Connecting to SQL Server using SSMS - Part 1

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

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

You will now see, Connect to Server window.

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

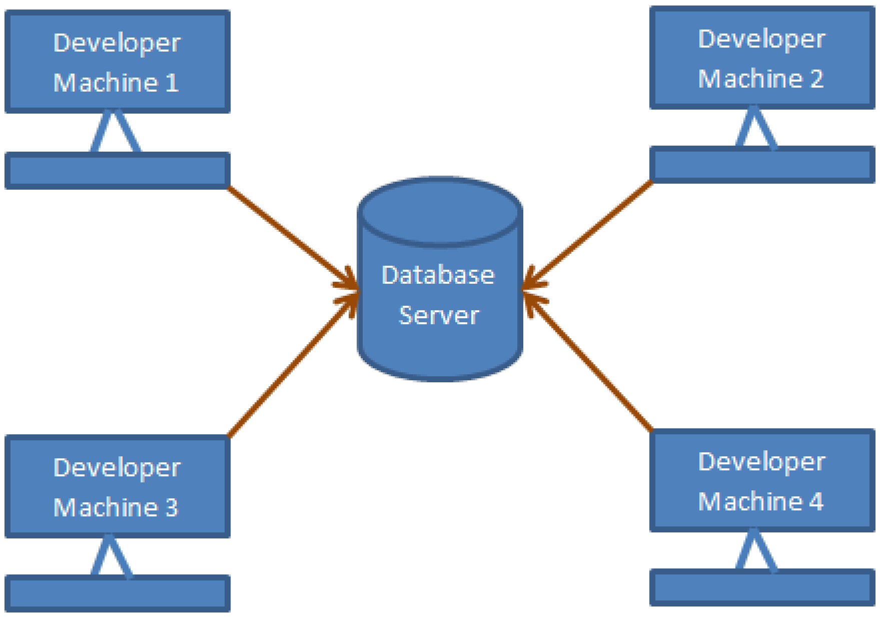
2. Next you need to specify the Server Name. Here we can specify the name or the server or IP Address.If you have SQL Server installed on your local machine, you can specify, (local) or just. (Period) or 127.0.0.1
Server name = (local)

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

4. If you have chosen Windows Authentication, you dont have to enter user name and password, otherwise enter the user name and password and click connect.

You should now be connected to SQL Server. Now, click on **New Query**, on the top left hand corner of SSMS. This should open a new query editor window, where we can type sql queries and execute.

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



Creating, altering and dropping a database - Part 2

In Part 1 of SQL Server, we have seen, using SSMS to connect to SQL Server. In this part we will learn creating, altering and dropping a database.

A SQL Server database can be created, altered and dropped
1. Graphically using SQL Server Management Studio (SSMS) or
2. Using a Query

To create the database graphically
1. Right Click on Databases folder in the Object explorer
2. Select New Database
3. In the New Database dialog box, enter the Database name and click OK.

To Create the database using a query
```
Create database DatabaseName
```

Whether, you create a database graphically using the designer or, using a query, the following 2 files gets generated.
.MDF file - Data File (Contains actual data)
.LDF file - Transaction Log file (Used to recover the database)

To alter a database, once it's created
```
Alter database DatabaseName Modify Name = NewDatabaseName
```

Alternatively, you can also use system stored procedure
```
Execute sp_renameDB 'OldDatabaseName','NewDatabaseName'
```

To Delete or Drop a database
Drop Database DatabaseThatYouWantToDrop

Dropping a database, deletes the LDF and MDF files.

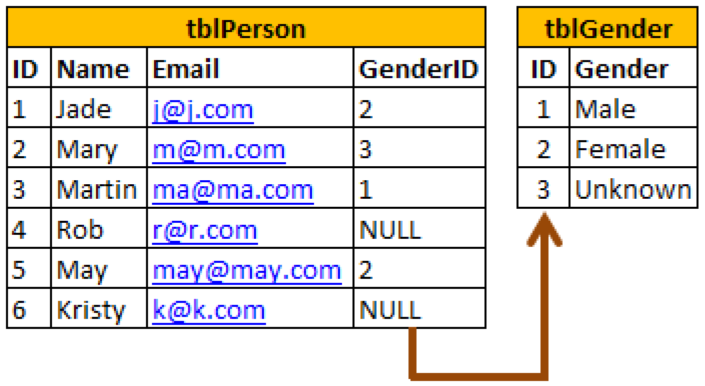
You cannot drop a database, if it is currently in use. You get an error stating - Cannot drop database "NewDatabaseName" because it is currently in use. So, if other users are connected, you need to put the database in single user mode and then drop the database.
Alter Database DatabaseName Set SINGLE_USER With Rollback Immediate

With Rollback Immediate option, will rollback all incomplete transactions and closes the connection to the database.

Note: System databases cannot be dropped.

Creating and Working with tables - Part 3

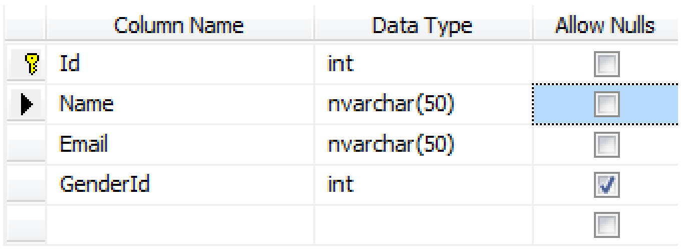
The aim of this article is to create tblPerson and tblGender tables and establish primary key and foreign key constraints. In SQL Server, tables can be created graphically using SQL Server Management Studio (SSMS) or using a query.



tblPerson			
ID	Name	Email	GenderID
1	Jade	j@j.com	2
2	Mary	m@m.com	3
3	Martin	ma@ma.com	1
4	Rob	r@r.com	NULL
5	May	may@may.com	2
6	Kristy	k@k.com	NULL

tblGender	
ID	Gender
1	Male
2	Female
3	Unknown

To create tblPerson table, graphically, using SQL Server Management Studio
1. Right click on Tables folder in Object explorer window
2. Select New Table
3. Fill Column Name, Data Type and Allow Nulls, as shown below and save the table as tblPerson.



	Column Name	Data Type	Allow Nulls
🔑	Id	int	☐
▶	Name	nvarchar(50)	☐
	Email	nvarchar(50)	☐
	GenderId	int	☑
			☐

The following statement creates **tblGender** table, with **ID** and **Gender** columns. The following statement creates tblGender table, with ID and Gender columns. **ID** column, is the **primary key** column. The primary key is used to uniquely identify each row in a table. Primary key does not allow nulls.

```
Create Table tblGender
(ID int Not Null Primary Key,
Gender nvarchar(50))
```

In **tblPerson** table, **GenderID** is the **foreign key** referencing **ID** column in **tblGender** table. Foreign key references can be added graphically using SSMS or using a query.

To graphically add a foreign key reference
1. Right click tblPerson table and select Design
2. In the table design window, right click on GenderId column and select Relationships
3. In the Foreign Key Relationships window, click Add button
4. Now expand, in Tables and Column Specification row, by clicking the, + sign
5. Click on the elipses button, that is present in Tables and Column Specification row
6. From the Primary Key Table, dropdownlist, select tblGender
7. Click on the row below, and select ID column
8. From the column on the right hand side, select GenderId
9. Click OK and then click close.
10. Finally save the table.

To add a foreign key reference using a query

```
Alter table tblPerson
add constraint tblPerson_GenderId_FK FOREIGN KEY (GenderId) references tblGender(ID)
```

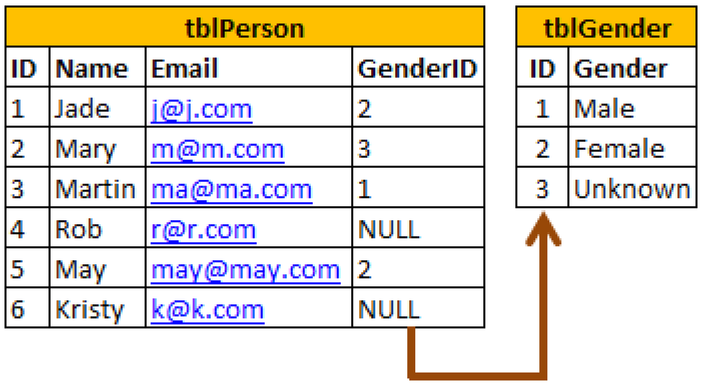
The general formula is here

```
Alter table ForeignKeyTable add constraint ForeignKeyTable_ForiegnKeyColumn_FK
FOREIGN KEY (ForiegnKeyColumn) references PrimaryKeyTable (PrimaryKeyColumn)
```

Foreign keys are used to enforce **database integrity**. In layman's terms, A **foreign key** in one table points to a **primary key** in another table. The foreign key constraint prevents invalid data form being inserted into the foreign key column. The values that you enter into the foreign key column, has to be one of the values contained in the table it points to.

Default constraint in sql server - Part 4

In Part 3 of this video series, we have seen how to create tables (tblPerson and tblGender) and enforce primary and foreign key constraints. Please watch Part 3, before continuing with this session.



tblPerson			
ID	Name	Email	GenderID
1	Jade	j@j.com	2
2	Mary	m@m.com	3
3	Martin	ma@ma.com	1
4	Rob	r@r.com	NULL
5	May	may@may.com	2
6	Kristy	k@k.com	NULL

tblGender	
ID	Gender
1	Male
2	Female
3	Unknown

In this video, we will learn adding a Default Constraint. A column default can be specified using Default constraint. The default constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified, including NULL.

Altering an existing column to add a default constraint:

```
ALTER TABLE { TABLE_NAME }
ADD CONSTRAINT { CONSTRAINT_NAME }
DEFAULT { DEFAULT_VALUE } FOR { EXISTING_COLUMN_NAME }
```

Adding a new column, with default value, to an existing table:

```
ALTER TABLE { TABLE_NAME }
ADD { COLUMN_NAME } { DATA_TYPE } { NULL | NOT NULL }
CONSTRAINT { CONSTRAINT_NAME } DEFAULT { DEFAULT_VALUE }
```

The following command will add a default constraint, DF_tblPerson_GenderId.

```
ALTER TABLE tblPerson
ADD CONSTRAINT DF_tblPerson_GenderId
DEFAULT 1 FOR GenderId
```

The insert statement below does not provide a value for GenderId column, so the default of 1 will be inserted for this record.

```
Insert into tblPerson(ID,Name,Email) values(5,'Sam','s@s.com')
```

On the other hand, the following insert statement will insert NULL, instead of using the default.

```
Insert into tblPerson(ID,Name,Email,GenderId) values (6,'Dan','d@d.com',NULL)
```

To drop a constraint
ALTER TABLE { TABLE_NAME }
DROP CONSTRAINT { CONSTRAINT_NAME }

In the next session, we will learn about cascading referential integrity

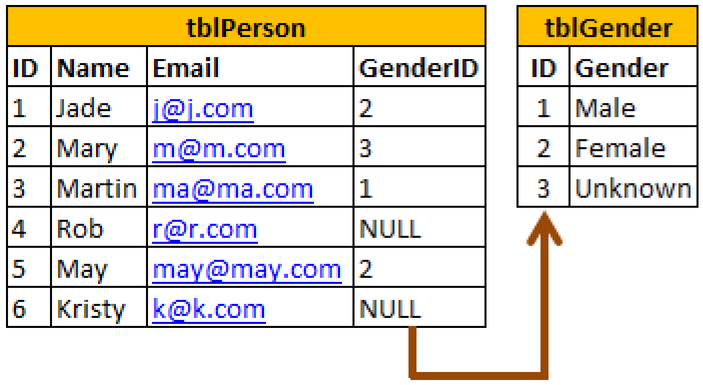
Cascading referential integrity constraint - Part 5

In Part 3 of this video series, we have seen how to create tables (tblPerson and tblGender) and enforce primary and foreign key constraints. In Part 4, we have learnt adding a default constraint. Please watch Parts 3 and 4, before continuing with this session.

In this video, we will learn about **Cascading referential integrity constraint**

Cascading referential integrity constraint allows to define the actions Microsoft SQL Server should take when a user attempts to delete or update a key to which an existing foreign keys points.

For example, consider the 2 tables shown below. If you delete row with **ID = 1** from **tblGender** table, then row with **ID = 3** from **tblPerson** table becomes an **orphan record**. You will not be able to tell the Gender for this row. So, Cascading referential integrity constraint can be used to define actions Microsoft SQL Server should take when this happens. By default, we get an error and the DELETE or UPDATE statement is rolled back.



tblPerson			
ID	Name	Email	GenderID
1	Jade	j@j.com	2
2	Mary	m@m.com	3
3	Martin	ma@ma.com	1
4	Rob	r@r.com	NULL
5	May	may@may.com	2
6	Kristy	k@k.com	NULL

tblGender	
ID	Gender
1	Male
2	Female
3	Unknown

However, you have the following options when setting up Cascading referential integrity constraint
1. No Action: This is the default behaviour. No Action specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, an error is raised and the DELETE or UPDATE is rolled back.

2. Cascade: Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are also deleted or updated.

3. Set NULL: Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to NULL.

4. Set Default: Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to default values.

Check constraint in SQL Server - Part 6

CHECK constraint is used to **limit the range of the values**, that can be entered for a column.

Let's say, we have an integer AGE column, in a table. The AGE in general cannot be less than ZERO and at the same time cannot be greater than 150. But, since AGE is an integer column it can accept negative values and values much greater than 150.

So, to limit the values, that can be added, we can use CHECK constraint. In SQL Server, CHECK constraint can be created graphically, or using a query.

The following check constraint, limits the age between ZERO and 150.

```
ALTER TABLE tblPerson
ADD CONSTRAINT CK_tblPerson_Age CHECK (Age > 0 AND Age < 150)
```

The general formula for adding check constraint in SQL Server:

```
ALTER TABLE { TABLE_NAME }
ADD CONSTRAINT { CONSTRAINT_NAME } CHECK ( BOOLEAN_EXPRESSION )
```

If the BOOLEAN_EXPRESSION returns true, then the CHECK constraint allows the value, otherwise it doesn't. Since, AGE is a nullable column, it's possible to pass null for this column, when inserting a row. When you pass NULL for the AGE column, the boolean expression evaluates to **UNKNOWN**, and allows the value.

To drop the CHECK constraint:

```
ALTER TABLE tblPerson
DROP CONSTRAINT CK_tblPerson_Age
```

Identity column in SQL Server - Part 7

If a column is marked as an identity column, then the values for this column are automatically generated, when you insert a new row into the table. The following, create table statement marks PersonId as an identity column with seed = 1 and Identity Increment = 1. Seed and Increment values are optional. If you don't specify the identity and seed they both default to 1.

```
Create Table tblPerson
(
PersonId int Identity(1,1) Primary Key,
```

Name nvarchar(20)
)

In the following 2 insert statements, we only supply values for Name column and not for PersonId column.
Insert into tblPerson values ('Sam')
Insert into tblPerson values ('Sara')

If you select all the rows from tblPerson table, you will see that, 'Sam' and 'Sara' rows have got 1 and 2 as PersonId.

Now, if I try to execute the following query, I get an error stating - An explicit value for the identity column in table 'tblPerson' can only be specified when a column list is used and IDENTITY_INSERT is ON.
Insert into tblPerson values (1,'Todd')

So if you mark a column as an Identity column, you dont have to explicitly supply a value for that column when you insert a new row. The value is automatically calculated and provided by SQL server. So, to insert a row into tblPerson table, just provide value for Name column.
Insert into tblPerson values ('Todd')

Delete the row, that you have just inserted and insert another row. You see that the value for PersonId is 2. Now if you insert another row, PersonId is 3. A record with PersonId = 1, does not exist, and I want to fill this gap. To do this, we should be able to explicitly supply the value for identity column. To explicitly supply a value for identity column
1. First turn on identity insert - SET Identity_Insert tblPerson ON
2. In the insert query specify the column list
   Insert into tblPerson(PersonId, Name) values(2, 'John')

As long as the Identity_Insert is turned on for a table, you need to explicitly provide the value for that column. If you don't provide the value, you get an error - Explicit value must be specified for identity column in table 'tblPerson1' either when IDENTITY_INSERT is set to ON or when a replication user is inserting into a NOT FOR REPLICATION identity column.

After, you have the gaps in the identity column filled, and if you wish SQL server to calculate the value, turn off Identity_Insert.
SET Identity_Insert tblPerson OFF

If you have deleted all the rows in a table, and you want to reset the identity column value, use DBCC CHECKIDENT command. This command will reset PersonId identity column.
DBCC CHECKIDENT(tblPerson, RESEED, 0)

How to get the last generated identity column value in SQL Server - Part 8

From the previous session, we understood that identity column values are auto generated. There are several ways in sql server, to retrieve the last identity value that is generated. The most common way is to use SCOPE_IDENTITY() built in function.

Apart, from using SCOPE_IDENTITY(), you also have @@IDENTITY and IDENT_CURRENT('TableName') function.

Example queries for getting the last generated identity value
Select SCOPE_IDENTITY()

```
Select @@IDENTITY
Select IDENT_CURRENT('tblPerson')
```

Let's now understand the difference between, these 3 approaches.

SCOPE_IDENTITY() returns the last identity value that is created in the same session (Connection) and in the same scope (in the same Stored procedure, function, trigger). Let's say, I have 2 tables tblPerson1 and tblPerson2, and I have a trigger on tblPerson1 table, which will insert a record into tblPerson2 table. Now, when you insert a record into tblPerson1 table, SCOPE_IDENTITY() returns the idetentity value that is generated in tblPerson1 table, where as @@IDENTITY returns, the value that is generated in tblPerson2 table. So, @@IDENTITY returns the last identity value that is created in the same session without any consideration to the scope. IDENT_CURRENT('tblPerson') returns the last identity value created for a specific table across any session and any scope.

In brief:
SCOPE_IDENTITY() - returns the last identity value that is created in the same session and in the same scope.
@@IDENTITY - returns the last identity value that is created in the same session and across any scope.
IDENT_CURRENT('TableName') - returns the last identity value that is created for a specific table across any session and any scope.

