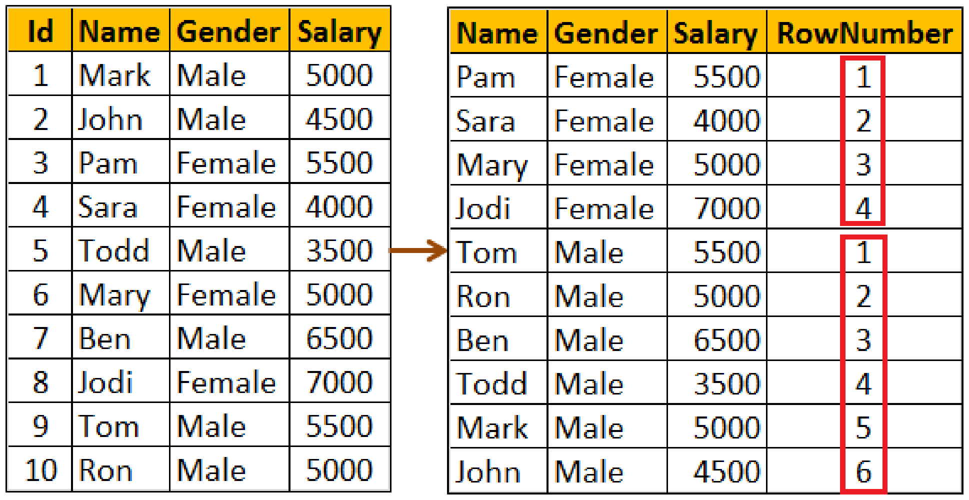
Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

Name	Gender	Salary	RowNumber
Pam	Female	5500	1
Sara	Female	4000	2
Mary	Female	5000	3
Jodi	Female	7000	4
Tom	Male	5500	5
Ron	Male	5000	6
Ben	Male	6500	7
Todd	Male	3500	8
Mark	Male	5000	9
John	Male	4500	10

Please note : If ORDER BY clause is not specified you will get the following error
The function 'ROW_NUMBER' must have an OVER clause with ORDER BY

Row_Number function with PARTITION BY : In this example, data is partitioned by Gender, so ROW_NUMBER will provide a consecutive numbering only for the rows with in a parttion. When the partition changes the row number is reset to 1.

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



Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

Name	Gender	Salary	RowNumber
Pam	Female	5500	1
Sara	Female	4000	2
Mary	Female	5000	3
Jodi	Female	7000	4
Tom	Male	5500	1
Ron	Male	5000	2
Ben	Male	6500	3
Todd	Male	3500	4
Mark	Male	5000	5
John	Male	4500	6

Use case for Row_Number function : Deleting all duplicate rows except one from a sql server table.

Discussed in detail in Part 4 of SQL Server Interview Questions and Answers video series.

Part110-Rank and Dense_Rank in SQL Server

In this video we will discuss **Rank and Dense_Rank functions in SQL Server**

Rank and Dense_Rank functions

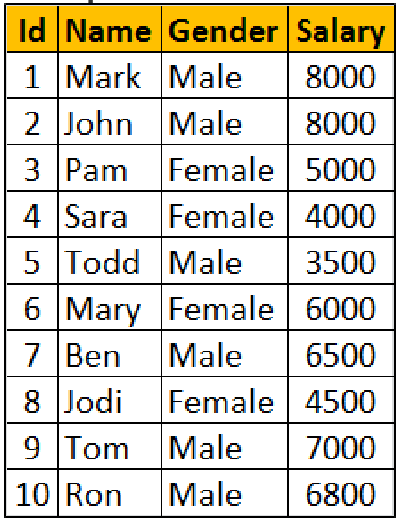
- Introduced in SQL Server 2005
- Returns a rank starting at 1 based on the ordering of rows imposed by the ORDER BY clause
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, rank is reset to 1 when the partition changes

Difference between Rank and Dense_Rank functions
Rank function skips ranking(s) if there is a tie where as Dense_Rank will not.

For example : If you have 2 rows at rank 1 and you have 5 rows in total.
RANK() returns - 1, 1, 3, 4, 5
DENSE_RANK returns - 1, 1, 2, 3, 4

Syntax :
```
RANK() OVER (ORDER BY Col1, Col2, ...)
DENSE_RANK() OVER (ORDER BY Col1, Col2, ...)
```

Example : We will use the following **Employees** table for the examples in this video



Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	5000
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	6000
7	Ben	Male	6500
8	Jodi	Female	4500
9	Tom	Male	7000
10	Ron	Male	6800

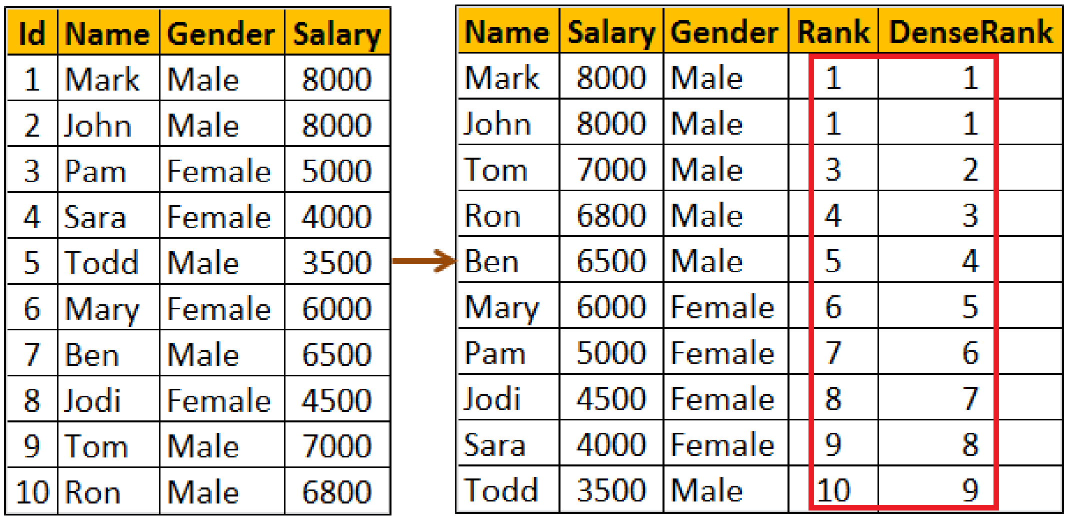
SQl Script to create Employees table
```
Create Table Employees
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10),
   Salary int
)
```

```
Go

Insert Into Employees Values (1, 'Mark', 'Male', 8000)
Insert Into Employees Values (2, 'John', 'Male', 8000)
Insert Into Employees Values (3, 'Pam', 'Female', 5000)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
Insert Into Employees Values (6, 'Mary', 'Female', 6000)
Insert Into Employees Values (7, 'Ben', 'Male', 6500)
Insert Into Employees Values (8, 'Jodi', 'Female', 4500)
Insert Into Employees Values (9, 'Tom', 'Male', 7000)
Insert Into Employees Values (10, 'Ron', 'Male', 6800)
Go
```

RANK() and DENSE_RANK() functions without PARTITION BY clause : In this example, data is not partitioned, so RANK() function provides a consecutive numbering except when there is a tie. Rank 2 is skipped as there are 2 rows at rank 1. The third row gets rank 3.

DENSE_RANK() on the other hand will not skip ranks if there is a tie. The first 2 rows get rank 1. Third row gets rank 2.