Unique key constraint - Part 9

We use UNIQUE constraint to enforce uniqueness of a column i.e the column shouldn't allow any duplicate values. We can add a Unique constraint thru the designer or using a query.
To add a unique constraint using SQL server management studio designer:
1. Right-click on the table and select Design
2. Right-click on the column, and select Indexes/Keys...
3. Click Add
4. For Columns, select the column name you want to be unique.
5. For Type, choose Unique Key.
6. Click Close, Save the table.

To create the unique key using a query:

```
Alter Table Table_Name
Add Constraint Constraint_Name Unique(Column_Name)
```

Both primary key and unique key are used to enforce, the uniqueness of a column. So, when do you choose one over the other?
A table can have, only one primary key. If you want to enforce uniqueness on 2 or more columns, then we use unique key constraint.

What is the difference between Primary key constraint and Unique key constraint? This question is asked very frequently in interviews.
1. A table can have only one primary key, but more than one unique key
2. Primary key does not allow nulls, where as unique key allows one null

To drop the constraint
1. Right click the constraint and delete.
Or

2. Using a query
Alter Table tblPerson
Drop COnstraint UQ_tblPerson_Email

Select statement - Part 10

Basic select statement syntax
SELECT Column_List
FROM Table_Name

If you want to select all the columns, you can also use *. For better performance use the column list, instead of using *.
SELECT *
FROM Table_Name

To Select distinct rows use DISTINCT keyword
SELECT DISTINCT Column_List
FROM Table_Name

Example: Select distinct city from tblPerson

Filtering rows with WHERE clause
SELECT Column_List
FROM Table_Name
WHERE Filter_Condition

Example: Select Name, Email from tblPerson where City = 'London'

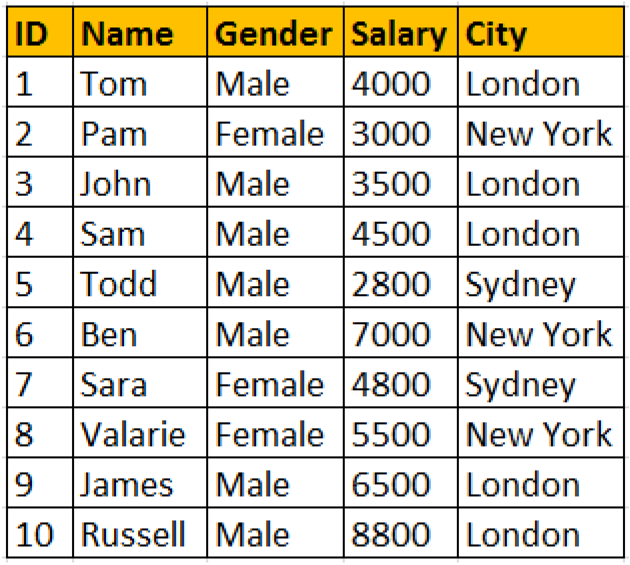
Note: Text values, should be present in single quotes, but not required for numeric values.

Different operators that can be used in a where clause

Group By - Part 11

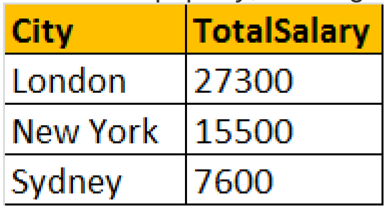
In SQL Server we have got lot of aggregate functions. Examples
1. Count()
2. Sum()
3. avg()
4. Min()
5. Max()

Group by clause is used to group a selected set of rows into a set of summary rows by the values of one or more columns or expressions. It is always used in conjunction with one or more aggregate functions.



ID	Name	Gender	Salary	City
1	Tom	Male	4000	London
2	Pam	Female	3000	New York
3	John	Male	3500	London
4	Sam	Male	4500	London
5	Todd	Male	2800	Sydney
6	Ben	Male	7000	New York
7	Sara	Female	4800	Sydney
8	Valarie	Female	5500	New York
9	James	Male	6500	London
10	Russell	Male	8800	London

I want an sql query, which gives total salaries paid by City. The output should be as shown below.



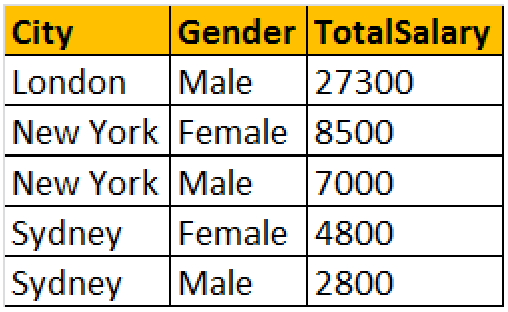
City	TotalSalary
London	27300
New York	15500
Sydney	7600

Query for retrieving total salaries by city:
We are applying SUM() aggregate function on Salary column, and grouping by city column. This effectively adds, all salaries of employees with in the same city.

```
Select City, SUM(Salary) as TotalSalary
from tblEmployee
Group by City
```

Note: If you omit, the group by clause and try to execute the query, you get an error - Column 'tblEmployee.City' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

Now, I want an sql query, which gives total salaries by City, by gender. The output should be as shown below.



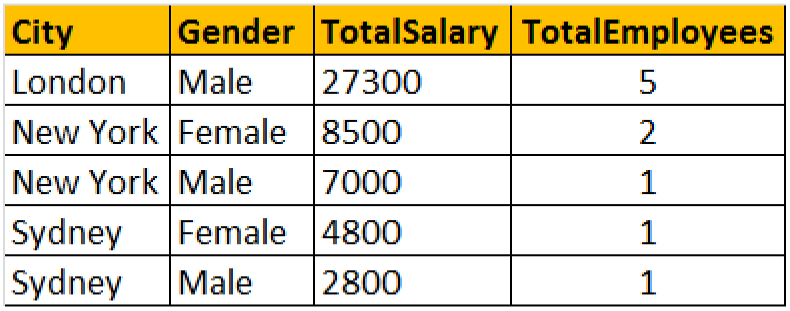
City	Gender	TotalSalary
London	Male	27300
New York	Female	8500
New York	Male	7000
Sydney	Female	4800
Sydney	Male	2800

Query for retrieving total salaries by city and by gender: It's possible to group by multiple columns. In this query, we are grouping first by city and then by gender.

```
Select City, Gender, SUM(Salary) as TotalSalary
```

```
from tblEmployee
group by City, Gender
```

Now, I want an sql query, which gives total salaries and total number of employees by City, and by gender. The output should be as shown below.



City	Gender	TotalSalary	TotalEmployees
London	Male	27300	5
New York	Female	8500	2
New York	Male	7000	1
Sydney	Female	4800	1
Sydney	Male	2800	1

Query for retrieving total salaries and total number of employees by City, and by gender:
The only difference here is that, we are using Count() aggregate function.

```
Select City, Gender, SUM(Salary) as TotalSalary,
COUNT(ID) as TotalEmployees
from tblEmployee
group by City, Gender
```

Filtering Groups:
WHERE clause is used to filter rows before aggregation, where as HAVING clause is used to filter groups after aggregations. The following 2 queries produce the same result.

Filtering rows using WHERE clause, before aggrgations take place:

```
Select City, SUM(Salary) as TotalSalary
from tblEmployee
Where City = 'London'
group by City
```

Filtering groups using HAVING clause, after all aggrgations take place:

```
Select City, SUM(Salary) as TotalSalary
from tblEmployee
group by City
Having City = 'London'
```

From a performance standpoint, you cannot say that one method is less efficient than the other. Sql server optimizer analyzes each statement and selects an efficient way of executing it. As a best practice, use the syntax that clearly describes the desired result. Try to eliminate rows that you wouldn't need, as early as possible.

It is also possible to combine WHERE and HAVING

```
Select City, SUM(Salary) as TotalSalary
from tblEmployee
Where Gender = 'Male'
group by City
Having City = 'London'
```

Difference between WHERE and HAVING clause:
1. WHERE clause can be used with - Select, Insert, and Update statements, where as HAVING

clause can only be used with the Select statement.
2. WHERE filters rows before aggregation (GROUPING), where as, HAVING filters groups, after the aggregations are performed.
3. Aggregate functions cannot be used in the WHERE clause, unless it is in a sub query contained in a HAVING clause, whereas, aggregate functions can be used in Having clause.

Joins in sql server - Part 12

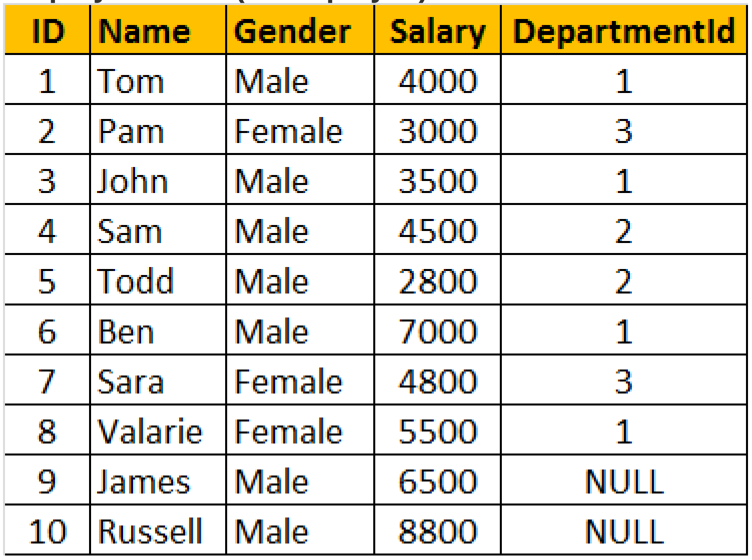
Joins in SQL server are used to query (retrieve) data from 2 or more related tables. In general tables are related to each other using foreign key constraints.

Please watch Parts 3 and 5 in this video series, before continuing with this video.
Part 3 - Creating and working with tables
Part 5 - Cascading referential integrity constraint

In SQL server, there are different types of JOINS.
1. CROSS JOIN
2. INNER JOIN
3. OUTER JOIN

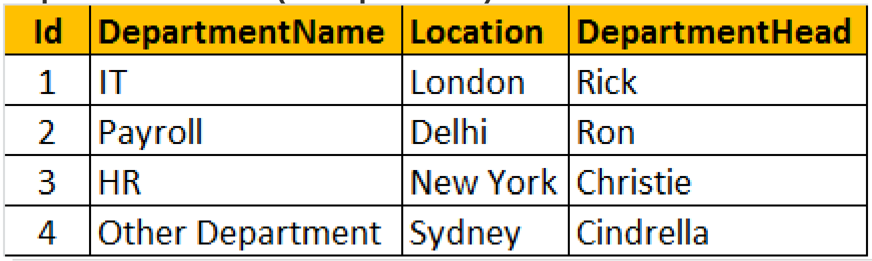
Outer Joins are again divided into 3 types
1. Left Join or Left Outer Join
2. Right Join or Right Outer Join
3. Full Join or Full Outer Join

Now let's understand all the JOIN types, with examples and the differences between them.
Employee Table (tblEmployee)



ID	Name	Gender	Salary	DepartmentId
1	Tom	Male	4000	1
2	Pam	Female	3000	3
3	John	Male	3500	1
4	Sam	Male	4500	2
5	Todd	Male	2800	2
6	Ben	Male	7000	1
7	Sara	Female	4800	3
8	Valarie	Female	5500	1
9	James	Male	6500	NULL
10	Russell	Male	8800	NULL

Departments Table (tblDepartment)



Id	DepartmentName	Location	DepartmentHead
1	IT	London	Rick
2	Payroll	Delhi	Ron
3	HR	New York	Christie
4	Other Department	Sydney	Cindrella

SQL Script to create tblEmployee and tblDepartment tables

```
Create table tblDepartment
(
     ID int primary key,
     DepartmentName nvarchar(50),
     Location nvarchar(50),
     DepartmentHead nvarchar(50)
)
Go

Insert into tblDepartment values (1, 'IT', 'London', 'Rick')
Insert into tblDepartment values (2, 'Payroll', 'Delhi', 'Ron')
Insert into tblDepartment values (3, 'HR', 'New York', 'Christie')
Insert into tblDepartment values (4, 'Other Department', 'Sydney', 'Cindrella')
Go

Create table tblEmployee
(
     ID int primary key,
     Name nvarchar(50),
     Gender nvarchar(50),
     Salary int,
     DepartmentId int foreign key references tblDepartment(Id)
)
Go

Insert into tblEmployee values (1, 'Tom', 'Male', 4000, 1)
Insert into tblEmployee values (2, 'Pam', 'Female', 3000, 3)
Insert into tblEmployee values (3, 'John', 'Male', 3500, 1)
Insert into tblEmployee values (4, 'Sam', 'Male', 4500, 2)
Insert into tblEmployee values (5, 'Todd', 'Male', 2800, 2)
Insert into tblEmployee values (6, 'Ben', 'Male', 7000, 1)
Insert into tblEmployee values (7, 'Sara', 'Female', 4800, 3)
Insert into tblEmployee values (8, 'Valarie', 'Female', 5500, 1)
Insert into tblEmployee values (9, 'James', 'Male', 6500, NULL)
Insert into tblEmployee values (10, 'Russell', 'Male', 8800, NULL)
Go
```

General Formula for Joins

```
SELECT      ColumnList
FROM         LeftTableName
JOIN_TYPE  RightTableName
ON              JoinCondition
```

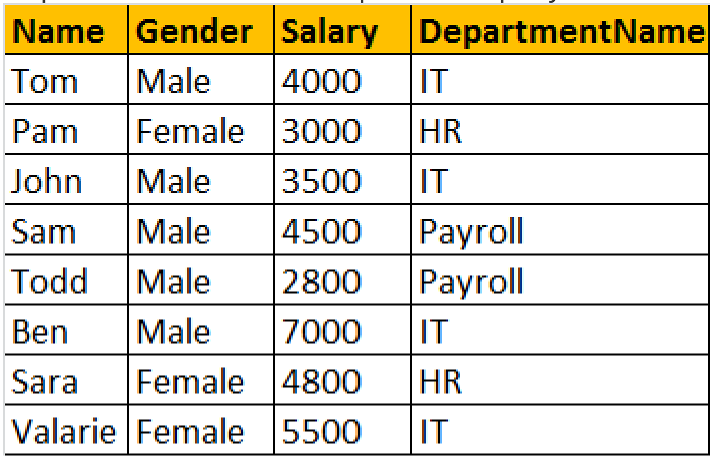
CROSS JOIN

CROSS JOIN, produces the cartesian product of the 2 tables involved in the join. For example, in the Employees table we have 10 rows and in the Departments table we have 4 rows. So, a cross join between these 2 tables produces 40 rows. Cross Join shouldn't have ON clause.

CROSS JOIN Query:

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
CROSS JOIN tblDepartment
```

JOIN or INNER JOIN
Write a query, to retrieve Name, Gender, Salary and DepartmentName from Employees and Departments table. The output of the query should be as shown below.



Name	Gender	Salary	DepartmentName
Tom	Male	4000	IT
Pam	Female	3000	HR
John	Male	3500	IT
Sam	Male	4500	Payroll
Todd	Male	2800	Payroll
Ben	Male	7000	IT
Sara	Female	4800	HR
Valarie	Female	5500	IT

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
INNER JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

OR

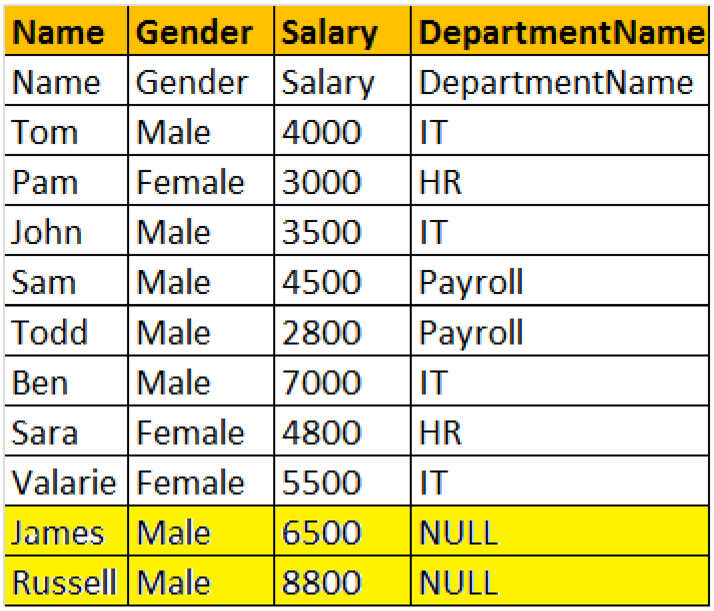
```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

Note: JOIN or INNER JOIN means the same. It's always better to use INNER JOIN, as this explicitly specifies your intention.

If you look at the output, we got only 8 rows, but in the Employees table, we have 10 rows. We didn't get JAMES and RUSSELL records. This is because the DEPARTMENTID, in Employees table is NULL for these two employees and doesn't match with ID column in Departments table.

So, in summary, INNER JOIN, returns only the matching rows between both the tables. Non matching rows are eliminated.

LEFT JOIN or LEFT OUTER JOIN
Now, let's say, I want all the rows from the Employees table, including JAMES and RUSSELL records. I want the output, as shown below.