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



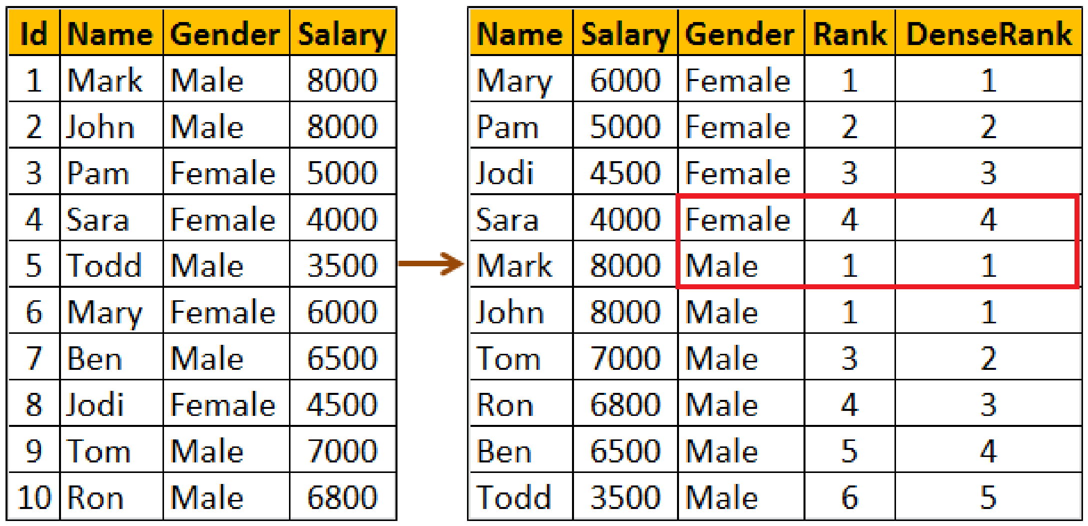
Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	5000
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	6000
7	Ben	Male	6500
8	Jodi	Female	4500
9	Tom	Male	7000
10	Ron	Male	6800

Name	Salary	Gender	Rank	DenseRank
Mark	8000	Male	1	1
John	8000	Male	1	1
Tom	7000	Male	3	2
Ron	6800	Male	4	3
Ben	6500	Male	5	4
Mary	6000	Female	6	5
Pam	5000	Female	7	6
Jodi	4500	Female	8	7
Sara	4000	Female	9	8
Todd	3500	Male	10	9

RANK() and DENSE_RANK() functions with PARTITION BY clause : Notice when the partition changes from Female to Male Rank is reset to 1

```
SELECT Name, Salary, Gender,
RANK() OVER (PARTITION BY Gender ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (PARTITION BY Gender ORDER BY Salary DESC)
AS DenseRank
```

```
FROM Employees
```



Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	5000
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	6000
7	Ben	Male	6500
8	Jodi	Female	4500
9	Tom	Male	7000
10	Ron	Male	6800

Name	Salary	Gender	Rank	DenseRank
Mary	6000	Female	1	1
Pam	5000	Female	2	2
Jodi	4500	Female	3	3
Sara	4000	Female	4	4
Mark	8000	Male	1	1
John	8000	Male	1	1
Tom	7000	Male	3	2
Ron	6800	Male	4	3
Ben	6500	Male	5	4
Todd	3500	Male	6	5

Use case for RANK and DENSE_RANK functions : Both these functions can be used to find Nth highest salary. However, which function to use depends on what you want to do when there is a tie. Let me explain with an example.

If there are 2 employees with the FIRST highest salary, there are 2 different business cases

- If your business case is, not to produce any result for the SECOND highest salary, then use RANK function
- If your business case is to return the next Salary after the tied rows as the SECOND highest Salary, then use DENSE_RANK function

Since we have 2 Employees with the FIRST highest salary. Rank() function will not return any rows for the SECOND highest Salary.

```
WITH Result AS
(
   SELECT Salary, RANK() OVER (ORDER BY Salary DESC) AS Salary_Rank
   FROM Employees
)
SELECT TOP 1 Salary FROM Result WHERE Salary_Rank = 2
```

Though we have 2 Employees with the FIRST highest salary. Dense_Rank() function returns, the next Salary after the tied rows as the SECOND highest Salary

```
WITH Result AS
(
   SELECT Salary, DENSE_RANK() OVER (ORDER BY Salary DESC) AS Salary_Rank
   FROM Employees
)
SELECT TOP 1 Salary FROM Result WHERE Salary_Rank = 2
```

You can also use RANK and DENSE_RANK functions to find the Nth highest Salary among Male or Female employee groups. The following query finds the 3rd highest salary amount paid among the Female employees group

```
WITH Result AS
(
   SELECT Salary, Gender,
        DENSE_RANK() OVER (PARTITION BY Gender ORDER BY Salary DESC)
        AS Salary_Rank
   FROM Employees
)
SELECT TOP 1 Salary FROM Result WHERE Salary_Rank = 3
AND Gender = 'Female'
```

Part 111-Difference between rank dense_rank and row_number in SQL

In this video we will discuss the similarities and **difference between RANK, DENSE_RANK and ROW_NUMBER** functions in SQL Server.
Similarities between RANK, DENSE_RANK and ROW_NUMBER functions

- Returns an increasing integer value starting at 1 based on the ordering of rows imposed by the ORDER BY clause (if there are no ties)
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, the integer value is reset to 1 when the partition changes

We will use the following **Employees** table for the examples in this video

Id	Name	Gender	Salary
1	Mark	Male	6000
2	John	Male	8000
3	Pam	Female	4000
4	Sara	Female	5000
5	Todd	Male	3000

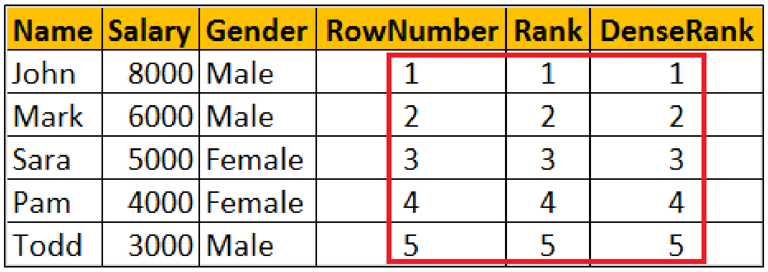
SQL Script to create the Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go
```

```
Insert Into Employees Values (1, 'Mark', 'Male', 6000)
Insert Into Employees Values (2, 'John', 'Male', 8000)
Insert Into Employees Values (3, 'Pam', 'Female', 4000)
Insert Into Employees Values (4, 'Sara', 'Female', 5000)
Insert Into Employees Values (5, 'Todd', 'Male', 3000)
```

Notice that no two employees in the table have the same salary. So all the 3 functions RANK, DENSE_RANK and ROW_NUMBER produce the same increasing integer value when ordered by Salary column.