Name	Gender	Salary	DepartmentName
Name	Gender	Salary	DepartmentName
Tom	Male	4000	IT
Pam	Female	3000	HR
John	Male	3500	IT
Sam	Male	4500	Payroll
Todd	Male	2800	Payroll
Ben	Male	7000	IT
Sara	Female	4800	HR
Valarie	Female	5500	IT
James	Male	6500	NULL
Russell	Male	8800	NULL

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
LEFT OUTER JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

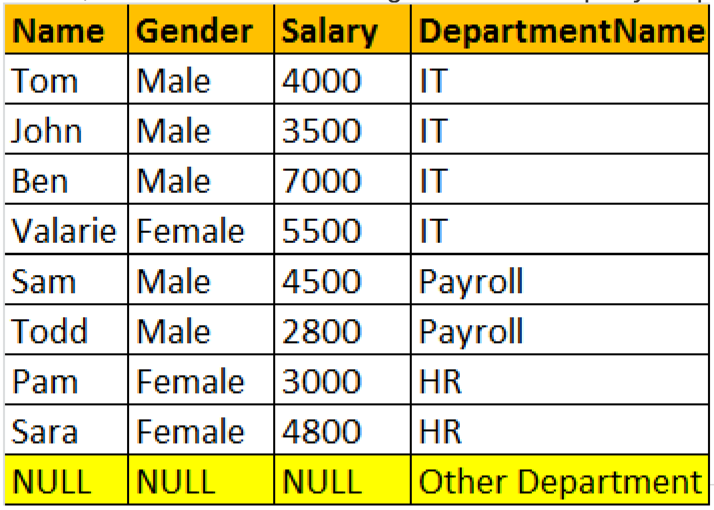
OR

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
LEFT JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

Note: You can use, LEFT JOIN or LEFT OUTER JOIN. OUTER keyowrd is optional

LEFT JOIN, returns all the matching rows + non matching rows from the left table. In reality, INNER JOIN and LEFT JOIN are extensively used.

RIGHT JOIN or RIGHT OUTER JOIN
I want, all the rows from the right table. The query output should be, as shown below.



Name	Gender	Salary	DepartmentName
Tom	Male	4000	IT
John	Male	3500	IT
Ben	Male	7000	IT
Valarie	Female	5500	IT
Sam	Male	4500	Payroll
Todd	Male	2800	Payroll
Pam	Female	3000	HR
Sara	Female	4800	HR
NULL	NULL	NULL	Other Department

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
RIGHT OUTER JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

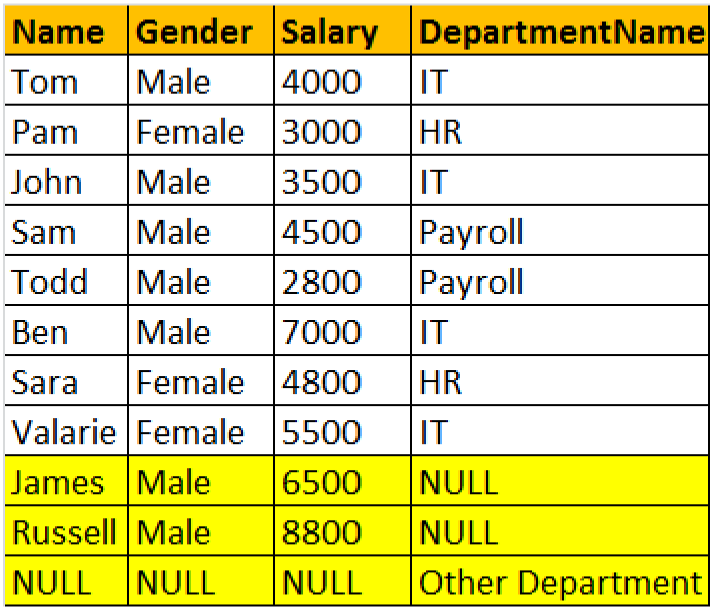
OR

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
RIGHT JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

Note: You can use, RIGHT JOIN or RIGHT OUTER JOIN. OUTER keyowrd is optional

RIGHT JOIN, returns all the matching rows + non matching rows from the right table.

FULL JOIN or FULL OUTER JOIN
I want all the rows from both the tables involved in the join. The query output should be, as shown below.



Name	Gender	Salary	DepartmentName
Tom	Male	4000	IT
Pam	Female	3000	HR
John	Male	3500	IT
Sam	Male	4500	Payroll
Todd	Male	2800	Payroll
Ben	Male	7000	IT
Sara	Female	4800	HR
Valarie	Female	5500	IT
James	Male	6500	NULL
Russell	Male	8800	NULL
NULL	NULL	NULL	Other Department

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
FULL OUTER JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

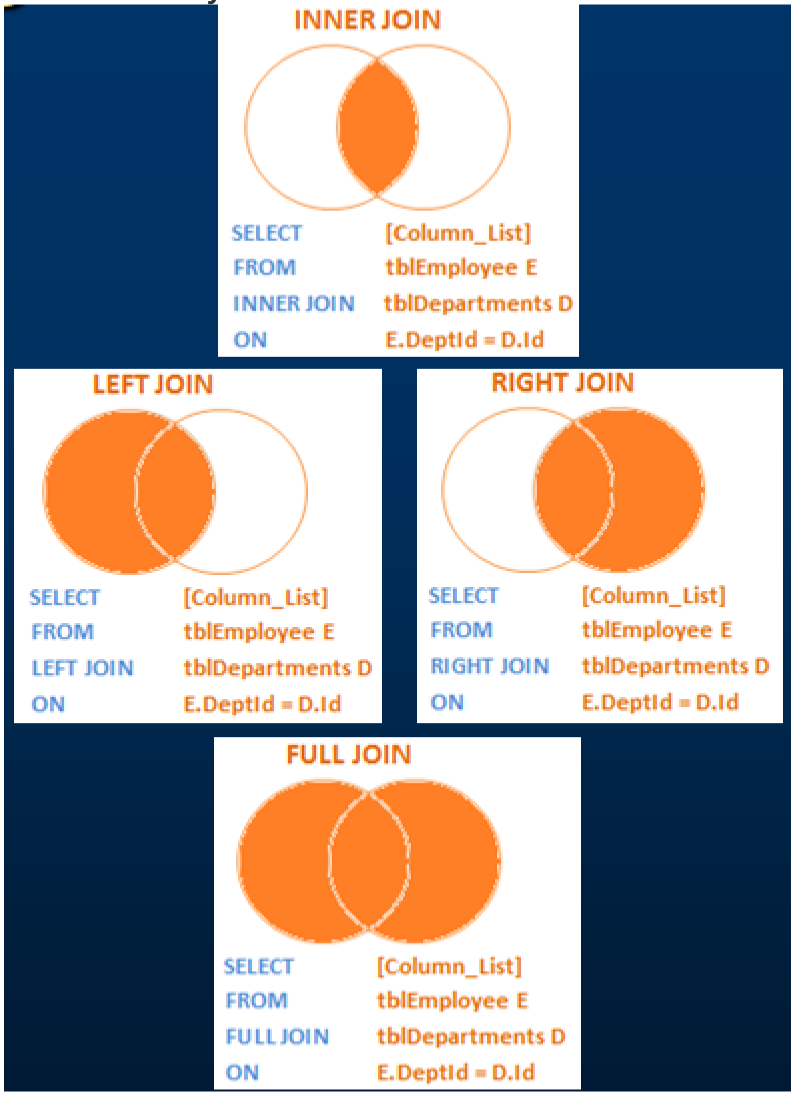
OR

```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
FULL JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

Note: You can use, FULLJOIN or FULL OUTER JOIN. OUTER keyowrd is optional

FULL JOIN, returns all rows from both the left and right tables, including the non matching rows.

Joins Summary

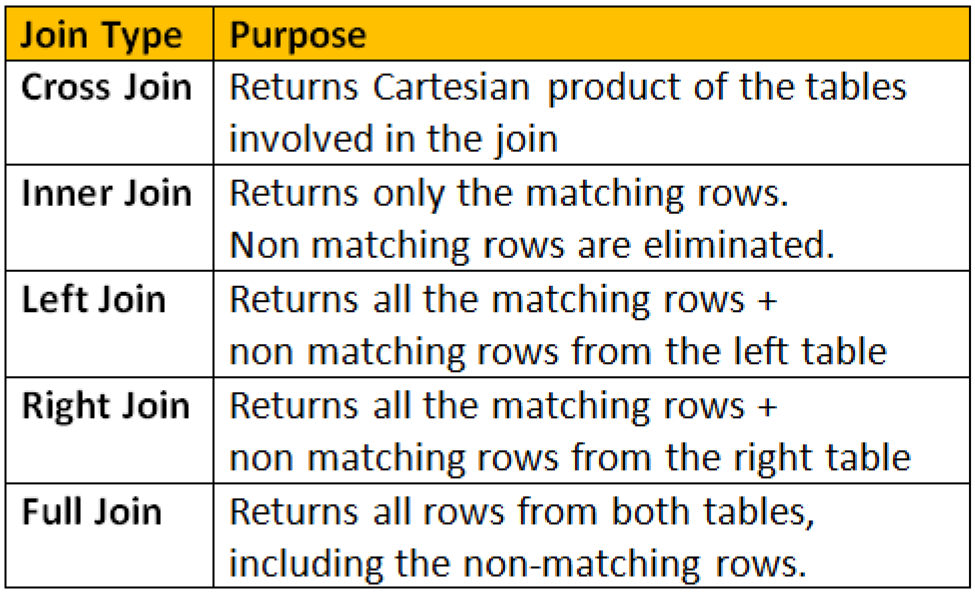


```
SELECT      [Column_List]
FROM        tblEmployee E
INNER JOIN  tblDepartments D
ON          E.DeptId = D.Id
```

```
SELECT      [Column_List]
FROM        tblEmployee E
LEFT JOIN   tblDepartments D
ON          E.DeptId = D.Id
```

```
SELECT      [Column_List]
FROM        tblEmployee E
RIGHT JOIN  tblDepartments D
ON          E.DeptId = D.Id
```

```
SELECT      [Column_List]
FROM        tblEmployee E
FULL JOIN   tblDepartments D
ON          E.DeptId = D.Id
```



Join Type	Purpose
Cross Join	Returns Cartesian product of the tables involved in the join
Inner Join	Returns only the matching rows. Non matching rows are eliminated.
Left Join	Returns all the matching rows + non matching rows from the left table
Right Join	Returns all the matching rows + non matching rows from the right table
Full Join	Returns all rows from both tables, including the non-matching rows.

Advanced Joins - Part 13

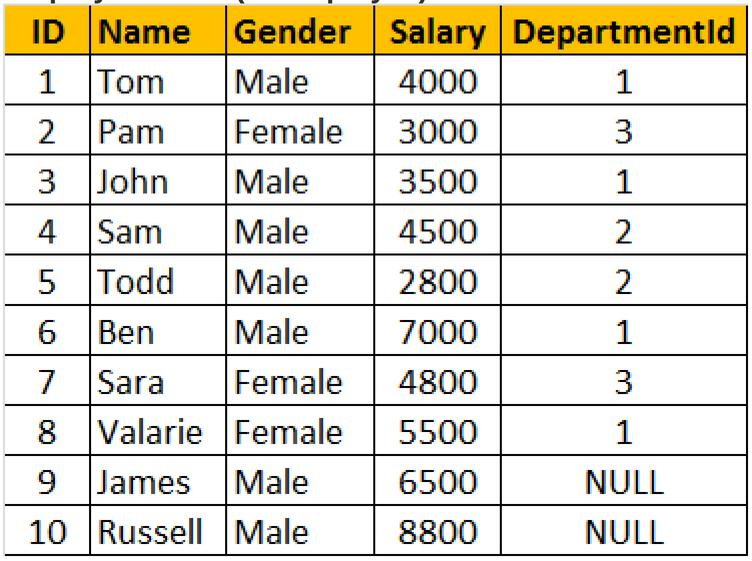
In this video session we will learn about

1. Advanced or intelligent joins in SQL Server
2. Retrieve only the non matching rows from the left table
3. Retrieve only the non matching rows from the right table
4. Retrieve only the non matching rows from both the left and right table

Before watching this video, please watch Part 12 - Joins in SQL Server

Considers Employees (tblEmployee) and Departments (tblDepartment) tables

Employee Table (tblEmployee)



ID	Name	Gender	Salary	DepartmentId
1	Tom	Male	4000	1
2	Pam	Female	3000	3
3	John	Male	3500	1
4	Sam	Male	4500	2
5	Todd	Male	2800	2
6	Ben	Male	7000	1
7	Sara	Female	4800	3
8	Valarie	Female	5500	1
9	James	Male	6500	NULL
10	Russell	Male	8800	NULL

Departments Table (tblDepartment)

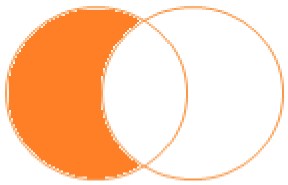
Id	DepartmentName	Location	DepartmentHead
1	IT	London	Rick
2	Payroll	Delhi	Ron
3	HR	New York	Christie
4	Other Department	Sydney	Cindrella

How to retrieve only the non matching rows from the left table. The output should be as shown below:

Name	Gender	Salary	DepartmentName
James	Male	6500	NULL
Russell	Male	8800	NULL

Query:

```
SELECT      Name, Gender, Salary, DepartmentName
FROM        tblEmployee E
LEFT JOIN   tblDepartment D
ON           E.DepartmentId = D.Id
WHERE       D.Id IS NULL
```



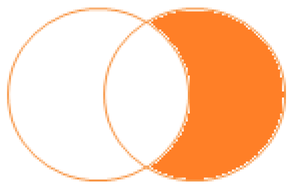
```
SELECT      [Column_List]
FROM        tblEmployee E
LEFT JOIN   tblDepartments D
ON          E.DeptId = D.Id
WHERE       D.Id IS NULL
```

How to retrieve only the non matching rows from the right table

Name	Gender	Salary	DepartmentName
NULL	NULL	NULL	Other Department

Query:

```
SELECT        Name, Gender, Salary, DepartmentName
FROM          tblEmployee E
RIGHT JOIN    tblDepartment D
ON            E.DepartmentId = D.Id
WHERE         E.DepartmentId IS NULL
```



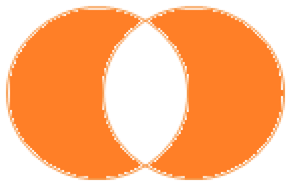
```
SELECT        [Column_List]
FROM          tblEmployee E
RIGHT JOIN    tblDepartments D
ON            E.DeptId = D.Id
WHERE         E.DeptId IS NULL
```

How to retrieve only the non matching rows from both the left and right table. Matching rows should be eliminated.

Name	Gender	Salary	DepartmentName
James	Male	6500	NULL
Russell	Male	8800	NULL
NULL	NULL	NULL	Other Department

Query:

```
SELECT       Name, Gender, Salary, DepartmentName
FROM          tblEmployee E
FULL JOIN    tblDepartment D
ON            E.DepartmentId = D.Id
WHERE        E.DepartmentId IS NULL
OR            D.Id IS NULL
```



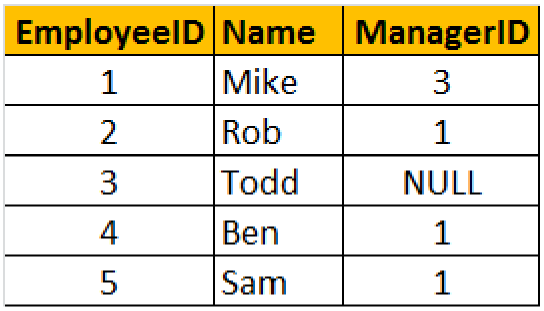
```
SELECT        [Column_List]
FROM          tblEmployee E
FULL JOIN     tblDepartments D
ON            E.DeptId = D.Id
WHERE         E.DeptId IS NULL
OR            D.Id IS NULL
```

Self join in sql server - Part 14

In **Part 12** of this video series we have learnt the **basics of joins** and in **Part 13** we have learnt about **advanced or intelligent joins**. Please watch Parts 12 and 13 before watching this video

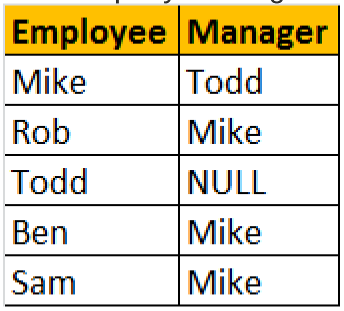
Part 12 - Basic joins
Part 13 - Advanced joins

In parts 12 and 13, we have seen joining 2 different tables - **tblEmployees** and **tblDepartments**. Have you ever thought of a need to join a table with itself. Consider tblEmployees table shown below.



EmployeeID	Name	ManagerID
1	Mike	3
2	Rob	1
3	Todd	NULL
4	Ben	1
5	Sam	1

Write a query which gives the following result.



Employee	Manager
Mike	Todd
Rob	Mike
Todd	NULL
Ben	Mike
Sam	Mike

Self Join Query:
A MANAGER is also an EMPLOYEE. Both the, EMPLOYEE and MANAGER rows, are present in the same table. Here we are joining tblEmployee with itself using different alias names, E for Employee and M for Manager. We are using LEFT JOIN, to get the rows with ManagerId NULL. You can see in the output TODD's record is also retrieved, but the MANAGER is NULL. If you replace LEFT JOIN with INNER JOIN, you will not get TODD's record.

```
Select E.Name as Employee, M.Name as Manager
from tblEmployee E
Left Join tblEmployee M
On E.ManagerId = M.EmployeeId
```

In short, joining a table with itself is called as **SELF JOIN**. SELF JOIN is not a different type of JOIN. It can be classified under any type of JOIN - INNER, OUTER or CROSS Joins. The above query is, LEFT OUTER SELF Join.

Inner Self Join tblEmployee table:

```
Select E.Name as Employee, M.Name as Manager
from tblEmployee E
Inner Join tblEmployee M
On E.ManagerId = M.EmployeeId
```

Cross Self Join tblEmployee table:

```
Select E.Name as Employee, M.Name as Manager
from tblEmployee
Cross Join tblEmployee
```

Different ways to replace NULL in sql server - Part 15

In this video session, we will learn about different ways to replace NULL values in SQL Server. Please watch Part 14, before continuing.

Consider the Employees table below.

EmployeeID	Name	ManagerID
1	Mike	3
2	Rob	1
3	Todd	NULL
4	Ben	1
5	Sam	1

In Part 14, we have learnt writing a LEFT OUTER SELF JOIN query, which produced the following output.

Employee	Manager
Mike	Todd
Rob	Mike
Todd	NULL
Ben	Mike
Sam	Mike

In the output, **MANAGER** column, for **Todd's** rows is **NULL**. I want to replace the **NULL** value, with **'No Manager'**

Replacing NULL value using ISNULL() function: We are passing 2 parameters to IsNULL() function. If M.Name returns NULL, then 'No Manager' string is used as the replacement value.