```
SELECT Name, Salary, Gender,
ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNumber,
RANK() OVER (ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank
FROM Employees
```



Name	Salary	Gender	RowNumber	Rank	DenseRank
John	8000	Male	1	1	1
Mark	6000	Male	2	2	2
Sara	5000	Female	3	3	3
Pam	4000	Female	4	4	4
Todd	3000	Male	5	5	5

You will only see the difference when there ties (duplicate values in the column used in the ORDER BY clause).

Now let's include duplicate values for Salary column.

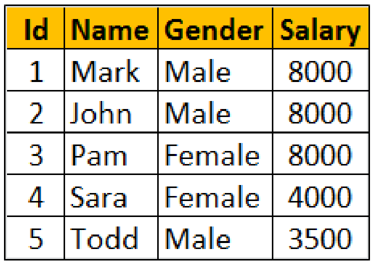
To do this
First delete existing data from the Employees table

```
DELETE FROM Employees
```

Insert new rows with duplicate valuse for Salary column

```
Insert Into Employees Values (1, 'Mark', 'Male', 8000)
Insert Into Employees Values (2, 'John', 'Male', 8000)
Insert Into Employees Values (3, 'Pam', 'Female', 8000)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
```

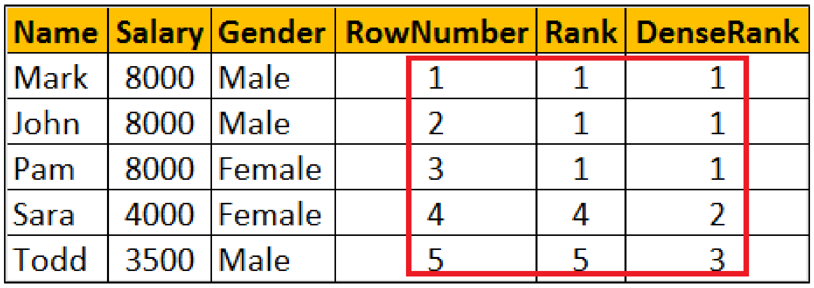
At this point data in the Employees table should be as shown below



Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	8000
4	Sara	Female	4000
5	Todd	Male	3500

Notice 3 employees have the same salary 8000. When you execute the following query you can clearly see the difference between RANK, DENSE_RANK and ROW_NUMBER functions.

```
SELECT Name, Salary, Gender,
ROW_NUMBER() OVER (ORDER BY Salary DESC) AS RowNumber,
RANK() OVER (ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank
FROM Employees
```



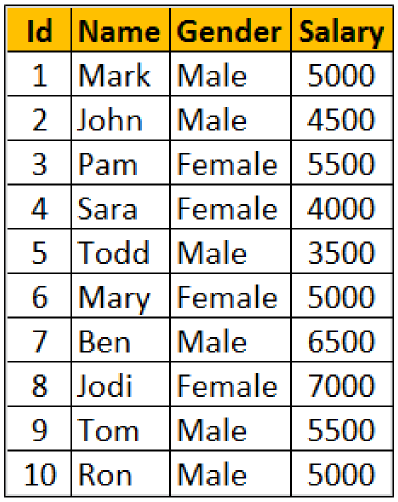
Name	Salary	Gender	RowNumber	Rank	DenseRank
Mark	8000	Male	1	1	1
John	8000	Male	2	1	1
Pam	8000	Female	3	1	1
Sara	4000	Female	4	4	2
Todd	3500	Male	5	5	3

Difference between RANK, DENSE_RANK and ROW_NUMBER functions

- **ROW_NUMBER :** Returns an increasing unique number for each row starting at 1, even if there are duplicates.
- **RANK :** Returns an increasing unique number for each row starting at 1. When there are duplicates, 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 RANK function skips rankings if there are duplicates.
- **DENSE_RANK :** Returns an increasing unique number for each row starting at 1. When there are duplicates, same rank is assigned to all the duplicate rows but the DENSE_RANK function will not skip any ranks. This means the next row after the duplicate rows will have the next rank in the sequence

Part112-Calculate running total in SQL Server 2012

In this video we will discuss how to calculate running total in SQL Server 2012 and later versionsWe will use the following **Employees table** for the examples in this video.



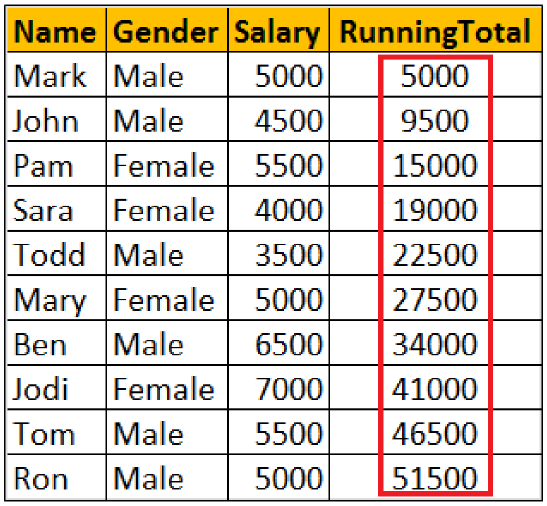
Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

SQL Script to create Employees table

```
Create Table Employees

(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go

Insert Into Employees Values (1, 'Mark', 'Male', 5000)
Insert Into Employees Values (2, 'John', 'Male', 4500)
Insert Into Employees Values (3, 'Pam', 'Female', 5500)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 3500)
Insert Into Employees Values (6, 'Mary', 'Female', 5000)
Insert Into Employees Values (7, 'Ben', 'Male', 6500)
Insert Into Employees Values (8, 'Jodi', 'Female', 7000)
Insert Into Employees Values (9, 'Tom', 'Male', 5500)
Insert Into Employees Values (10, 'Ron', 'Male', 5000)
Go
```

SQL Query to compute running total without partitions

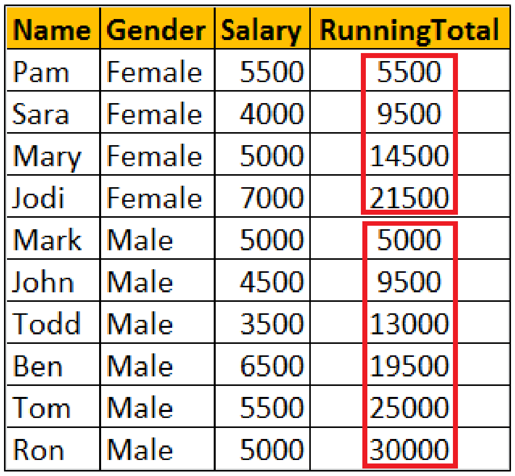
```
SELECT Name, Gender, Salary,
      SUM(Salary) OVER (ORDER BY ID) AS RunningTotal
FROM Employees
```



Name	Gender	Salary	RunningTotal
Mark	Male	5000	5000
John	Male	4500	9500
Pam	Female	5500	15000
Sara	Female	4000	19000
Todd	Male	3500	22500
Mary	Female	5000	27500
Ben	Male	6500	34000
Jodi	Female	7000	41000
Tom	Male	5500	46500
Ron	Male	5000	51500

SQL Query to compute running total with partitions

```
SELECT Name, Gender, Salary,
       SUM(Salary) OVER (PARTITION BY Gender ORDER BY ID) AS RunningTotal
FROM Employees
```



Name	Gender	Salary	RunningTotal
Pam	Female	5500	5500
Sara	Female	4000	9500
Mary	Female	5000	14500
Jodi	Female	7000	21500
Mark	Male	5000	5000
John	Male	4500	9500
Todd	Male	3500	13000
Ben	Male	6500	19500
Tom	Male	5500	25000
Ron	Male	5000	30000

What happens if I use order by on Salary column
If you have duplicate values in the Salary column, all the duplicate values will be added to the running total at once. In the example below notice that we have 5000 repeated 3 times. So 15000 (i.e 5000 + 5000 + 5000) is added to the running total at once.