```
SELECT E.Name as Employee, ISNULL(M.Name,'No Manager') as Manager
FROM tblEmployee E
LEFT JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
```

Replacing NULL value using CASE Statement:

```
SELECT E.Name as Employee, CASE WHEN M.Name IS NULL THEN 'No Manager'
   ELSE M.Name END as Manager
FROM  tblEmployee E
LEFT JOIN tblEmployee M
ON   E.ManagerID = M.EmployeeID
```

Replacing NULL value using COALESCE() function: COALESCE() function, returns the first NON NULL value.

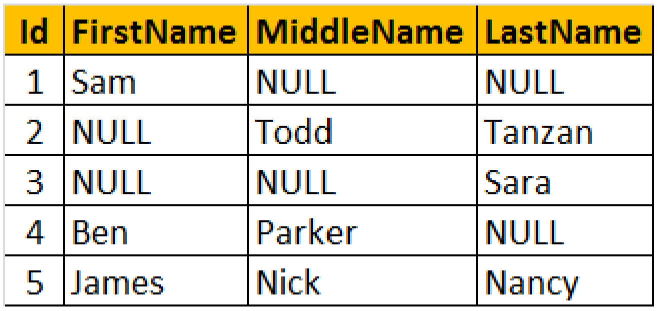
```
SELECT E.Name as Employee, COALESCE(M.Name, 'No Manager') as Manager
FROM tblEmployee E
LEFT JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
```

We will discuss about COALESCE() function in detail, in the next session

Coalesce() function in sql server - Part 16

According to the MSDN Books online COALESCE() returns the first Non NULL value. Let's understand this with an example.

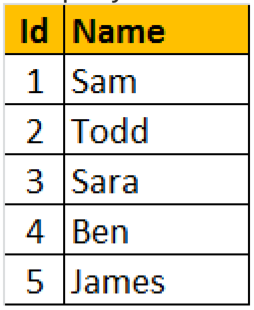
Consider the Employees Table below. Not all employees have their First, Midde and Last Names filled. Some of the employees has First name missing, some of them have Middle Name missing and some of them last name.



Id	FirstName	MiddleName	LastName
1	Sam	NULL	NULL
2	NULL	Todd	Tanzan
3	NULL	NULL	Sara
4	Ben	Parker	NULL
5	James	Nick	Nancy

Now, let's write a query that returns the **Name of the Employee**. If an employee, has all the columns filled - **First, Middle and Last Names**, then we only want the **first name**.

If the **FirstName is NULL**, and if **Middle and Last Names are filled** then, we only want the **middle name**. For example, Employee row with Id = 1, has the FirstName filled, so we want to retrieve his FirstName "Sam". Employee row with Id = 2, has Middle and Last names filled, but the First name is missing. Here, we want to retrieve his middle name "Todd". In short, The output of the query should be as shown below.



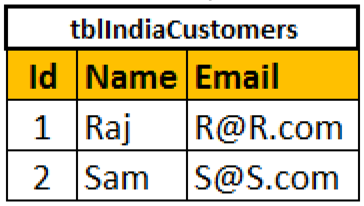
Id	Name
1	Sam
2	Todd
3	Sara
4	Ben
5	James

We are passing **FirstName, MiddleName and LastName** columns as parameters to the COALESCE() function. The COALESCE() function returns the first non null value from the 3 columns.

```
SELECT Id, COALESCE(FirstName, MiddleName, LastName) AS Name
FROM tblEmployee
```

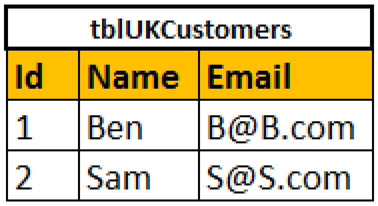
Union and union all in sql server - Part 17

UNION and UNION ALL operators in SQL Server, are used to combine the result-set of two or more SELECT queries. Please consider India and UK customer tables below



tblIndiaCustomers

Id	Name	Email
1	Raj	R@R.com
2	Sam	S@S.com



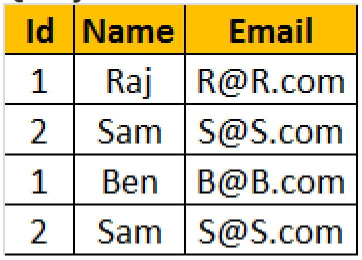
tblUKCustomers

Id	Name	Email
1	Ben	B@B.com
2	Sam	S@S.com

Combining the rows of tblIndiaCustomers and tblUKCustomers using UNION ALL

```
Select Id, Name, Email from tblIndiaCustomers
UNION ALL
Select Id, Name, Email from tblUKCustomers
```

Query Results of UNION ALL

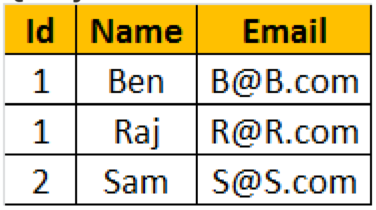


Id	Name	Email
1	Raj	R@R.com
2	Sam	S@S.com
1	Ben	B@B.com
2	Sam	S@S.com

Combining the rows of tblIndiaCustomers and tblUKCustomers using UNION

```
Select Id, Name, Email from tblIndiaCustomers
UNION
Select Id, Name, Email from tblUKCustomers
```

Query Results of UNION



Id	Name	Email
1	Ben	B@B.com
1	Raj	R@R.com
2	Sam	S@S.com

Differences between UNION and UNION ALL (Common Interview Question)

From the output, it is very clear that, **UNION removes duplicate** rows, where as **UNION ALL does not**. When use UNION, to remove the duplicate rows, sql server has to to do a distinct sort, which is time consuming. For this reason, UNION ALL is much faster than UNION.

Note: If you want to see the cost of DISTINCT SORT, you can turn on the estimated query execution plan using CTRL + L.

Note: For UNION and UNION ALL to work, the Number, Data types, and the order of the columns in the select statements should be same.

If you want to sort, the results of UNION or UNION ALL, the ORDER BY caluse should be

used on the last SELECT statement as shown below.

```
Select Id, Name, Email from tblIndiaCustomers
UNION ALL
Select Id, Name, Email from tblUKCustomers
UNION ALL
Select Id, Name, Email from tblUSCustomers
Order by Name
```

The following query, raises a syntax error

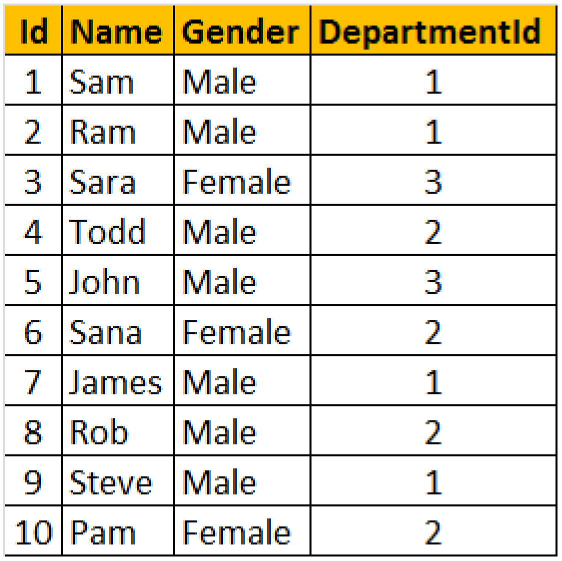
```
SELECT Id, Name, Email FROM tblIndiaCustomers
ORDER BY Name
UNION ALL
SELECT Id, Name, Email FROM tblUKCustomers
UNION ALL
SELECT Id, Name, Email FROM tblUSCustomers
```

Difference between JOIN and UNION
JOINS and **UNIONS** are different things. However, this question is being asked very frequently now. UNION combines the result-set of two or more select queries into a single result-set which includes all the rows from all the queries in the union, where as JOINS, retrieve data from two or more tables based on logical relationships between the tables. In short, UNION combines rows from 2 or more tables, where JOINS combine columns from 2 or more table.

Stored procedures - Part 18

A stored procedure is group of T-SQL (Transact SQL) statements. If you have a situation, where you write the same query over and over again, you can save that specific query as a stored procedure and call it just by it's name.

There are several advantages of using stored procedures, which we will discuss in a later video session. In this session, we will learn how to create, execute, change and delete stored procedures.



Id	Name	Gender	DepartmentId
1	Sam	Male	1
2	Ram	Male	1
3	Sara	Female	3
4	Todd	Male	2
5	John	Male	3
6	Sana	Female	2
7	James	Male	1
8	Rob	Male	2
9	Steve	Male	1
10	Pam	Female	2

Creating a simple stored procedure without any parameters: This stored procedure, retrieves Name and Gender of all the employees. To create a stored procedure we use, **CREATE PROCEDURE** or **CREATE PROC** statement.

```
Create Procedure spGetEmployees
as
Begin
  Select Name, Gender from tblEmployee
End
```

Note: When naming user defined stored procedures, Microsoft recommends not to use **"sp_"** as a prefix. All system stored procedures, are prefixed with **"sp_"**. This avoids any ambiguity between user defined and system stored procedures and any conflicts, with some future system procedure.

To execute the stored procedure, you can just type the procedure name and press F5, or use EXEC or EXECUTE keywords followed by the procedure name as shown below.
1. spGetEmployees
2. EXEC spGetEmployees
3. Execute spGetEmployees

Note: You can also right click on the procedure name, in object explorer in SQL Server Management Studio and select EXECUTE STORED PROCEDURE.

Creating a stored procedure with input parameters: This SP, accepts GENDER and DEPARTMENTID parameters. Parameters and variables have an @ prefix in their name.

```
Create Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
as
Begin
  Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId =
@DepartmentId
End
```

To invoke this procedure, we need to pass the value for @Gender and @DepartmentId parameters. If you don't specify the name of the parameters, you have to first pass value for @Gender parameter and then for @DepartmentId.

```
EXECUTE spGetEmployeesByGenderAndDepartment 'Male', 1
```

On the other hand, if you change the order, you will get an error stating "Error converting data type varchar to int." This is because, the value of **"Male"** is passed into @DepartmentId parameter. Since @DepartmentId is an integer, we get the type conversion error.
spGetEmployeesByGenderAndDepartment 1, 'Male'

When you specify the names of the parameters when executing the stored procedure the order doesn't matter.

```
EXECUTE spGetEmployeesByGenderAndDepartment @DepartmentId=1, @Gender = 'Male'
```

To view the text, of the stored procedure
1. Use system stored procedure sp_helptext 'SPName'

OR
2. Right Click the SP in Object explorer -> Scrip Procedure as -> Create To -> New Query Editor Window

To change the stored procedure, use ALTER PROCEDURE statement:

```
Alter Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
as
Begin
  Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId =
@DepartmentId order by Name
End
```

To encrypt the text of the SP, use WITH ENCRYPTION option. Once, encrypted, you cannot view the text of the procedure, using sp_helptext system stored procedure. There are ways to obtain the original text, which we will talk about in a later session.

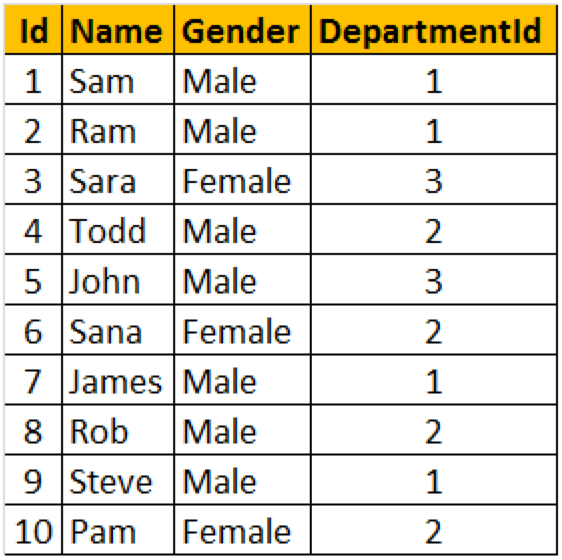
```
Alter Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
WITH ENCRYPTION
as
Begin
  Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId =
@DepartmentId
End
```

To delete the SP, use DROP PROC 'SPName' or DROP PROCEDURE 'SPName'

In the next seesion, we will learn creating stored procedures with OUTPUT parameters.

Stored procedures with output parameters - Part 19

In this video, we will learn about, creating stored procedures with output parameters. Please watch Part 18 of this video series, before watching this video.



Id	Name	Gender	DepartmentId
1	Sam	Male	1
2	Ram	Male	1
3	Sara	Female	3
4	Todd	Male	2
5	John	Male	3
6	Sana	Female	2
7	James	Male	1
8	Rob	Male	2
9	Steve	Male	1
10	Pam	Female	2

To create an SP with output parameter, we use the keywords OUT or OUTPUT.
@EmployeeCount is an OUTPUT parameter. Notice, it is specified with OUTPUT keyword.

```
Create Procedure spGetEmployeeCountByGender
@Gender nvarchar(20),
@EmployeeCount int Output
as
Begin
 Select @EmployeeCount = COUNT(Id)
 from tblEmployee
 where Gender = @Gender
End
```

To execute this stored procedure with OUTPUT parameter

1. First initialise a variable of the **same datatype** as that of the **output parameter**. We have declared @EmployeeTotal integer variable.
2. Then pass the @EmployeeTotal variable to the SP. You have to specify the **OUTPUT** keyword. If you don't specify the OUTPUT keyword, the variable will be **NULL**.
3. Execute

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender 'Female', @EmployeeTotal output
Print @EmployeeTotal
```

If you don't specify the OUTPUT keyword, when executing the stored procedure, the @EmployeeTotal variable will be NULL. Here, we have not specified OUTPUT keyword. When you execute, you will see **'@EmployeeTotal is null'** printed.

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender 'Female', @EmployeeTotal
if(@EmployeeTotal is null)
 Print '@EmployeeTotal is null'
else
 Print '@EmployeeTotal is not null'
```

You can pass parameters in any order, when you use the parameter names. Here, we are first passing the OUTPUT parameter and then the input @Gender parameter.

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender @EmployeeCount = @EmployeeTotal OUT, @Gender
= 'Male'
Print @EmployeeTotal
```

The following system stored procedures, are extremely useful when working procedures.
sp_help SP_Name : View the information about the stored procedure, like parameter names, their datatypes etc. sp_help can be used with any database object, like tables, views, SP's, triggers etc. Alternatively, you can also press ALT+F1, when the name of the object is highlighted.

sp_helptext SP_Name : View the Text of the stored procedure

sp_depends SP_Name : View the dependencies of the stored procedure. This system SP is very useful, especially if you want to check, if there are any stored procedures that are referencing a table that you are abput to drop. sp_depends can also be used with other database objects like table etc.

Note: All parameter and variable names in SQL server, need to have the @symbol.

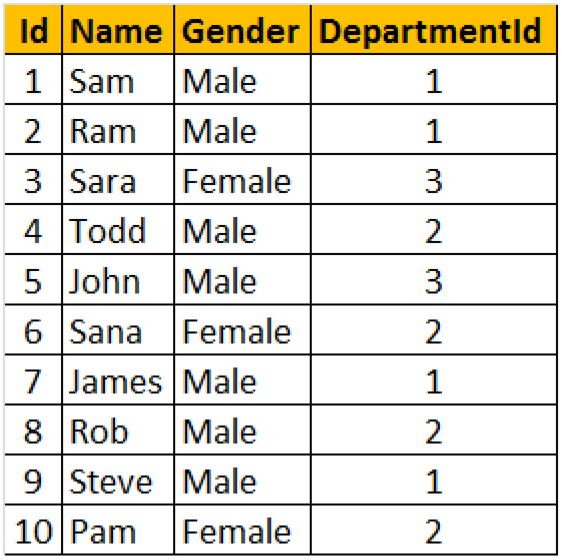
Stored procedure output parameters or return values - Part 20

In this video, we will
1. Understand what are stored procedure return values
2. Difference between stored procedure return values and output parameters
3. When to use output parameters over return values

Before watching this video, please watch
Part 18 - Stored procedure basics in sql server
Part 19 - Stored procedures with output parameters

What are stored procedure status variables?
Whenever, you execute a stored procedure, it returns an integer status variable. Usually, zero indicates success, and non-zero indicates failure. To see this yourself, execute any stored procedure from the object explorer, in sql server management studio.
1. Right Click and select 'Execute Stored Procedure
2. If the procedure, expects parameters, provide the values and click OK.
3. Along with the result that you expect, the stored procedure, also returns a Return Value = 0

So, from this we understood that, when a stored procedure is executed, it returns an integer status variable. With this in mind, let's understand the difference between output parameters and RETURN values. We will use the Employees table below for this purpose.



Id	Name	Gender	DepartmentId
1	Sam	Male	1
2	Ram	Male	1
3	Sara	Female	3
4	Todd	Male	2
5	John	Male	3
6	Sana	Female	2
7	James	Male	1
8	Rob	Male	2
9	Steve	Male	1
10	Pam	Female	2

The following procedure returns total number of employees in the Employees table, using output parameter - @TotalCount.

```
Create Procedure spGetTotalCountOfEmployees1
@TotalCount int output
as
Begin
 Select @TotalCount = COUNT(ID) from tblEmployee
End
```

Executing spGetTotalCountOfEmployees1 returns 3.

```
Declare @TotalEmployees int
Execute spGetTotalCountOfEmployees @TotalEmployees Output
Select @TotalEmployees
```

Re-written stored procedure using return variables

```
Create Procedure spGetTotalCountOfEmployees2
as
Begin
 return (Select COUNT(ID) from Employees)
End
```

Executing spGetTotalCountOfEmployees2 returns 3.

```
Declare @TotalEmployees int
Execute @TotalEmployees = spGetTotalCountOfEmployees2
Select @TotalEmployees
```

So, we are able to achieve what we want, using output parameters as well as return values. Now, let's look at example, where return status variables cannot be used, but Output parameters can be used.

In this SP, we are retrieving the Name of the employee, based on their Id, using the output parameter @Name.

```
Create Procedure spGetNameById1
@Id int,
@Name nvarchar(20) Output
as
Begin
 Select @Name = Name from tblEmployee Where Id = @Id
End
```

Executing spGetNameById1, prints the name of the employee

```
Declare @EmployeeName nvarchar(20)
Execute spGetNameById1 3, @EmployeeName out
Print 'Name of the Employee = ' + @EmployeeName
```

Now let's try to achieve the same thing, using return status variables.

```
Create Procedure spGetNameById2
@Id int
as
Begin
 Return (Select Name from tblEmployee Where Id = @Id)
End
```

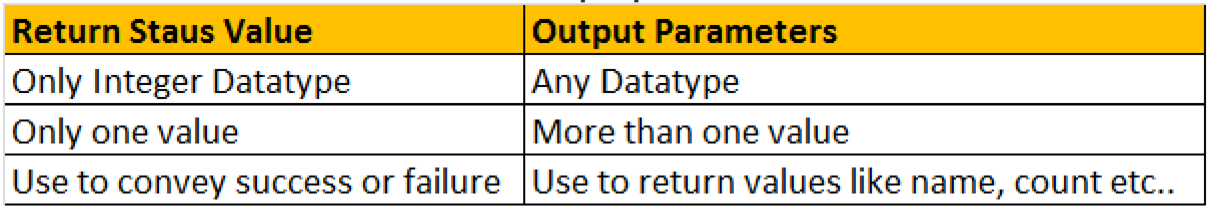
Executing spGetNameById2 returns an error stating 'Conversion failed when converting the nvarchar value 'Sam' to data type int.'. The return status variable is an integer, and hence, when we select Name of an employee and try to return that we get a converion error.

```
Declare @EmployeeName nvarchar(20)
Execute @EmployeeName = spGetNameById2 1
Print 'Name of the Employee = ' + @EmployeeName
```

So, using return values, we can only return integers, and that too, only one integer. It is not possible, to return more than one value using return values, where as output parameters, can return any datatype and an sp can have more than one output parameters. I always prefer, using output parameters, over RETURN values.

In general, RETURN values are used to indicate success or failure of stored procedure, especially when we are dealing with nested stored procedures.Return a value of 0, indicates success, and any nonzero value indicates failure.

Difference between return values and output parameters



Return Staus Value	Output Parameters
Only Integer Datatype	Any Datatype
Only one value	More than one value
Use to convey success or failure	Use to return values like name, count etc..

Advantages of using stored procedures - Part 21

Please watch Part 18 - Basics of Stored Procedures

The following advantages of using Stored Procedures over adhoc queries (inline SQL)

1. Execution plan retention and reusability - Stored Procedures are compiled and their execution plan is cached and used again, when the same SP is executed again. Although adhoc queries also create and reuse plan, the plan is reused only when the query is textual match and the datatypes are matching with the previous call. Any change in the datatype or you have an extra space in the query then, a new plan is created.

2. Reduces network traffic - You only need to send, EXECUTE SP_Name statement, over the network, instead of the entire batch of adhoc SQL code.

3. Code reusability and better maintainability - A stored procedure can be reused with multiple applications. If the logic has to change, we only have one place to change, where as if it is inline sql, and if you have to use it in multiple applications, we end up with multiple copies of this inline sql. If the logic has to change, we have to change at all the places, which makes it harder maintaining inline sql.

4. Better Security - A database user can be granted access to an SP and prevent them from executing direct "select" statements against a table. This is fine grain access control which will help control what data a user has access to.

5. Avoids SQL Injection attack - SP's prevent sql injection attack. Please watch this video on SQL Injection Attack, for more information.

Built in string functions in sql server 2008 - Part 22

Functions in SQL server can be broadly divided into 2 categoris
1. Built-in functions
2. User Defined functions

There are several built-in functions. In this video session, we will look at the most common string functions available.

ASCII(Character_Expression) - Returns the ASCII code of the given character expression.
To find the ACII Code of capital letter 'A'

Example: Select ASCII('A')
Output: 65

CHAR(Integer_Expression) - Converts an int ASCII code to a character. The Integer_Expression, should be between 0 and 255.
The following SQL, prints all the characters for the ASCII values from o thru 255

```
Declare @Number int
Set @Number = 1
While(@Number <= 255)
Begin
 Print CHAR(@Number)
 Set @Number = @Number + 1
End
```

Note: The while loop will become an infinite loop, if you forget to include the following line.
Set @Number = @Number + 1

Printing uppercase alphabets using CHAR() function:
```
Declare @Number int
Set @Number = 65
While(@Number <= 90)
Begin
 Print CHAR(@Number)
 Set @Number = @Number + 1
End
```

Printing lowercase alphabets using CHAR() function:
```
Declare @Number int
Set @Number = 97
While(@Number <= 122)
Begin
 Print CHAR(@Number)
 Set @Number = @Number + 1
End
```

Another way of printing lower case alphabets using CHAR() and LOWER() functions.
```
Declare @Number int
Set @Number = 65
```

```
While(@Number <= 90)
Begin
 Print LOWER(CHAR(@Number))
 Set @Number = @Number + 1
End
```

LTRIM(Character_Expression) - Removes blanks on the left handside of the given character expression.

Example: Removing the 3 white spaces on the left hand side of the '   Hello' string using LTRIM() function.

```
Select LTRIM('   Hello')
```

Output: Hello

RTRIM(Character_Expression) - Removes blanks on the right hand side of the given character expression.

Example: Removing the 3 white spaces on the left hand side of the 'Hello   ' string using RTRIM() function.

```
Select RTRIM('Hello   ')
```

Output: Hello

Example: To remove white spaces on either sides of the given character expression, use LTRIM() and RTRIM() as shown below.

```
Select LTRIM(RTRIM('   Hello   '))
```

Output: Hello

LOWER(Character_Expression) - Converts all the characters in the given Character_Expression, to lowercase letters.

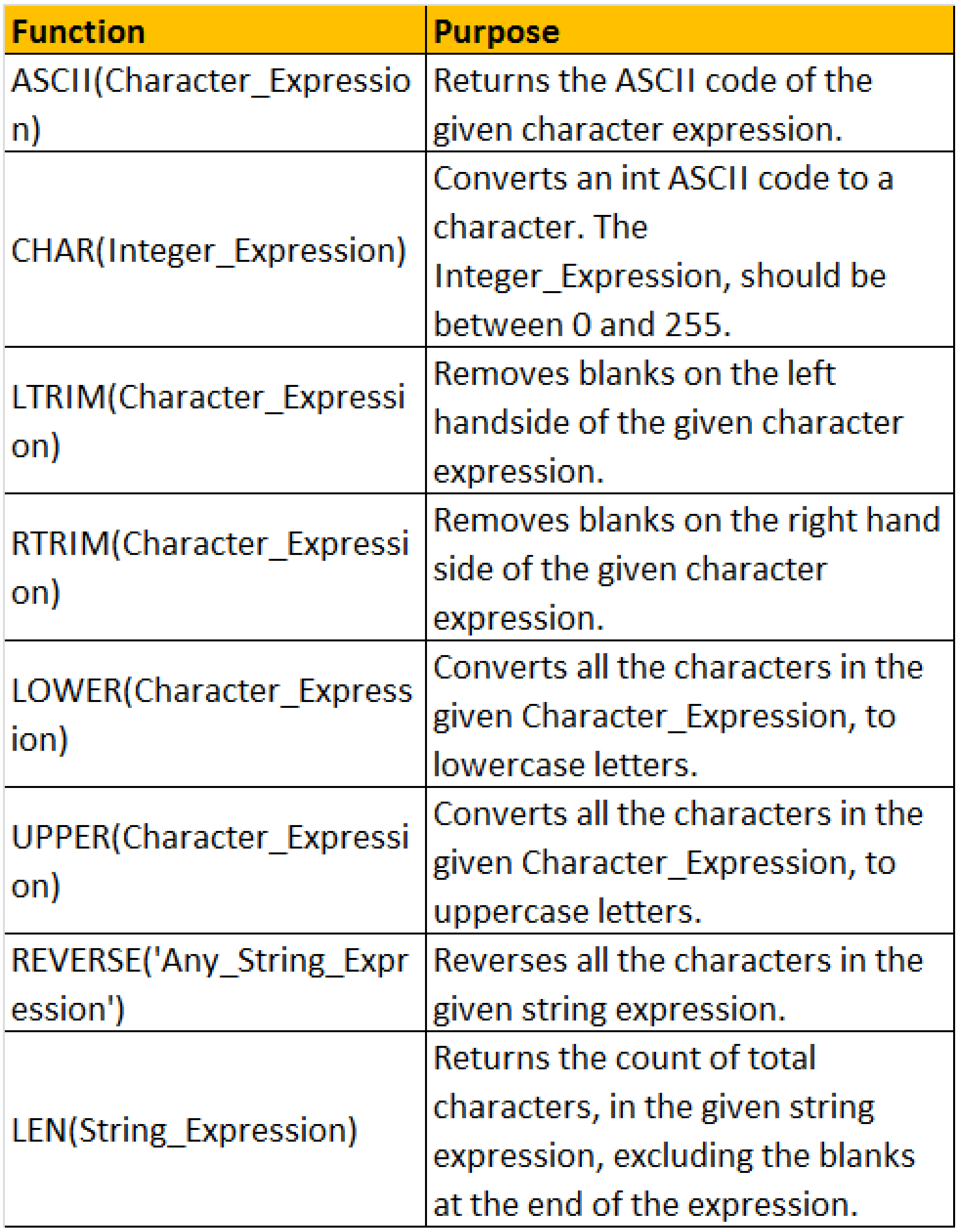
Example: `Select LOWER('CONVERT This String Into Lower Case')`
Output: convert this string into lower case

UPPER(Character_Expression) - Converts all the characters in the given Character_Expression, to uppercase letters.
Example: `Select UPPER('CONVERT This String Into upper Case')`
Output: CONVERT THIS STRING INTO UPPER CASE

REVERSE('Any_String_Expression') - Reverses all the characters in the given string expression.
Example: `Select REVERSE('ABCDEFGHIJKLMNOPQRSTUVWXYZ')`
Output: ZYXWVUTSRQPONMLKJIHGFEDCBA

LEN(String_Expression) - Returns the count of total characters, in the given string expression, excluding the blanks at the end of the expression.

Example: `Select LEN('SQL Functions   ')`
Output: 13



Function	Purpose
ASCII(Character_Expression)	Returns the ASCII code of the given character expression.
CHAR(Integer_Expression)	Converts an int ASCII code to a character. The Integer_Expression, should be between 0 and 255.
LTRIM(Character_Expression)	Removes blanks on the left handside of the given character expression.
RTRIM(Character_Expression)	Removes blanks on the right hand side of the given character expression.
LOWER(Character_Expression)	Converts all the characters in the given Character_Expression, to lowercase letters.
UPPER(Character_Expression)	Converts all the characters in the given Character_Expression, to uppercase letters.
REVERSE('Any_String_Expression')	Reverses all the characters in the given string expression.
LEN(String_Expression)	Returns the count of total characters, in the given string expression, excluding the blanks at the end of the expression.

In the next video session, we will discuss about the rest of the commonly used built-in string functions.

LEFT, RIGHT, CHARINDEX and SUBSTRING functions - Part 23

In this video we will learn about the commonly used built-in string functions in SQL server and finally, a real time example of using string functions. Please watch the following videos, before continuing with this video.
Part 11 – Group By
Part 22 – Built in string functions

LEFT(Character_Expression, Integer_Expression) - Returns the specified number of characters from the left hand side of the given character expression.

Example: Select LEFT('ABCDE', 3)

Output: ABC

RIGHT(Character_Expression, Integer_Expression) - Returns the specified number of characters from the right hand side of the given character expression.

Example: Select RIGHT('ABCDE', 3)
Output: CDE

CHARINDEX('Expression_To_Find', 'Expression_To_Search', 'Start_Location') - Returns the starting position of the specified expression in a character string. Start_Location parameter is optional.

Example: In this example, we get the starting position of '@' character in the email string 'sara@aaa.com'.
Select CHARINDEX('@','sara@aaa.com',1)
Output: 5

SUBSTRING('Expression', 'Start', 'Length') - As the name, suggests, this function returns substring (part of the string), from the given expression. You specify the starting location using the 'start' parameter and the number of characters in the substring using 'Length' parameter. All the 3 parameters are mandatory.

Example: Display just the domain part of the given email 'John@bbb.com'.
Select SUBSTRING('John@bbb.com',6, 7)
Output: bbb.com

In the above example, we have hardcoded the starting position and the length parameters. Instead of hardcoding we can dynamically retrieve them using CHARINDEX() and LEN() string functions as shown below.

Example:
Select SUBSTRING('John@bbb.com',(CHARINDEX('@', 'John@bbb.com') + 1),
(LEN('John@bbb.com') - CHARINDEX('@','John@bbb.com')))
Output: bbb.com

Real time example, where we can use LEN(), CHARINDEX() and SUBSTRING() functions. Let us assume we have table as shown below.

Id	FirstName	LastName	Email
1	Sam	Sony	Sam@aaa.com
2	Ram	Barber	Ram@aaa.com
3	Sara	Sanosky	Sara@ccc.com
4	Todd	Gartner	Todd@bbb.com
5	John	Grover	John@aaa.com
6	Sana	Lenin	Sana@ccc.com
7	James	Bond	James@bbb.com
8	Rob	Hunter	Rob@ccc.com
9	Steve	Wilson	Steve@aaa.com
10	Pam	Broker	Pam@bbb.com

Write a query to find out total number of emails, by domain. The result of the query should be as shown below.

EmailDomain	Total
aaa.com	4
bbb.com	3
ccc.com	3

Query

```
Select SUBSTRING(Email, CHARINDEX('@', Email) + 1,
LEN(Email) - CHARINDEX('@', Email)) as EmailDomain,
COUNT(Email) as Total
from tblEmployee
Group By SUBSTRING(Email, CHARINDEX('@', Email) + 1,
LEN(Email) - CHARINDEX('@', Email))
```

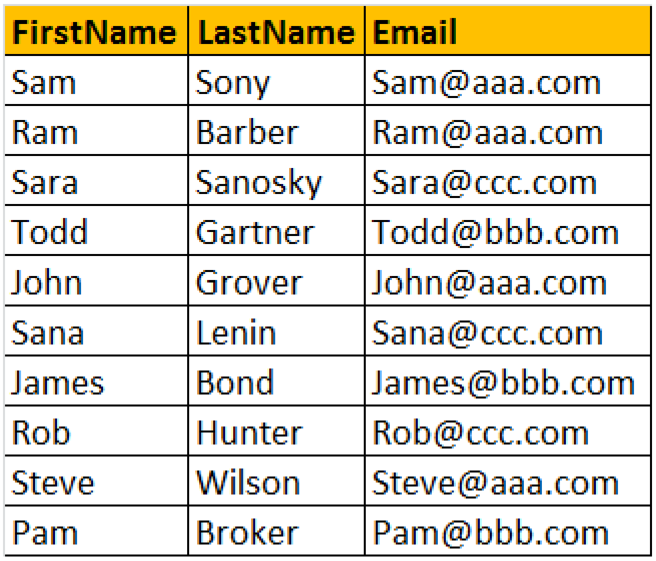
Replicate, Space, Patindex, Replace and Stuff functions - Part 24

Before watching this video, please watch
Part 22 – Built in string functions in sql server
Part 23 – Left, Right, CharIndex and Substring functions

REPLICATE(String_To_Be_Replicated, Number_Of_Times_To_Replicate) - Repeats the given string, for the specified number of times.

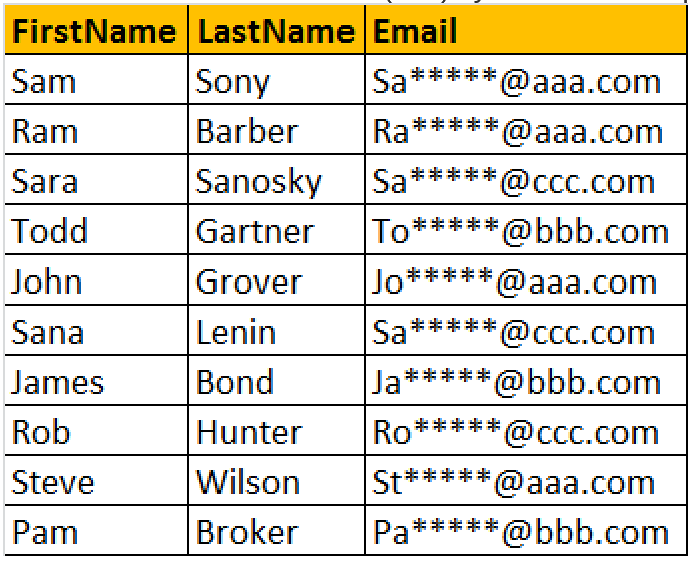
Example: SELECT REPLICATE('Pragim', 3)
Output: Pragim Pragim Pragim

A practical example of using REPLICATE() function: We will be using this table, for the rest of our examples in this article.



FirstName	LastName	Email
Sam	Sony	Sam@aaa.com
Ram	Barber	Ram@aaa.com
Sara	Sanosky	Sara@ccc.com
Todd	Gartner	Todd@bbb.com
John	Grover	John@aaa.com
Sana	Lenin	Sana@ccc.com
James	Bond	James@bbb.com
Rob	Hunter	Rob@ccc.com
Steve	Wilson	Steve@aaa.com
Pam	Broker	Pam@bbb.com

Let's mask the email with 5 * (star) symbols. The output should be as shown below.