```
SELECT Name, Gender, Salary,
       SUM(Salary) OVER (ORDER BY Salary) AS RunningTotal

FROM Employees
```

Name	Gender	Salary	RunningTotal
Todd	Male	3500	3500
Sara	Female	4000	7500
John	Male	4500	12000
Mark	Male	5000	27000
Mary	Female	5000	27000
Ron	Male	5000	27000
Tom	Male	5500	38000
Pam	Female	5500	38000
Ben	Male	6500	44500
Jodi	Female	7000	51500

So when computing running total, it is better to use a column that has unique data in the ORDER BY clause

Part113-NTILE function in SQL Server

In this video we will discuss **NTILE function in SQL Server**
Introduced in SQL Server 2005

- ORDER BY Clause is required
- PARTITION BY clause is optional
- Distributes the rows into a specified number of groups
- If the number of rows is not divisible by number of groups, you may have groups of two different sizes.
- Larger groups come before smaller groups
- **For example**

- NTILE(2) of 10 rows divides the rows in 2 Groups (5 in each group)
- NTILE(3) of 10 rows divides the rows in 3 Groups (4 in first group, 3 in 2nd & 3rd group)

Syntax : NTILE (Number_of_Groups) OVER (ORDER BY Col1, Col2, ...)

We will use the following **Employees table** for the examples in this video.

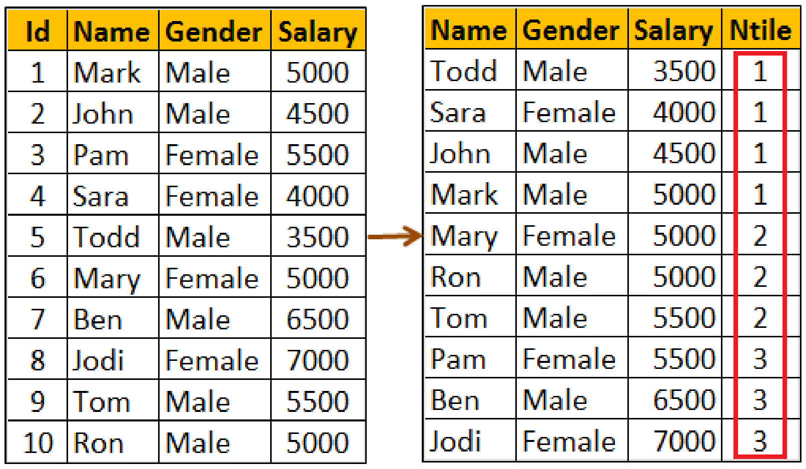
Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

SQL Script to create Employees table

```
Create Table Employees

(

   Id int primary key,

   Name nvarchar(50),

   Gender nvarchar(10),

   Salary int

)

Go


Insert Into Employees Values (1, 'Mark', 'Male', 5000)

Insert Into Employees Values (2, 'John', 'Male', 4500)

Insert Into Employees Values (3, 'Pam', 'Female', 5500)

Insert Into Employees Values (4, 'Sara', 'Female', 4000)

Insert Into Employees Values (5, 'Todd', 'Male', 3500)

Insert Into Employees Values (6, 'Mary', 'Female', 5000)

Insert Into Employees Values (7, 'Ben', 'Male', 6500)

Insert Into Employees Values (8, 'Jodi', 'Female', 7000)

Insert Into Employees Values (9, 'Tom', 'Male', 5500)

Insert Into Employees Values (10, 'Ron', 'Male', 5000)

Go
```

NTILE function without PARTITION BY clause : Divides the 10 rows into 3 groups. 4 rows in first group, 3 rows in the 2nd & 3rd group.

```
SELECT Name, Gender, Salary,
NTILE(3) OVER (ORDER BY Salary) AS [Ntile]

FROM Employees
```

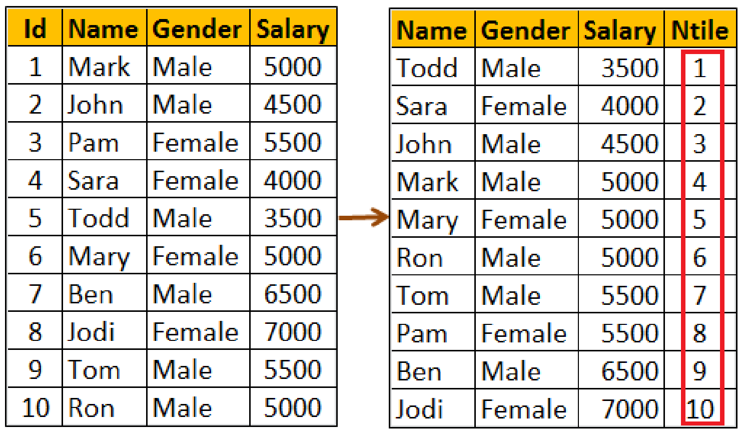


Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

Name	Gender	Salary	Ntile
Todd	Male	3500	1
Sara	Female	4000	1
John	Male	4500	1
Mark	Male	5000	1
Mary	Female	5000	2
Ron	Male	5000	2
Tom	Male	5500	2
Pam	Female	5500	3
Ben	Male	6500	3
Jodi	Female	7000	3

What if the specified number of groups is GREATER THAN the number of rows
NTILE function will try to create as many groups as possible with one row in each group.
With 10 rows in the table, NTILE(11) will create 10 groups with 1 row in each group.

```
SELECT Name, Gender, Salary,
NTILE(11) OVER (ORDER BY Salary) AS [Ntile]

FROM Employees
```



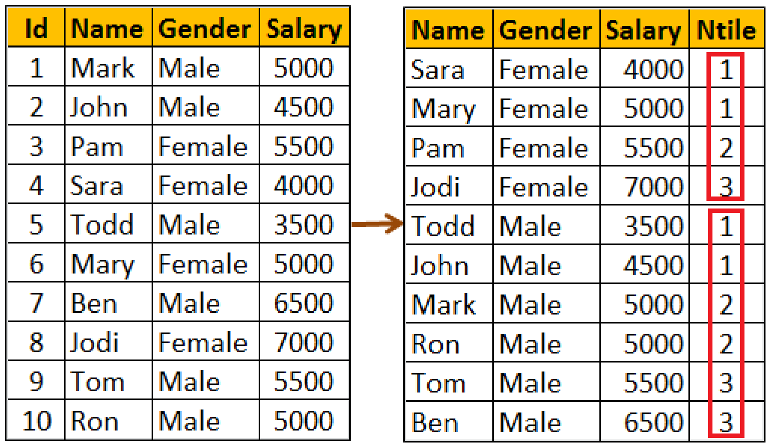
Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

Name	Gender	Salary	Ntile
Todd	Male	3500	1
Sara	Female	4000	2
John	Male	4500	3
Mark	Male	5000	4
Mary	Female	5000	5
Ron	Male	5000	6
Tom	Male	5500	7
Pam	Female	5500	8
Ben	Male	6500	9
Jodi	Female	7000	10

NTILE function with PARTITION BY clause : When the data is partitioned, NTILE function creates the specified number of groups with in each partition.

The following query partitions the data into 2 partitions (Male & Female). NTILE(3) creates 3 groups in each of the partitions.