FirstName	LastName	Email
Sam	Sony	Sa*****@aaa.com
Ram	Barber	Ra*****@aaa.com
Sara	Sanosky	Sa*****@ccc.com
Todd	Gartner	To*****@bbb.com
John	Grover	Jo*****@aaa.com
Sana	Lenin	Sa*****@ccc.com
James	Bond	Ja*****@bbb.com
Rob	Hunter	Ro*****@ccc.com
Steve	Wilson	St*****@aaa.com
Pam	Broker	Pa*****@bbb.com

Query:

```
Select FirstName, LastName, SUBSTRING(Email, 1, 2) + REPLICATE('*',5) +
SUBSTRING(Email, CHARINDEX('@',Email), LEN(Email) - CHARINDEX('@',Email)+1) as Email
from tblEmployee
```

SPACE(Number_Of_Spaces) - Returns number of spaces, specified by the Number_Of_Spaces argument.

Example: The SPACE(5) function, inserts 5 spaces between FirstName and LastName

```
Select FirstName + SPACE(5) + LastName as FullName
From tblEmployee
```

Output:

FullName
Sam   Sony
Ram   Barber
Sara   Sanosky
Todd   Gartner
John   Grover
Sana   Lenin
James   Bond
Rob   Hunter
Steve   Wilson
Pam   Broker

PATINDEX('%Pattern%', Expression)
Returns the starting position of the first occurrence of a pattern in a specified expression. It takes two arguments, the pattern to be searched and the expression. PATINDEX() is simial to CHARINDEX(). With CHARINDEX() we cannot use wildcards, where as PATINDEX() provides this capability. If the specified pattern is not found, PATINDEX() returns ZERO.

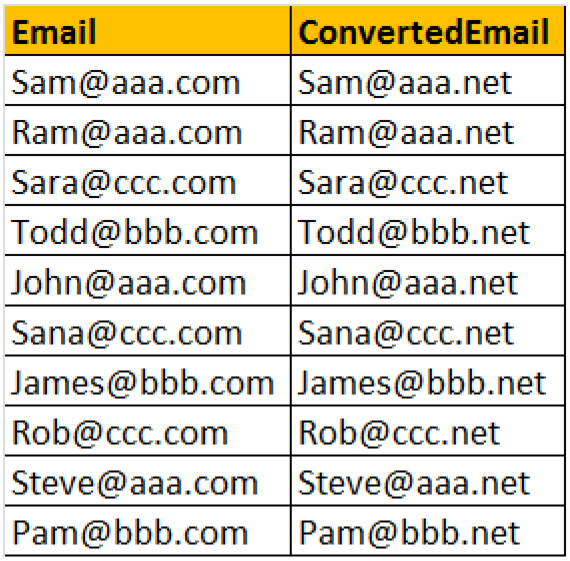
Example:
```
Select Email, PATINDEX('%@aaa.com', Email) as FirstOccurence
from tblEmployee
Where PATINDEX('%@aaa.com', Email) > 0
```

Output:

Email	FirstOccurence
Sam@aaa.com	4
Ram@aaa.com	4
John@aaa.com	5
Steve@aaa.com	6

REPLACE(String_Expression, Pattern , Replacement_Value)
Replaces all occurrences of a specified string value with another string value.

Example: All .COM strings are replaced with .NET
```
Select Email, REPLACE(Email, '.com', '.net') as ConvertedEmail
from  tblEmployee
```

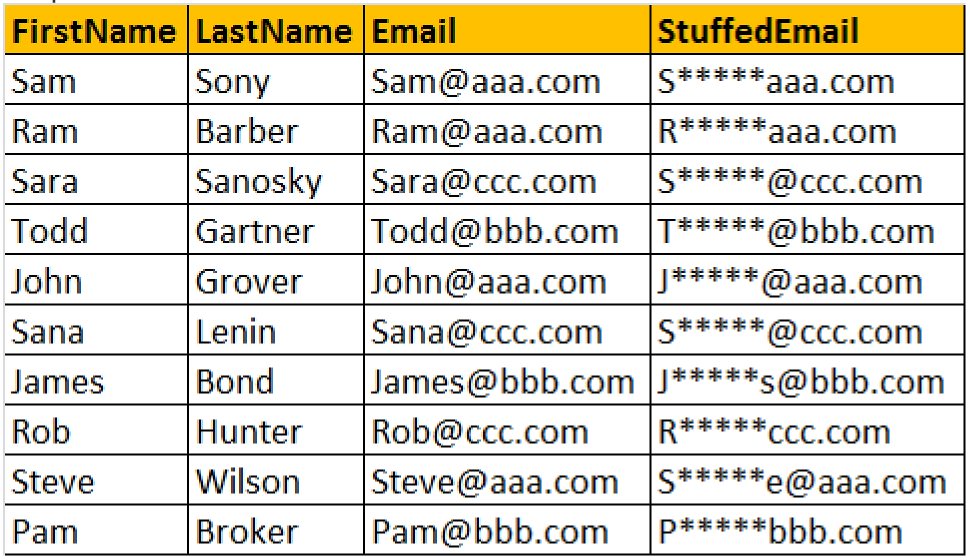


Email	ConvertedEmail
Sam@aaa.com	Sam@aaa.net
Ram@aaa.com	Ram@aaa.net
Sara@ccc.com	Sara@ccc.net
Todd@bbb.com	Todd@bbb.net
John@aaa.com	John@aaa.net
Sana@ccc.com	Sana@ccc.net
James@bbb.com	James@bbb.net
Rob@ccc.com	Rob@ccc.net
Steve@aaa.com	Steve@aaa.net
Pam@bbb.com	Pam@bbb.net

STUFF(Original_Expression, Start, Length, Replacement_expression)
STUFF() function inserts Replacement_expression, at the start position specified, along with removing the charactes specified using Length parameter.

Example:
```
Select FirstName, LastName,Email, STUFF(Email, 2, 3, '*****') as StuffedEmail
From tblEmployee
```

Output:



FirstName	LastName	Email	StuffedEmail
Sam	Sony	Sam@aaa.com	S*****aaa.com
Ram	Barber	Ram@aaa.com	R*****aaa.com
Sara	Sanosky	Sara@ccc.com	S*****@ccc.com
Todd	Gartner	Todd@bbb.com	T*****@bbb.com
John	Grover	John@aaa.com	J*****@aaa.com
Sana	Lenin	Sana@ccc.com	S*****@ccc.com
James	Bond	James@bbb.com	J*****s@bbb.com
Rob	Hunter	Rob@ccc.com	R*****ccc.com
Steve	Wilson	Steve@aaa.com	S*****e@aaa.com
Pam	Broker	Pam@bbb.com	P*****bbb.com

DateTime functions in SQL Server - Part 25

In this video session we will learn about
1. DateTime data types
2. DateTime functions available to select the current system date and time
3. Understanding concepts - UTC time and Time Zone offset

There are several built-in DateTime functions available in SQL Server. All the following functions can be used to get the current system date and time, where you have sql server installed.

Function	Date Time Format	Description
GETDATE()	2012-08-31 20:15:04.543	Commonly used function
CURRENT_TIMESTAMP	2012-08-31 20:15:04.543	ANSI SQL equivalent to GETDATE
SYSDATETIME()	2012-08-31 20:15:04.5380028	More fractional seconds precision
SYSDATETIMEOFFSET()	2012-08-31 20:15:04.5380028 + 01:00	More fractional seconds precision + Time zone offset
GETUTCDATE()	2012-08-31 19:15:04.543	UTC Date and Time
SYSUTCDATETIME()	2012-08-31 19:15:04.5380028	UTC Date and Time, with More fractional seconds precision

Note: **UTC** stands for **Coordinated Universal Time**, based on which, the world regulates clocks and time. There are slight differences between GMT and UTC, but for most common purposes, UTC is synonymous with GMT.

To practically understand how the different date time datatypes available in SQL Server, store data, create the sample table **tblDateTime**.

```
CREATE TABLE [tblDateTime]
(
 [c_time] [time](7) NULL,
 [c_date] [date] NULL,
 [c_smalldatetime] [smalldatetime] NULL,
 [c_datetime] [datetime] NULL,
 [c_datetime2] [datetime2](7) NULL,
 [c_datetimeoffset] [datetimeoffset](7) NULL
)
```

To Insert some sample data, execute the following query.

```
INSERT
INTO tblDateTime VALUES (GETDATE(),GETDATE(),GETDATE(),GETDATE(),GETDATE(),GET
DATE())
```

Now, issue a select statement, and you should see, the different types of datetime datatypes, storing the current datetime, in different formats.

IsDate, Day, Month, Year and DateName DateTime functions in SQL Server - Part 26

ISDATE() - Checks if the given value, is a valid date, time, or datetime. Returns 1 for success, 0 for failure.

Examples:

```
Select ISDATE('PRAGIM') -- returns 0
Select ISDATE(Getdate()) -- returns 1
Select ISDATE('2012-08-31 21:02:04.167') -- returns 1
```

Note: For datetime2 values, IsDate returns ZERO.

Example:

```
Select ISDATE('2012-09-01 11:34:21.1918447') -- returns 0.
```

Day() - Returns the **'Day number of the Month'** of the given date

Examples:
Select DAY(GETDATE()) -- Returns the day number of the month, based on current system datetime.
Select DAY('01/31/2012') -- Returns 31

Month() - Returns the **'Month number of the year'** of the given date

Examples:
Select Month(GETDATE()) -- Returns the **Month number of the year**, based on the current system date and time
Select Month('01/31/2012') -- Returns 1

Year() - Returns the **'Year number'** of the given date

Examples:
Select Year(GETDATE()) -- Returns the year number, based on the current system date
Select Year('01/31/2012') -- Returns 2012

DateName(DatePart, Date) - Returns a string, that represents a part of the given date. This functions takes 2 parameters. The first parameter **'DatePart'** specifies, the part of the date, we want. The second parameter, is the actual date, from which we want the part of the Date.

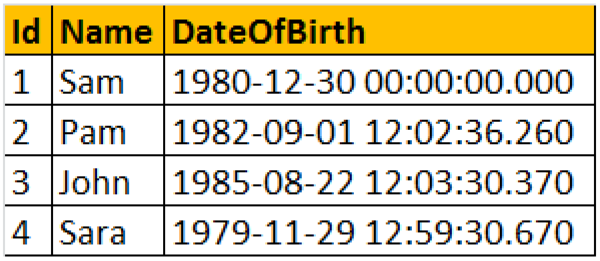
Valid Datepart parameter values

DatePart	Abbreviation
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns
TZoffset	tz

Examples:
Select DATENAME(Day, '2012-09-30 12:43:46.837') -- Returns 30

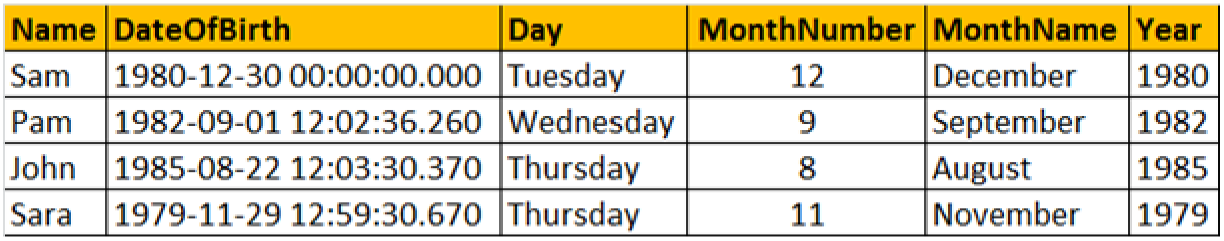
```
Select DATENAME(WEEKDAY, '2012-09-30 12:43:46.837') -- Returns Sunday
Select DATENAME(MONTH, '2012-09-30 12:43:46.837') -- Returns September
```

A simple practical example using some of these DateTime functions. Consider the table tblEmployees.



Id	Name	DateOfBirth
1	Sam	1980-12-30 00:00:00.000
2	Pam	1982-09-01 12:02:36.260
3	John	1985-08-22 12:03:30.370
4	Sara	1979-11-29 12:59:30.670

Write a query, which returns Name, DateOfBirth, Day, MonthNumber, MonthName, and Year as shown below.



Name	DateOfBirth	Day	MonthNumber	MonthName	Year
Sam	1980-12-30 00:00:00.000	Tuesday	12	December	1980
Pam	1982-09-01 12:02:36.260	Wednesday	9	September	1982
John	1985-08-22 12:03:30.370	Thursday	8	August	1985
Sara	1979-11-29 12:59:30.670	Thursday	11	November	1979

Query:

```
Select Name, DateOfBirth, DateName(WEEKDAY,DateOfBirth) as [Day],
          Month(DateOfBirth) as MonthNumber,
          DateName(MONTH, DateOfBirth) as [MonthName],
          Year(DateOfBirth) as [Year]
From   tblEmployees
```

DatePart, DateAdd and DateDiff functions in SQL Server - Part 27

DatePart(DatePart, Date) - Returns an integer representing the specified DatePart. This function is simialar to DateName(). DateName() returns nvarchar, where as DatePart() returns an integer. The valid DatePart parameter values are shown below.

DatePart	Abbreviation
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns
TZoffset	tz

Examples:
```
Select DATEPART(weekday, '2012-08-30 19:45:31.793') -- returns 5
Select DATENAME(weekday, '2012-08-30 19:45:31.793') -- returns Thursday
```

DATEADD (datepart, NumberToAdd, date) - Returns the DateTime, after adding specified NumberToAdd, to the datepart specified of the given date.

Examples:
```
Select DateAdd(DAY, 20, '2012-08-30 19:45:31.793')
-- Returns 2012-09-19 19:45:31.793
Select DateAdd(DAY, -20, '2012-08-30 19:45:31.793')
-- Returns 2012-08-10 19:45:31.793
```

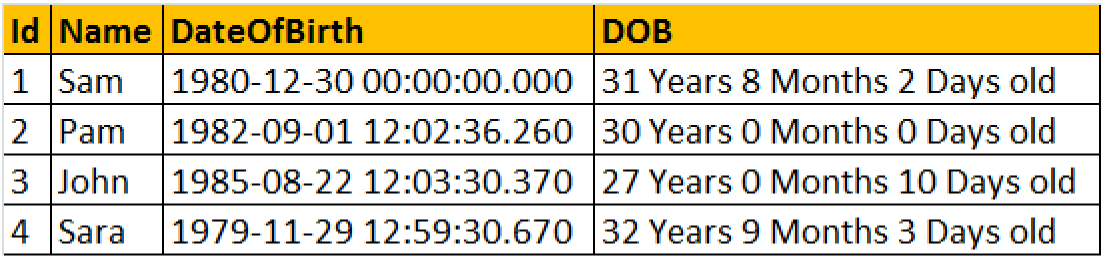
DATEDIFF(datepart, startdate, enddate) - Returns the count of the specified datepart boundaries crossed between the specified startdate and enddate.

Examples:
```
Select DATEDIFF(MONTH, '11/30/2005','01/31/2006') -- returns 2
Select DATEDIFF(DAY, '11/30/2005','01/31/2006') -- returns 62
```

Consider the emaployees table below.

Id	Name	DateOfBirth
1	Sam	1980-12-30 00:00:00.000
2	Pam	1982-09-01 12:02:36.260
3	John	1985-08-22 12:03:30.370
4	Sara	1979-11-29 12:59:30.670

Write a query to compute the age of a person, when the date of birth is given. The output should be as shown below.



Id	Name	DateOfBirth	DOB
1	Sam	1980-12-30 00:00:00.000	31 Years 8 Months 2 Days old
2	Pam	1982-09-01 12:02:36.260	30 Years 0 Months 0 Days old
3	John	1985-08-22 12:03:30.370	27 Years 0 Months 10 Days old
4	Sara	1979-11-29 12:59:30.670	32 Years 9 Months 3 Days old

```
CREATE FUNCTION fnComputeAge(@DOB DATETIME)
RETURNS NVARCHAR(50)
AS
BEGIN

DECLARE @tempdate DATETIME, @years INT, @months INT, @days INT
SELECT @tempdate = @DOB

SELECT @years = DATEDIFF(YEAR, @tempdate, GETDATE()) - CASE
WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR (MONTH(@DOB)
= MONTH(GETDATE()) AND DAY(@DOB) > DAY(GETDATE())) THEN 1 ELSE 0 END
SELECT @tempdate = DATEADD(YEAR, @years, @tempdate)

SELECT @months = DATEDIFF(MONTH, @tempdate, GETDATE()) - CASE
WHEN DAY(@DOB) > DAY(GETDATE()) THEN 1 ELSE 0 END
SELECT @tempdate = DATEADD(MONTH, @months, @tempdate)

SELECT @days = DATEDIFF(DAY, @tempdate, GETDATE())

DECLARE @Age NVARCHAR(50)
SET @Age = Cast(@years AS  NVARCHAR(4)) + ' Years
' + Cast(@months AS  NVARCHAR(2))+ ' Months ' +  Cast(@days AS  NVARCHAR(2))+ ' Days
Old'
RETURN @Age

End
```

Using the function in a query to get the expected output along with the age of the person.

```
Select Id, Name, DateOfBirth, dbo.fnComputeAge(DateOfBirth) as Age from tblEmployees
```

Cast and Convert functions in SQL Server - Part 28

To convert one data type to another, CAST and CONVERT functions can be used.

Syntax of CAST and CONVERT functions from MSDN:

```
CAST ( expression AS data_type [ ( length ) ] )
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

From the syntax, it is clear that CONVERT() function has an optional style parameter, where as CAST() function lacks this capability.

Consider the Employees Table below

Id	Name	DateOfBirth
1	Sam	1980-12-30 00:00:00.000
2	Pam	1982-09-01 12:02:36.260
3	John	1985-08-22 12:03:30.370
4	Sara	1979-11-29 12:59:30.670

The following 2 queries convert, **DateOfBirth's DateTime datatype** to **NVARCHAR**. The first query uses the CAST() function, and the second one uses CONVERT() function. The output is exactly the same for both the queries as shown below.