```
SELECT Name, Gender, Salary,
NTILE(3) OVER (PARTITION BY GENDER ORDER BY Salary) AS [Ntile]

FROM Employees
```



Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

Name	Gender	Salary	Ntile
Sara	Female	4000	1
Mary	Female	5000	1
Pam	Female	5500	2
Jodi	Female	7000	3
Todd	Male	3500	1
John	Male	4500	1
Mark	Male	5000	2
Ron	Male	5000	2
Tom	Male	5500	3
Ben	Male	6500	3

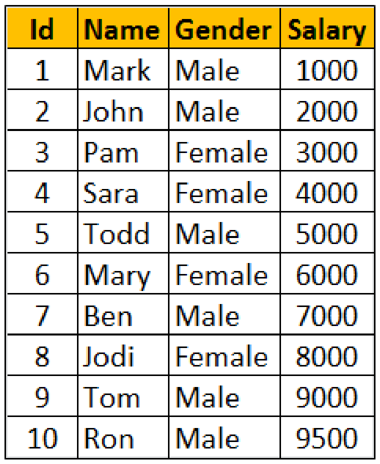
Part114-Lead and Lag functions in SQL Server 2012

In this video we will discuss about Lead and Lag functions.
Lead and Lag functions
Introduced in SQL Server 2012

- Lead function is used to access subsequent row data along with current row data
- Lag function is used to access previous row data along with current row data
- ORDER BY clause is required
- PARTITION BY clause is optional

Syntax
```
LEAD(Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)
LAG(Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)
```

- **Offset -** Number of rows to lead or lag.
- **Default_Value -** The default value to return if the number of rows to lead or lag goes beyond first row or last row in a table or partition. If default value is not specified NULL is returned. We will use the following **Employees table** for the examples in this video



Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

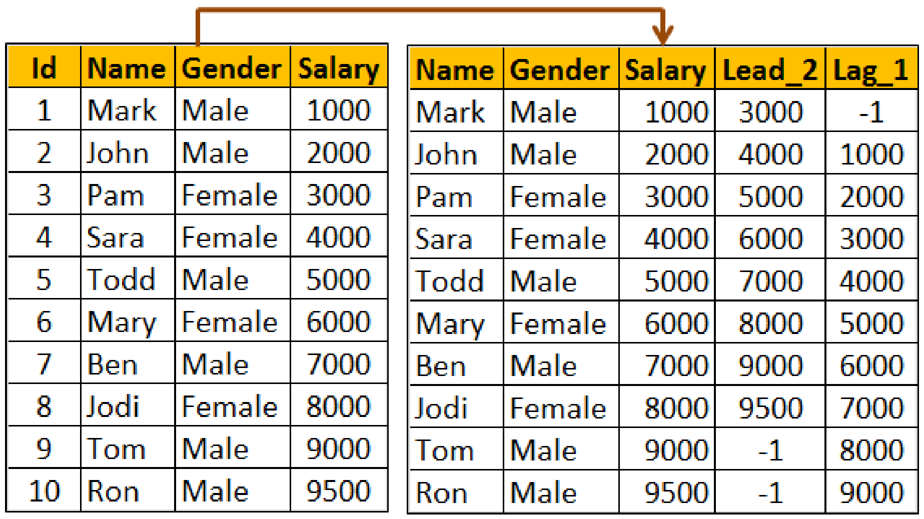
SQL Script to create the Employees table

```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go
Insert Into Employees Values (1, 'Mark', 'Male', 1000)
Insert Into Employees Values (2, 'John', 'Male', 2000)
Insert Into Employees Values (3, 'Pam', 'Female', 3000)
Insert Into Employees Values (4, 'Sara', 'Female', 4000)
Insert Into Employees Values (5, 'Todd', 'Male', 5000)
Insert Into Employees Values (6, 'Mary', 'Female', 6000)
Insert Into Employees Values (7, 'Ben', 'Male', 7000)
Insert Into Employees Values (8, 'Jodi', 'Female', 8000)
Insert Into Employees Values (9, 'Tom', 'Male', 9000)
Insert Into Employees Values (10, 'Ron', 'Male', 9500)
Go
```

Lead and Lag functions example WITHOUT partitions : This example Leads 2 rows and Lags 1 row from the current row.

- When you are on the first row, LEAD(Salary, 2, -1) allows you to move forward 2 rows and retrieve the salary from the 3rd row.
- When you are on the first row, LAG(Salary, 1, -1) allows us to move backward 1 row. Since there no rows beyond row 1, Lag function in this case returns the default value -1.
- When you are on the last row, LEAD(Salary, 2, -1) allows you to move forward 2 rows. Since there no rows beyond the last row 1, Lead function in this case returns the default value -1.
- When you are on the last row, LAG(Salary, 1, -1) allows us to move backward 1 row and retrieve the salary from the previous row.

```
SELECT Name, Gender, Salary,
       LEAD(Salary, 2, -1) OVER (ORDER BY Salary) AS Lead_2,
       LAG(Salary, 1, -1) OVER (ORDER BY Salary) AS Lag_1
FROM Employees
```

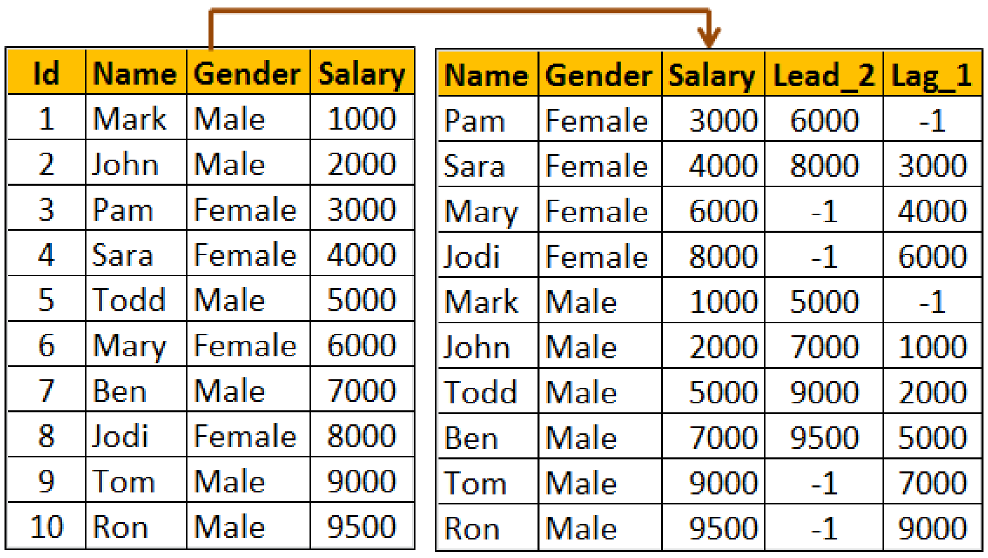


Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Lead_2	Lag_1
Mark	Male	1000	3000	-1
John	Male	2000	4000	1000
Pam	Female	3000	5000	2000
Sara	Female	4000	6000	3000
Todd	Male	5000	7000	4000
Mary	Female	6000	8000	5000
Ben	Male	7000	9000	6000
Jodi	Female	8000	9500	7000
Tom	Male	9000	-1	8000
Ron	Male	9500	-1	9000

Lead and Lag functions example WITH partitions : Notice that in this example, Lead and Lag functions return default value if the number of rows to lead or lag goes beyond first row or last row in the partition.

```
SELECT Name, Gender, Salary,
       LEAD(Salary, 2, -1) OVER (PARTITION By Gender ORDER BY Salary) AS Lead_2,
       LAG(Salary, 1, -1) OVER (PARTITION By Gender ORDER BY Salary) AS Lag_1

FROM Employees
```



Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Lead_2	Lag_1
Pam	Female	3000	6000	-1
Sara	Female	4000	8000	3000
Mary	Female	6000	-1	4000
Jodi	Female	8000	-1	6000
Mark	Male	1000	5000	-1
John	Male	2000	7000	1000
Todd	Male	5000	9000	2000
Ben	Male	7000	9500	5000
Tom	Male	9000	-1	7000
Ron	Male	9500	-1	9000

Part115-FIRST_VALUE function in SQL Server

In this video we will discuss FIRST_VALUE function in SQL Server

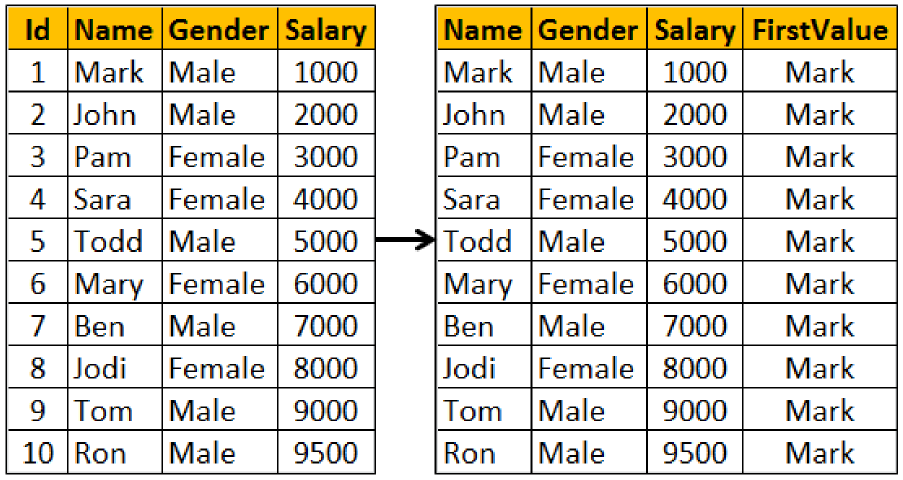
FIRST_VALUE function

- Introduced in SQL Server 2012
- Retrieves the first value from the specified column
- ORDER BY clause is required
- PARTITION BY clause is optional

Syntax : `FIRST_VALUE(Column_Name) OVER (ORDER BY Col1, Col2, ...)`

FIRST_VALUE function example WITHOUT partitions : In the following example, FIRST_VALUE function returns the name of the lowest paid employee from the entire table.

```
SELECT Name, Gender, Salary,
FIRST_VALUE(Name) OVER (ORDER BY Salary) AS FirstValue
FROM Employees
```



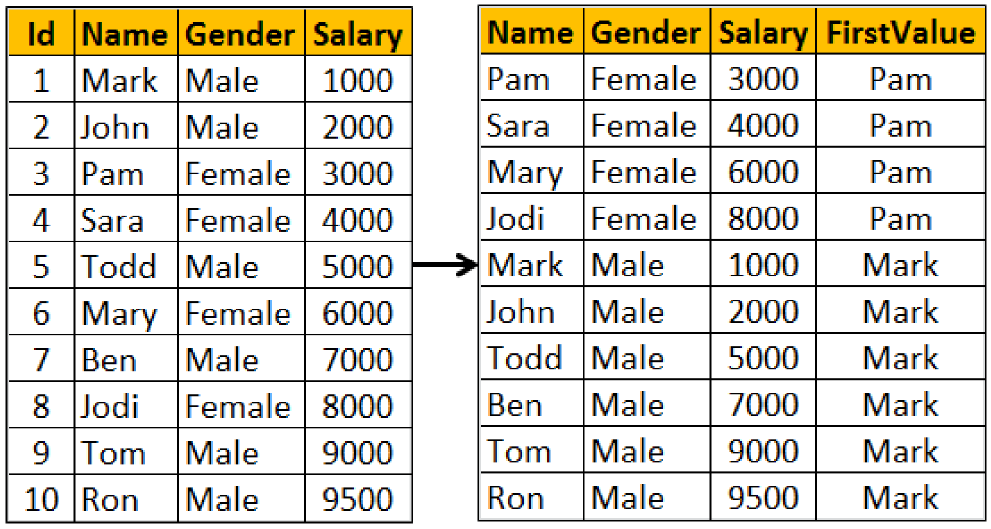
Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	FirstValue
Mark	Male	1000	Mark
John	Male	2000	Mark
Pam	Female	3000	Mark
Sara	Female	4000	Mark
Todd	Male	5000	Mark
Mary	Female	6000	Mark
Ben	Male	7000	Mark
Jodi	Female	8000	Mark
Tom	Male	9000	Mark
Ron	Male	9500	Mark

FIRST_VALUE function example WITH partitions : In the following example, FIRST_VALUE function returns the name of the lowest paid employee from the respective partition.

```
SELECT Name, Gender, Salary,
FIRST_VALUE(Name) OVER (PARTITION BY Gender ORDER BY Salary) AS FirstValue
```

```
FROM Employees
```



Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	FirstValue
Pam	Female	3000	Pam
Sara	Female	4000	Pam
Mary	Female	6000	Pam
Jodi	Female	8000	Pam
Mark	Male	1000	Mark
John	Male	2000	Mark
Todd	Male	5000	Mark
Ben	Male	7000	Mark
Tom	Male	9000	Mark
Ron	Male	9500	Mark

Part116-Window functions in SQL Server

In this video we will discuss **window functions in SQL Server**

In SQL Server we have different categories of window functions

- **Aggregate functions -** AVG, SUM, COUNT, MIN, MAX etc..
- **Ranking functions -** RANK, DENSE_RANK, ROW_NUMBER etc..
- **Analytic functions -** LEAD, LAG, FIRST_VALUE, LAST_VALUE etc...

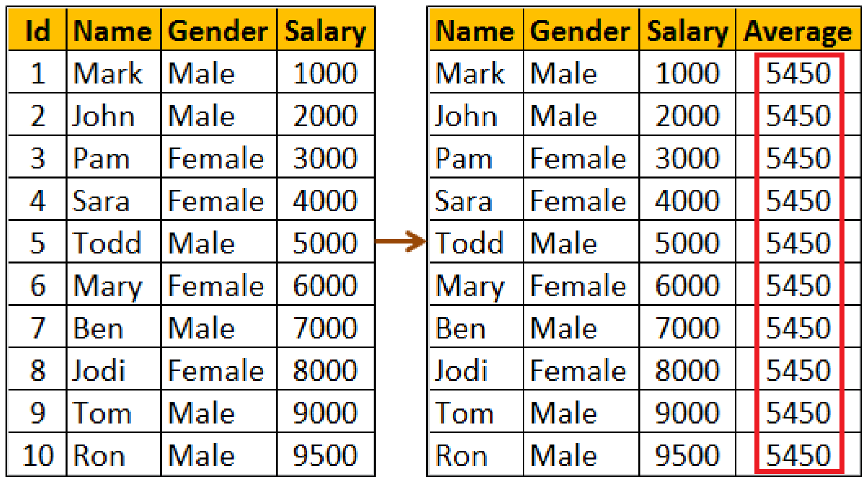
OVER Clause defines the partitioning and ordering of a rows (i.e a window) for the above functions to operate on. Hence these functions are called window functions. The OVER clause accepts the following three arguments to define a window for these functions to operate on.

- **ORDER BY :** Defines the logical order of the rows
- **PARTITION BY :** Divides the query result set into partitions. The window function is applied to each partition separately.
- **ROWSor RANGE clause :** Further limits the rows within the partition by specifying start and end points within the partition.

The default for **ROWS** or **RANGE** clause is
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

Let us understand the use of **ROWS** or **RANGE** clause with an example.

Compute average salary and display it against every employee row as shown below.



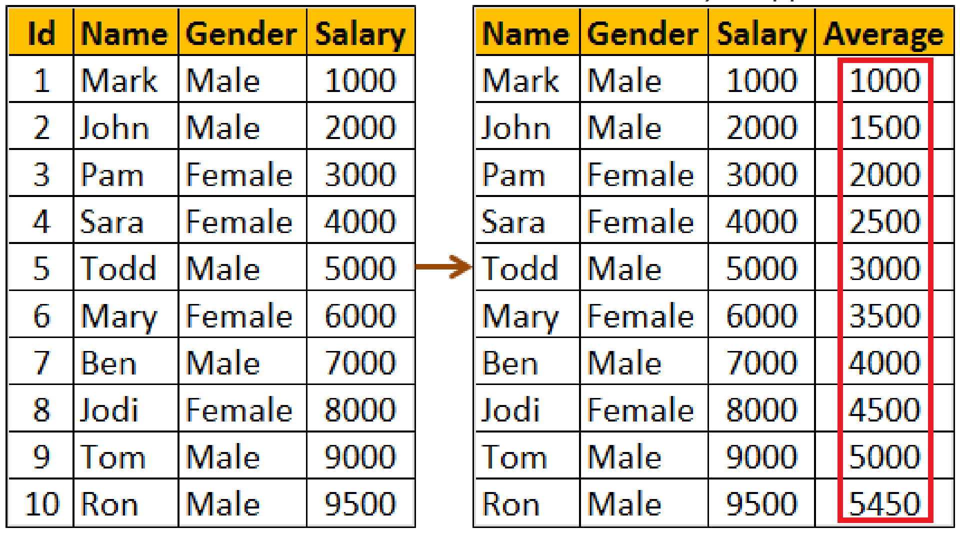
Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Average
Mark	Male	1000	5450
John	Male	2000	5450
Pam	Female	3000	5450
Sara	Female	4000	5450
Todd	Male	5000	5450
Mary	Female	6000	5450
Ben	Male	7000	5450
Jodi	Female	8000	5450
Tom	Male	9000	5450
Ron	Male	9500	5450

We might think the following query would do the job.