```
Select Id, Name, DateOfBirth, CAST(DateofBirth as nvarchar) as ConvertedDOB
from tblEmployees
Select Id, Name, DateOfBirth, Convert(nvarchar, DateOfBirth) as ConvertedDOB
from tblEmployees
```

Output:

Id	Name	DateOfBirth	ConvertedDOB
1	Sam	1980-12-30 00:00:00.000	Dec 30 1980 12:00AM
2	Pam	1982-09-01 12:02:36.260	Sep 1 1982 12:02PM
3	John	1985-08-22 12:03:30.370	Aug 22 1985 12:03PM
4	Sara	1979-11-29 12:59:30.670	Nov 29 1979 12:59PM

Now, let's use the **style** parameter of the CONVERT() function, to format the Date as we would like it. In the query below, we are using **103** as the argument for **style** parameter, which formats the date as **dd/mm/yyyy**.

```
Select Id, Name, DateOfBirth, Convert(nvarchar, DateOfBirth, 103) as ConvertedDOB
from tblEmployees
```

Output:

Id	Name	DateOfBirth	ConvertedDOB
1	Sam	1980-12-30 00:00:00.000	30/12/1980
2	Pam	1982-09-01 12:02:36.260	01/09/1982
3	John	1985-08-22 12:03:30.370	22/08/1985
4	Sara	1979-11-29 12:59:30.670	29/11/1979

The following table lists a few of the common DateTime styles:

Style	DateFormat
101	mm/dd/yyyy
102	yy.mm.dd
103	dd/mm/yyyy
104	dd.mm.yy
105	dd-mm-yy

For complete list of all the Date and Time Styles, please check MSDN.

To get just the date part, from DateTime

```
SELECT CONVERT(VARCHAR(10),GETDATE(),101)
```

In SQL Server 2008, Date datatype is introduced, so you can also use

```
SELECT CAST(GETDATE() as DATE)
SELECT CONVERT(DATE, GETDATE())
```

Note: To control the formatting of the Date part, DateTime has to be converted to NVARCHAR using the styles provided. When converting to DATE data type, the CONVERT() function will ignore the style parameter.

Now, let's write a query which produces the following output:

Id	Name	Name-Id
1	Sam	Sam - 1
2	Pam	Pam - 2
3	John	John - 3
4	Sara	Sara - 4

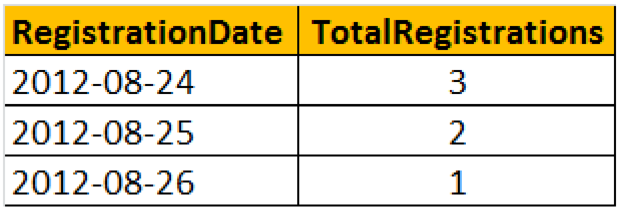
In this query, we are using CAST() function, to convert **Id (int)** to **nvarchar**, so it can be appended with the **NAME** column. If you remove the CAST() function, you will get an error stating - 'Conversion failed when converting the nvarchar value 'Sam - ' to data type int.'

```
Select Id, Name, Name + ' - ' + CAST(Id AS NVARCHAR) AS [Name-Id]
FROM tblEmployees
```

Now let's look at a practical example of using CAST function. Consider the registrations table below.

Id	Name	Email	RegisteredDate
1	John	j@j.com	2012-08-24 11:04:30.230
2	Sam	s@s.com	2012-08-25 14:04:29.780
3	Todd	t@t.com	2012-08-25 15:04:29.780
4	Mary	m@m.com	2012-08-24 15:04:30.730
5	Sunil	sunil@s.com	2012-08-24 15:05:30.330
6	Mike	mike@m.com	2012-08-26 15:05:30.330

Write a query which returns the total number of registrations by day



RegistrationDate	TotalRegistrations
2012-08-24	3
2012-08-25	2
2012-08-26	1

Query:
```
Select CAST(RegisteredDate as DATE) as RegistrationDate,
COUNT(Id) as TotalRegistrations
From tblRegistrations
Group By CAST(RegisteredDate as DATE)
```

The following are the differences between the 2 functions.
1. Cast is based on ANSI standard and Convert is specific to SQL Server. So, if **portability** is a concern and if you want to use the script with other database applications, use Cast().
2. Convert provides **more flexibility** than Cast. For example, it's possible to control how you want DateTime datatypes to be converted using styles with convert function.

The general guideline is to use CAST(), unless you want to take advantage of the style functionality in CONVERT().

Mathematical functions in sql server - Part 29

In this video session, we will understand the commonly used mathematical functions in sql server like, Abs, Ceiling, Floor, Power, Rand, Square, Sqrt, and Round functions

ABS (numeric_expression) - ABS stands for absolute and returns, the absolute (positive) number.

For example, `Select ABS(-101.5) -- returns 101.5, without the - sign.`
CEILING (numeric_expression) and FLOOR (numeric_expression)
CEILING and **FLOOR** functions accept a numeric expression as a single parameter. CEILING() returns the smallest integer value greater than or equal to the parameter, whereas FLOOR() returns the largest integer less than or equal to the parameter.

Examples:
```
Select CEILING(15.2) -- Returns 16
Select CEILING(-15.2) -- Returns -15

Select FLOOR(15.2) -- Returns 15
Select FLOOR(-15.2) -- Returns -16
```

Power(expression, power) - Returns the power value of the specified expression to the specified power.

Example: The following example calculates '2 TO THE POWER OF 3' = 2*2*2 = 8
```
Select POWER(2,3) -- Returns 8
```

RAND([Seed_Value]) - Returns a random float number between 0 and 1. Rand() function takes an optional seed parameter. When seed value is supplied the

RADN() function always returns the same value for the same seed.

Example:

```
Select RAND(1) -- Always returns the same value
```

If you want to generate a random number between 1 and 100, RAND() and FLOOR() functions can be used as shown below. Every time, you execute this query, you get a random number between 1 and 100.

```
Select FLOOR(RAND() * 100)
```

The following query prints 10 random numbers between 1 and 100.

```
Declare @Counter INT
Set @Counter = 1
While(@Counter <= 10)
Begin
 Print FLOOR(RAND() * 100)
 Set @Counter = @Counter + 1
End
```

SQUARE (Number) - Returns the square of the given number.

Example:

```
Select SQUARE(9) -- Returns 81
```

SQRT (Number) - SQRT stands for Square Root. This function returns the square root of the given value.

Example:

```
Select SQRT(81) -- Returns 9
```

ROUND (numeric_expression , length [,function]) - Rounds the given numeric expression based on the given length. This function takes 3 parameters.
1. Numeric_Expression is the number that we want to round.
2. Length parameter, specifies the number of the digits that we want to round to. If the length is a positive number, then the rounding is applied for the decimal part, where as if the length is negative, then the rounding is applied to the number before the decimal.
3. The optional function parameter, is used to indicate rounding or truncation operations. A value of 0, indicates rounding, where as a value of non zero indicates truncation. Default, if not specified is 0.

Examples:

```
-- Round to 2 places after (to the right) the decimal point
Select ROUND(850.556, 2) -- Returns 850.560

-- Truncate anything after 2 places, after (to the right) the decimal point
Select ROUND(850.556, 2, 1) -- Returns 850.550

-- Round to 1 place after (to the right) the decimal point
Select ROUND(850.556, 1) -- Returns 850.600

-- Truncate anything after 1 place, after (to the right) the decimal point
Select ROUND(850.556, 1, 1) -- Returns 850.500

-- Round the last 2 places before (to the left) the decimal point
```

```
Select ROUND(850.556, -2) -- 900.000

-- Round the last 1 place before (to the left) the decimal point
Select ROUND(850.556, -1) -- 850.000
```

Scalar User Defined Functions in sql server - Part 30

From **Parts 22 to 29**, we have learnt how to use many of the built-in system functions that are available in SQL Server. In this session, we will turn our attention, to creating **user defined functions**. In short **UDF**.

We will cover
1. User Defined Functions in sql server
2. Types of User Defined Functions
3. Creating a Scalar User Defined Function
4. Calling a Scalar User Defined Function
5. Places where we can use Scalar User Defined Function
6. Altering and Dropping a User Defined Function

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

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

To create a function, we use the following syntax:

```
CREATE FUNCTION Function_Name(@Parameter1 DataType, @Parameter2
DataType,..@Parametern Datatype)
RETURNS Return_Datatype
AS
BEGIN
    Function Body
    Return Return_Datatype
END
```

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

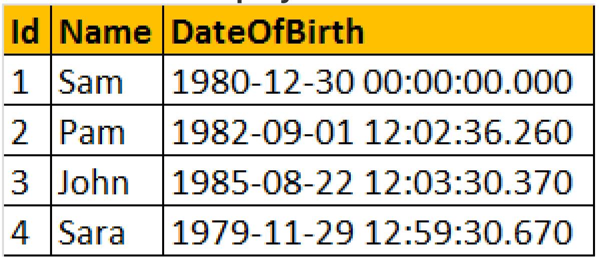
```
CREATE FUNCTION Age(@DOB Date)
RETURNS INT
AS
BEGIN
 DECLARE @Age INT
 SET @Age = DATEDIFF(YEAR, @DOB, GETDATE()) - CASE WHEN (MONTH(@DOB)
> MONTH(GETDATE())) OR (MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB)
> DAY(GETDATE())) THEN 1 ELSE 0 END
 RETURN @Age
END
```

When calling a scalar user-defined function, you must supply a two-part

name, **OwnerName.FunctionName**. **dbo** stands for database owner.
```
Select dbo.Age( dbo.Age('10/08/1982')
```

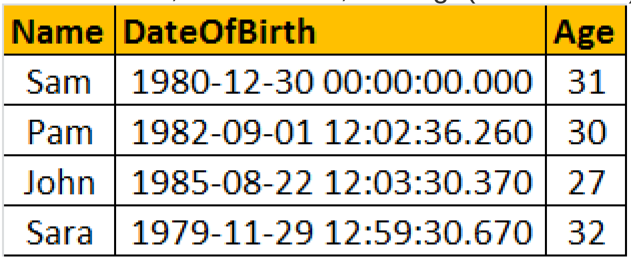
You can also invoke it using the complete 3 part name, DatabaseName.OwnerName.FunctionName.
```
Select SampleDB.dbo.Age('10/08/1982')
```

Consider the Employees table below.



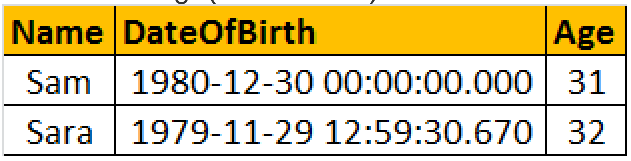
Id	Name	DateOfBirth
1	Sam	1980-12-30 00:00:00.000
2	Pam	1982-09-01 12:02:36.260
3	John	1985-08-22 12:03:30.370
4	Sara	1979-11-29 12:59:30.670

Scalar user defined functions can be used in the Select clause as shown below.
```
Select Name, DateOfBirth, dbo.Age(DateOfBirth) as Age from tblEmployees
```



Name	DateOfBirth	Age
Sam	1980-12-30 00:00:00.000	31
Pam	1982-09-01 12:02:36.260	30
John	1985-08-22 12:03:30.370	27
Sara	1979-11-29 12:59:30.670	32

Scalar user defined functions can be used in the Where clause, as shown below.
```
Select Name, DateOfBirth, dbo.Age(DateOfBirth) as Age
from tblEmployees
Where dbo.Age(DateOfBirth) > 30
```



Name	DateOfBirth	Age
Sam	1980-12-30 00:00:00.000	31
Sara	1979-11-29 12:59:30.670	32

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

To alter a function we use ALTER FUNCTION FuncationName statement and to delete it, we use DROP FUNCTION FuncationName.

To view the text of the function use sp_helptext FunctionName

Inline table valued functions - Part 31

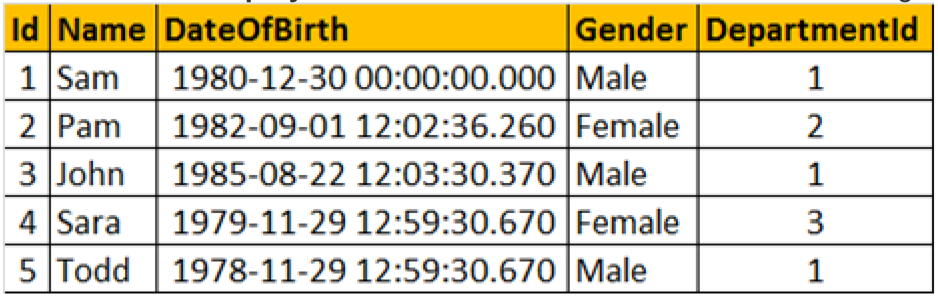
In Part 30 of this video series we have seen how to create and call '**scalar user defined functions**'. In this part of the video series, we will learn about '**Inline Table Valued Functions**'.

From Part 30, We learnt that, a scalar function, returns a **single** value. on the other hand, an Inline Table Valued function, return a **table**.

Syntax for creating an inline table valued function

```
CREATE FUNCTION Function_Name(@Param1 DataType, @Param2 DataType..., @ParamN
DataType)
RETURNS TABLE
AS
RETURN (Select_Statement)
```

Consider this Employees table shown below, which we will be using for our example.



Id	Name	DateOfBirth	Gender	DepartmentId
1	Sam	1980-12-30 00:00:00.000	Male	1
2	Pam	1982-09-01 12:02:36.260	Female	2
3	John	1985-08-22 12:03:30.370	Male	1
4	Sara	1979-11-29 12:59:30.670	Female	3
5	Todd	1978-11-29 12:59:30.670	Male	1

Create a function that returns EMPLOYEES by GENDER.

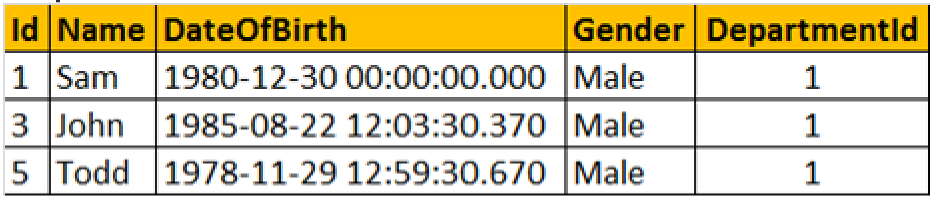
```
CREATE FUNCTION fn_EmployeesByGender(@Gender nvarchar(10))
RETURNS TABLE
AS
RETURN (Select Id, Name, DateOfBirth, Gender, DepartmentId
    from tblEmployees
    where Gender = @Gender)
```

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

Calling the user defined function

```
Select * from fn_EmployeesByGender('Male')
```

Output:



Id	Name	DateOfBirth	Gender	DepartmentId
1	Sam	1980-12-30 00:00:00.000	Male	1
3	John	1985-08-22 12:03:30.370	Male	1
5	Todd	1978-11-29 12:59:30.670	Male	1

As the inline user defined function, is returning a table, issue the select statement against the function, as if you are selecting the data from a TABLE.

Where can we use Inline Table Valued functions
1. Inline Table Valued functions can be used to achieve the functionality of parameterized views. We will talk about views, in a later session.

2. The table returned by the table valued function, can also be used in joins with other tables.

Consider the Departments Table

Id	DepartmentName	Location	DepartmentHead
1	IT	London	Rick
2	Payroll	Delhi	Ron
3	HR	New York	Christie
4	Other Department	Sydney	Cindrella

Joining the Employees returned by the function, with the Departments table

```
Select Name, Gender, DepartmentName
from fn_EmployeesByGender('Male') E
Join tblDepartment D on D.Id = E.DepartmentId
```

Executing the above query should produce the following output

Name	Gender	DepartmentName
Sam	Male	IT
John	Male	IT
Todd	Male	IT

New to joins in sql server. Please check the videos below
Part 12 - Basic Joins
Part 13 - Advanced Joins
Part 14 - Self Joins

Multi-Statement Table Valued Functions in SQL Server - Part 32

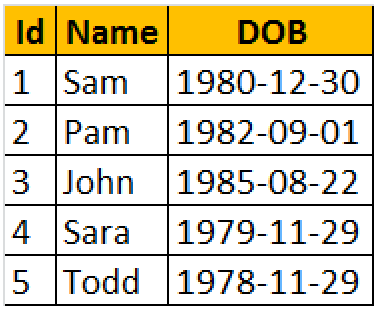
We have discussed about **scalar functions in Part 29** and **Inline Table Valued functions in Part 30**. In this video session, we will discuss about Multi-Statement Table Valued functions.

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

Employees Table:

Id	Name	DateOfBirth	Gender	DepartmentId
1	Sam	1980-12-30 00:00:00.000	Male	1
2	Pam	1982-09-01 12:02:36.260	Female	2
3	John	1985-08-22 12:03:30.370	Male	1
4	Sara	1979-11-29 12:59:30.670	Female	3
5	Todd	1978-11-29 12:59:30.670	Male	1

Let's write an Inline and multi-statement Table Valued functions that can return the output shown below.



Id	Name	DOB
1	Sam	1980-12-30
2	Pam	1982-09-01
3	John	1985-08-22
4	Sara	1979-11-29
5	Todd	1978-11-29

Inline Table Valued function(ILTVF):

```
Create Function fn_ILTVF_GetEmployees()
Returns Table
as
Return (Select Id, Name, Cast(DateOfBirth as Date) as DOB
        From tblEmployees)
```

Multi-statement Table Valued function(MSTVF):

```
Create Function fn_MSTVF_GetEmployees()
Returns @Table Table (Id int, Name nvarchar(20), DOB Date)
as
Begin
 Insert into @Table
 Select Id, Name, Cast(DateOfBirth as Date)
 From tblEmployees

 Return
End
```

Calling the Inline Table Valued Function:

```
Select * from fn_ILTVF_GetEmployees()
```

Calling the Multi-statement Table Valued Function:

```
Select * from fn_MSTVF_GetEmployees()
```

Now let's understand the differences between Inline Table Valued functions and Multi-statement Table Valued functions

1. In an Inline Table Valued function, the RETURNS clause cannot contain the structure of the table, the function returns. Where as, with the multi-statement table valued function, we specify the structure of the table that gets returned
2. Inline Table Valued function cannot have BEGIN and END block, where as the multi-statement function can have.
3. Inline Table valued functions are better for performance, than multi-statement table valued functions. If the given task, can be achieved using an inline table valued function, always prefer to use them, over multi-statement table valued functions.
4. It's possible to update the underlying table, using an inline table valued function, but not possible using multi-statement table valued function.

Updating the underlying table using inline table valued function:
This query will change **Sam** to **Sam1**, in the underlying table **tblEmployees**. When you try do the same thing with the multi-statement table valued function, you will get an error stating 'Object

'fn_MSTVF_GetEmployees' cannot be modified.'
Update fn_ILTVF_GetEmployees() set Name='Sam1' Where Id = 1

Reason for improved performance of an inline table valued function:
Internally, SQL Server treats an inline table valued function much like it would a view and treats a multi-statement table valued function similar to how it would a stored procedure.

Important concepts related to Functions in sql server - Part 33

All these concepts are asked in many interviews. Please watch the Parts 30, 31 and 32.
Scalar User Defined Functions - Part 30
Inline table valued functions - Part 31
Multi-Statement Table Valued Functions - Part 32

Deterministic and Nondeterministic Functions:
Deterministic functions always return the **same result** any time they are called with a specific set of input values and given the same state of the database.
Examples: Sum(), AVG(), Square(), Power() and Count()

Note: All aggregate functions are deterministic functions.

Nondeterministic functions may return **different results** each time they are called with a specific set of input values even if the database state that they access remains the same.
Examples: GetDate() and CURRENT_TIMESTAMP

Rand() function is a **Non-deterministic function**, but if you provide the **seed value**, the function becomes **deterministic**, as the same value gets returned for the same seed value.

We will be using tblEmployees table, for the rest of our examples. Please, create the table using this script.