```
SELECT Name, Gender, Salary,
        AVG(Salary) OVER(ORDER BY Salary) AS Average
FROM Employees
```

As you can see from the result below, the above query does not produce the overall salary average. It produces the average of the current row and the rows preceeding the current row. This is because, the default value of ROWS or RANGE clause (RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) is applied.

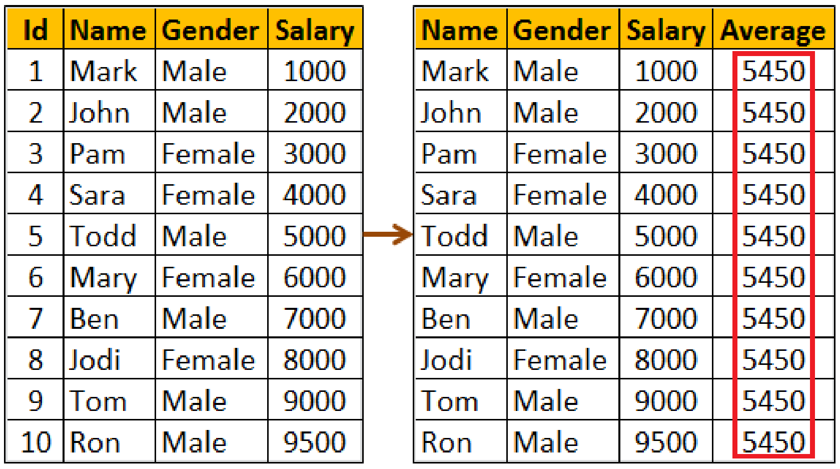


Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Average
Mark	Male	1000	1000
John	Male	2000	1500
Pam	Female	3000	2000
Sara	Female	4000	2500
Todd	Male	5000	3000
Mary	Female	6000	3500
Ben	Male	7000	4000
Jodi	Female	8000	4500
Tom	Male	9000	5000
Ron	Male	9500	5450

To fix this, provide an explicit value for ROWS or RANGE clause as shown below. ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING tells the window function to operate on the set of rows starting from the first row in the partition to the last row in the partition.

```
SELECT Name, Gender, Salary,
        AVG(Salary) OVER(ORDER BY Salary ROWS BETWEEN
        UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS Average
FROM Employees
```



Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

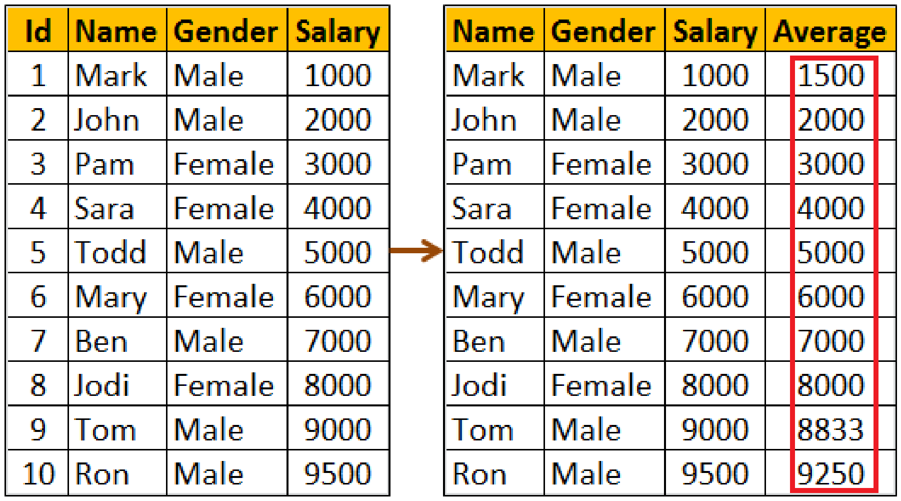
Name	Gender	Salary	Average
Mark	Male	1000	5450
John	Male	2000	5450
Pam	Female	3000	5450
Sara	Female	4000	5450
Todd	Male	5000	5450
Mary	Female	6000	5450
Ben	Male	7000	5450
Jodi	Female	8000	5450
Tom	Male	9000	5450
Ron	Male	9500	5450

The same result can also be achieved by using RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

What is the difference between ROWS and RANGE
We will discuss this in a later video

The following query can be used if you want to compute the average salary of
1. The current row
2. One row PRECEDING the current row and
3. One row FOLLOWING the current row

```
SELECT Name, Gender, Salary,
       AVG(Salary) OVER(ORDER BY Salary
       ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS Average
FROM Employees
```



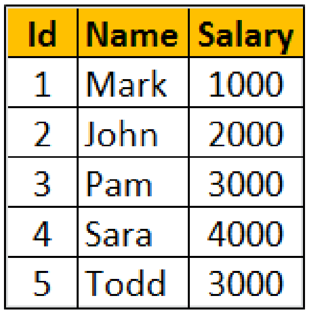
Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	Average
Mark	Male	1000	1500
John	Male	2000	2000
Pam	Female	3000	3000
Sara	Female	4000	4000
Todd	Male	5000	5000
Mary	Female	6000	6000
Ben	Male	7000	7000
Jodi	Female	8000	8000
Tom	Male	9000	8833
Ron	Male	9500	9250

Part117-Difference between rows and range

In this video we will discuss the **difference between rows and range in SQL Server**. This is continuation to Part 116. Please watch Part 116 from SQL Server tutorial before proceeding.

Let us understand the difference with an example. We will use the following **Employees** table in this demo.

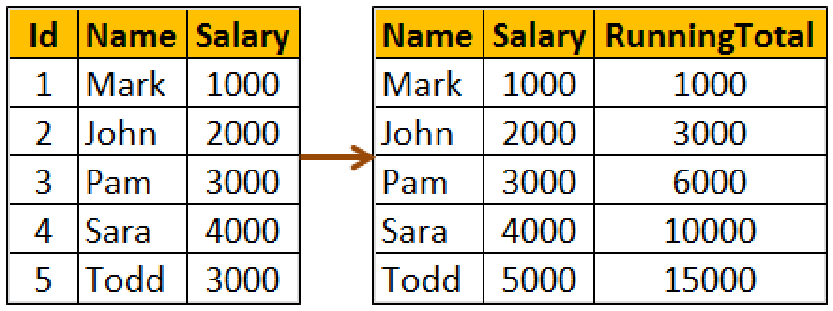


Id	Name	Salary
1	Mark	1000
2	John	2000
3	Pam	3000
4	Sara	4000
5	Todd	3000

SQL Script to create the Employees table

```
Create Table Employees

(

    Id int primary key,

    Name nvarchar(50),

    Salary int

)

Go


Insert Into Employees Values (1, 'Mark', 1000)

Insert Into Employees Values (2, 'John', 2000)

Insert Into Employees Values (3, 'Pam', 3000)

Insert Into Employees Values (4, 'Sara', 4000)

Insert Into Employees Values (5, 'Todd', 5000)

Go
```

Calculate the running total of Salary and display it against every employee row



Id	Name	Salary
1	Mark	1000
2	John	2000
3	Pam	3000
4	Sara	4000
5	Todd	3000

Name	Salary	RunningTotal
Mark	1000	1000
John	2000	3000
Pam	3000	6000
Sara	4000	10000
Todd	5000	15000

The following query calculates the running total. We have not specified an explicit value for ROWS or RANGE clause.

```
SELECT Name, Salary,

      SUM(Salary) OVER(ORDER BY Salary) AS RunningTotal

FROM Employees
```

So the above query is using the default value which is

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

This means the above query can be re-written using an explicit value for ROWS or RANGE clause as shown below.

```
SELECT Name, Salary,

      SUM(Salary) OVER(ORDER BY Salary

      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
RunningTotal

FROM Employees
```

We can also achieve the same result, by replacing RANGE with ROWS

```
SELECT Name, Salary,

      SUM(Salary) OVER(ORDER BY Salary

      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
RunningTotal

FROM Employees
```

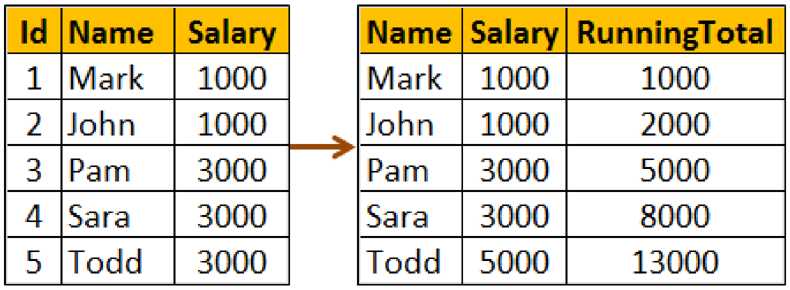
What is the difference between ROWS and RANGE
To understand the difference we need some duplicate values for the Salary column in the Employees table.

Execute the following UPDATE script to introduce duplicate values in the Salary column

```
Update Employees set Salary = 1000 where Id = 2

Update Employees set Salary = 3000 where Id = 4

Go
```

Now execute the following query. Notice that we get the running total as expected.

```
SELECT Name, Salary,

       SUM(Salary) OVER(ORDER BY Salary

       ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
RunningTotal

FROM Employees
```



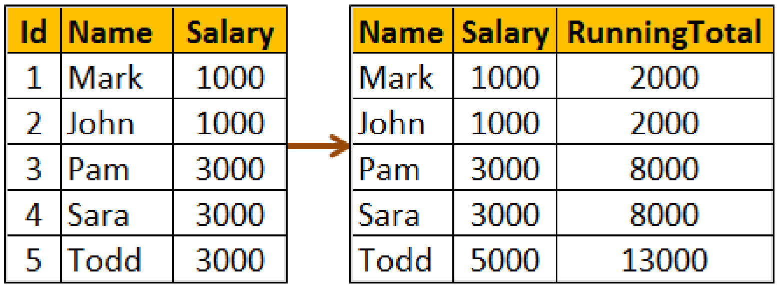
Id	Name	Salary
1	Mark	1000
2	John	1000
3	Pam	3000
4	Sara	3000
5	Todd	3000

Name	Salary	RunningTotal
Mark	1000	1000
John	1000	2000
Pam	3000	5000
Sara	3000	8000
Todd	5000	13000

The following query uses RANGE instead of ROWS

```
SELECT Name, Salary,

       SUM(Salary) OVER(ORDER BY Salary

       RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
RunningTotal

FROM Employees
```

You get the following result when you execute the above query. Notice we don't get the running total as expected.



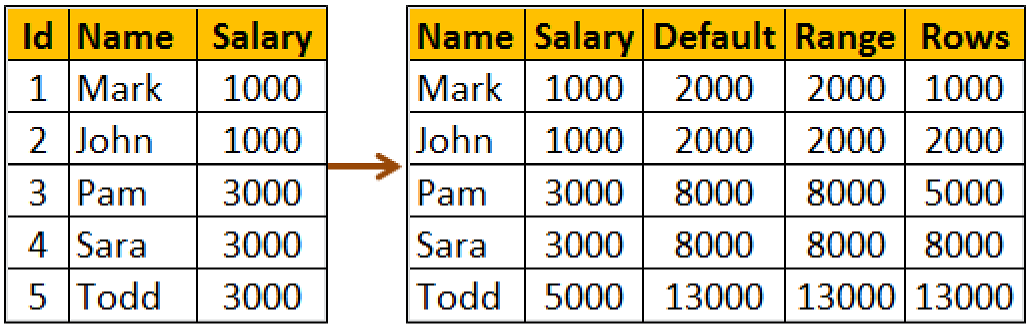
Id	Name	Salary
1	Mark	1000
2	John	1000
3	Pam	3000
4	Sara	3000
5	Todd	3000

Name	Salary	RunningTotal
Mark	1000	2000
John	1000	2000
Pam	3000	8000
Sara	3000	8000
Todd	5000	13000

So, the main difference between ROWS and RANGE is in the way duplicate rows are treated. ROWS treat duplicates as distinct values, where as RANGE treats them as a single entity.

All together side by side. The following query shows how running total changes
1. When no value is specified for ROWS or RANGE clause
2. When RANGE clause is used explicitly with it's default value
3. When ROWS clause is used instead of RANGE clause

```
SELECT Name, Salary,
       SUM(Salary) OVER(ORDER BY Salary) AS [Default],

       SUM(Salary) OVER(ORDER BY Salary

       RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS [Range],

       SUM(Salary) OVER(ORDER BY Salary

       ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS [Rows]

FROM Employees
```



Id	Name	Salary
1	Mark	1000
2	John	1000
3	Pam	3000
4	Sara	3000
5	Todd	3000

Name	Salary	Default	Range	Rows
Mark	1000	2000	2000	1000
John	1000	2000	2000	2000
Pam	3000	8000	8000	5000
Sara	3000	8000	8000	8000
Todd	5000	13000	13000	13000