```
CREATE TABLE [dbo].[tblEmployees]
(
 [Id] [int] Primary Key,
 [Name] [nvarchar](50) NULL,
 [DateOfBirth] [datetime] NULL,
 [Gender] [nvarchar](10) NULL,
 [DepartmentId] [int] NULL
)
```

Insert rows into the table using the insert script below.

```
Insert into tblEmployees values(1,'Sam','1980-12-30 00:00:00.000','Male',1)
Insert into tblEmployees values(2,'Pam','1982-09-01 12:02:36.260','Female',2)
Insert into tblEmployees values(3,'John','1985-08-22 12:03:30.370','Male',1)
Insert into tblEmployees values(4,'Sara','1979-11-29 12:59:30.670','Female',3)
Insert into tblEmployees values(5,'Todd','1978-11-29 12:59:30.670','Male',1)
```

Encrypting a function definiton using WITH ENCRYPTION OPTION:
We have learnt how to encrypt Stored procedure text using WITH ENCRYPTION OPTION in **Part 18 of this video series**. Along the same lines, you can also encrypt a function text. Once, encrypted, you cannot view the text of the function, using **sp_helptext** system stored procedure. If you try to, you will get a message stating 'The text for object is encrypted.' There are ways to decrypt, which is beyond the scope of this video.

Scalar Function without encryption option:

```
Create Function fn_GetEmployeeNameById(@Id int)
Returns nvarchar(20)
as
Begin
 Return (Select Name from tblEmployees Where Id = @Id)
End
```

To view text of the function:

```
sp_helptex fn_GetEmployeeNameById
```

Now, let's alter the function to use WITH ENCRYPTION OPTION

```
Alter Function fn_GetEmployeeNameById(@Id int)
Returns nvarchar(20)
With Encryption
as
Begin
 Return (Select Name from tblEmployees Where Id = @Id)
End
```

Now try to retrieve, the text of the function, using sp_helptex fn_GetEmployeeNameById.
You will get a message stating 'The text for object 'fn_GetEmployeeNameById' is encrypted.'

Creating a function WITH SCHEMABINDING option:
1. **The function fn_GetEmployeeNameById**(), is dependent on tblEmployees table.
2. Delete the table **tblEmployees** from the database.

```
Drop Table tblEmployees
```

3. Now, execute the function fn_GetEmployeeNameById(), you will get an error stating 'Invalid object name tblEmployees'. So, we are able to delete the table, while the function is still refrencing it.
4. Now, **recreate the table** and insert data, using the scripts provided.
5. Next, **Alter the function fn_GetEmployeeNameById()**, to use WITH SCHEMABINDING option.

```
Alter Function fn_GetEmployeeNameById(@Id int)
Returns nvarchar(20)
With SchemaBinding
as
Begin
 Return (Select Name from dbo.tblEmployees Where Id = @Id)
End
```

Note: You have to use the **2 part object name** i.e, dbo.tblEmployees, to use WITH SCHEMABINDING option. dbo is the schema name or owner name, tblEmployees is the table name.
6. Now, **try to drop the table using** - Drop Table tblEmployees. You will get a message stating, 'Cannot DROP TABLE tblEmployees because it is being referenced by object fn_GetEmployeeNameById.'

So, Schemabinding, specifies that the function is bound to the database objects that it references. When SCHEMABINDING is specified, the base objects cannot be modified in any way that would affect the function definition. The function definition itself must first be modified or dropped to remove dependencies on the object that is to be modified.

Temporary tables in SQL Server - Part 34

What are Temporary tables?
Temporary tables, are very similar to the permanent tables. Permanent tables get created in the database you specify, and remain in the database permanently, until you delete (drop) them. On the other hand, temporary tables get created in the TempDB and are automatically deleted, when they are no longer used.

Different Types of Temporary tables
In SQL Server, there are 2 types of Temporary tables - Local Temporary tables and Global Temporary tables.

How to Create a Local Temporary Table:
Creating a local Temporary table is very similar to creating a permanent table, except that you prefix the **table name with 1 pound (#) symbol**. In the example below, **#PersonDetails** is a local temporary table, with Id and Name columns.

```
Create Table #PersonDetails(Id int, Name nvarchar(20))
```

Insert Data into the temporary table:

```
Insert into #PersonDetails Values(1, 'Mike')
Insert into #PersonDetails Values(2, 'John')
Insert into #PersonDetails Values(3, 'Todd')
```

Select the data from the temporary table:

```
Select * from #PersonDetails
```

How to check if the local temporary table is created
Temporary tables are created in the TEMPDB. Query the sysobjects system table in TEMPDB. The name of the table, is suffixed with lot of underscores and a random number. For this reason you have to use the LIKE operator in the query.

```
Select name from tempdb..sysobjects
where name like '#PersonDetails%'
```

You can also check the existence of temporary tables using object explorer. In the object explorer, expand TEMPDB database folder, and then exapand TEMPORARY TABLES folder, and you should see the temporary table that we have created.

A local temporary table is available, only for the connection that has created the table. If you open another query window, and execute the following query you get an error stating 'Invalid object name #PersonDetails'. This proves that local temporary tables are available, only for the connection that has created them.

A local temporary table is automatically dropped, when the connection that has created the it, is closed. If the user wants to explicitly drop the temporary table, he can do so using

```
DROP TABLE #PersonDetails
```

If the temporary table, is created inside the stored procedure, it get's dropped automatically upon the completion of stored procedure execution. The stored procedure below, creates **#PersonDetails** temporary table, populates it and then finally returns the data and destroys the temporary table immediately after the completion of the stored procedure execution.

```
Create Procedure spCreateLocalTempTable
as
```

```
Begin
Create Table #PersonDetails(Id int, Name nvarchar(20))

Insert into #PersonDetails Values(1, 'Mike')
Insert into #PersonDetails Values(2, 'John')
Insert into #PersonDetails Values(3, 'Todd')

Select * from #PersonDetails
End
```

It is also possible for different connections, to create a local temporary table with the same name. For example User1 and User2, both can create a local temporary table with the same name #PersonDetails. Now, if you expand the Temporary Tables folder in the TEMPDB database, you should see 2 tables with name #PersonDetails and some random number at the end of the name. To differentiate between, the User1 and User2 local temp tables, sql server appends the random number at the end of the temp table name.

How to Create a Global Temporary Table:
To create a Global Temporary Table, prefix the name of the table with 2 pound (##) symbols. EmployeeDetails Table is the global temporary table, as we have prefixed it with 2 ## symbols.

```
Create Table ##EmployeeDetails(Id int, Name nvarchar(20))
```

Global temporary tables are visible to all the connections of the sql server, and are only destroyed when the last connection referencing the table is closed.

Multiple users, across multiple connections can have local temporary tables with the same name, but, a global temporary table name has to be unique, and if you inspect the name of the global temp table, in the object explorer, there will be no random numbers suffixed at the end of the table name.

Difference Between Local and Global Temporary Tables:
1. Local Temp tables are prefixed with single pound (#) symbol, where as gloabl temp tables are prefixed with 2 pound (##) symbols.

2. SQL Server appends some random numbers at the end of the local temp table name, where this is not done for global temp table names.

3. Local temporary tables are only visible to that session of the SQL Server which has created it, where as Global temporary tables are visible to all the SQL server sessions

4. Local temporary tables are automatically dropped, when the session that created the temporary tables is closed, where as Global temporary tables are destroyed when the last connection that is referencing the global temp table is closed.

Indexes in sql server - Part 35

Why indexes?
Indexes are used by queries to find data from tables quickly. Indexes are created on tables and views. Index on a table or a view, is very similar to an index that we find in a book.

If you don't have an index in a book, and I ask you to locate a specific chapter in that book, you will have to look at every page starting from the first page of the book.

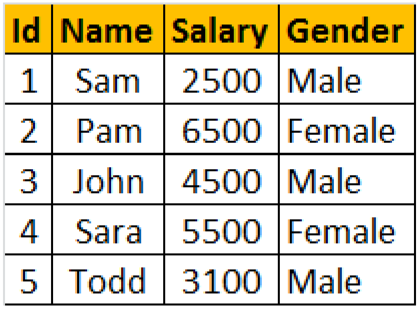
On, the other hand, if you have the index, you lookup the page number of the chapter in the index, and then directly go to that page number to locate the chapter.

Obviously, the book index is helping to drastically reduce the time it takes to find the chapter.

In a similar way, Table and View indexes, can help the query to find data quickly.

In fact, the existence of the right indexes, can drastically improve the performance of the query. If there is no index to help the query, then the query engine, checks every row in the table from the beginning to the end. This is called as Table Scan. Table scan is bad for performance.

Index Example: At the moment, the Employees table, does not have an index on SALARY column.



Id	Name	Salary	Gender
1	Sam	2500	Male
2	Pam	6500	Female
3	John	4500	Male
4	Sara	5500	Female
5	Todd	3100	Male

Consider, the following query

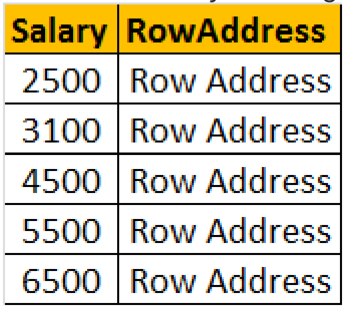
```
Select * from tblEmployee where Salary > 5000 and Salary < 7000
```

To find all the employees, who has salary **greater than 5000 and less than 7000**, the query engine has to check each and every row in the table, resulting in a table scan, which can adversely affect the performance, especially if the table is large. Since there is no index, to help the query, the query engine performs an entire table scan.

Now Let's Create the Index to help the query: Here, we are creating an index on Salary column in the employee table

```
CREATE Index IX_tblEmployee_Salary
ON tblEmployee (SALARY ASC)
```

The index stores salary of each employee, in the ascending order as shown below. The actual index may look slightly different.



Salary	RowAddress
2500	Row Address
3100	Row Address
4500	Row Address
5500	Row Address
6500	Row Address

Now, when the SQL server has to execute the same query, it has an index on the salary column to help this query. Salaries between the range of 5000 and 7000 are usually present at the bottom, since the salaries are arranged in an ascending order. SQL server picks up the row addresses from the index and directly fetch the records from the table, rather than scanning each row in the table. This is called as Index Seek.

An Index can also be created graphically using SQL Server Management Studio
1. In the Object Explorer, expand the Databases folder and then specific database you are working with.
2. Expand the Tables folder
3. Expand the Table on which you want to create the index
4. Right click on the Indexes folder and select New Index
5. In the New Index dialog box, type in a meaningful name
6. Select the Index Type and specify Unique or Non Unique Index
7. Click the Add
8. Select the columns that you want to add as index key
9 Click OK
10. Save the table

To view the Indexes: In the object explorer, expand Indexes folder. Alternatively use sp_helptext system stored procedure. The following command query returns all the indexes on tblEmployee table.

```
Execute sp_helptext tblEmployee
```

To delete or drop the index: When dropping an index, specify the table name as well

```
Drop Index tblEmployee.IX_tblEmployee_Salary
```

Clustered and Non-Clustered indexes - Part 36

Please watch Part 35 - Indexes in SQL Server, before continuing with this session

The following are the different types of indexes in SQL Server
1. Clustered
2. Nonclustered
3. Unique
4. Filtered
5. XML
6. Full Text
7. Spatial
8. Columnstore
9. Index with included columns
10. Index on computed columns

In this video session, we will talk about Clustered and Non-Clustered indexes.

Clustered Index:
A clustered index determines the physical order of data in a table. For this reason, a table can have only one clustered index.

Create tblEmployees table using the script below.

```
CREATE TABLE [tblEmployee]
(
 [Id] int Primary Key,
 [Name] nvarchar(50),
 [Salary] int,
 [Gender] nvarchar(10),
 [City] nvarchar(50)
)
```

Note that **Id** column is marked as **primary key**. Primary key, constraint create **clustered indexes automatically** if no clustered index already exists on the table and a nonclustered index is not specified when you create the PRIMARY KEY constraint.

To confirm this, execute sp_helpindex tblEmployee, which will show a unique clustered index created on the **Id** column.

Now execute the following insert queries. Note that, the values for Id column are not in a sequential order.

```
Insert into tblEmployee Values(3,'John',4500,'Male','New York')
Insert into tblEmployee Values(1,'Sam',2500,'Male','London')
Insert into tblEmployee Values(4,'Sara',5500,'Female','Tokyo')
Insert into tblEmployee Values(5,'Todd',3100,'Male','Toronto')
Insert into tblEmployee Values(2,'Pam',6500,'Female','Sydney')
```

Execute the following SELECT query

```
Select * from tblEmployee
```

Inspite, of inserting the rows in a random order, when we execute the select query we can see that all the rows in the table are arranged in an ascending order based on the Id column. This is because a clustered index determines the physical order of data in a table, and we have got a clustered index on the Id column.

Because of the fact that, a clustered index dictates the physical storage order of the data in a table, a table can contain only one clustered index. If you take the example of **tblEmployee** table, the data is already arranged by the Id column, and if we try to create another clustered index on the **Name column**, the data needs to be rearranged based on the **NAME column**, which will affect the ordering of rows that's already done based on the ID column.

For this reason, SQL server doesn't allow us to create more than one clustered index per table. The following SQL script, raises an error stating 'Cannot create more than one clustered index on table 'tblEmployee'. Drop the existing clustered index PK__tblEmplo__3214EC0706CD04F7 before creating another.'

```
Create Clustered Index IX_tblEmployee_Name
ON tblEmployee(Name)
```

A clustered index is analogous to a telephone directory, where the data is arranged by the last name. We just learnt that, a table can have only one clustered index. However, the index can contain multiple columns (a composite index), like the way a telephone directory is organized by last name and first name.

Let's now create a clustered index on 2 columns. To do this we first have to drop the existing clustered index on the Id column.

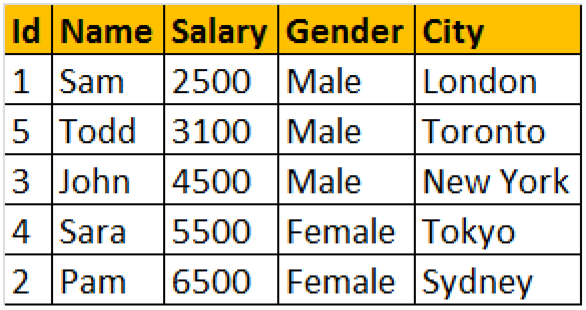
```
Drop index tblEmployee.PK__tblEmplo__3214EC070A9D95DB
```

When you execute this query, you get an error message stating 'An explicit DROP INDEX is not allowed on index 'tblEmployee.PK__tblEmplo__3214EC070A9D95DB'. It is being used for PRIMARY KEY constraint enforcement.' We will talk about the role of unique index in the next session. To successfully delete the clustered index, right click on the index in the Object explorer window and select DELETE.

Now, execute the following CREATE INDEX query, to create a composite clustered Index on the Gender and Salary columns.

```
Create Clustered Index IX_tblEmployee_Gender_Salary
ON tblEmployee(Gender DESC, Salary ASC)
```

Now, if you issue a select query against this table you should see the data physically arranged, FIRST by Gender in descending order and then by Salary in ascending order. The result is shown below.



Id	Name	Salary	Gender	City
1	Sam	2500	Male	London
5	Todd	3100	Male	Toronto
3	John	4500	Male	New York
4	Sara	5500	Female	Tokyo
2	Pam	6500	Female	Sydney

Non Clustered Index:
A nonclustered index is analogous to an index in a textbook. The data is stored in one place, the index in another place. The index will have pointers to the storage location of the data. Since, the nonclustered index is stored separately from the actual data, a table can have more than one non clustered index, just like how a book can have an index by Chapters at the beginning and another index by common terms at the end.

In the index itself, the data is stored in an ascending or descending order of the index key, which doesn't in any way influence the storage of data in the table.

The following SQL creates a Nonclustered index on the NAME column on tblEmployee table:

```
Create NonClustered Index IX_tblEmployee_Name
ON tblEmployee(Name)
```

Difference between Clustered and NonClustered Index:
1. **Only one clustered index per table**, where as you can have more than one non clustered index
2. **Clustered index is faster than a non clustered index**, because, the non-clustered index has to refer back to the table, if the selected column is not present in the index.
3. **Clustered index determines the storage order of rows in the table**, and hence doesn't require additional disk space, but where as a Non Clustered index is stored seperately from the table, additional storage space is required.

Unique and Non-Unique Indexes - Part 37

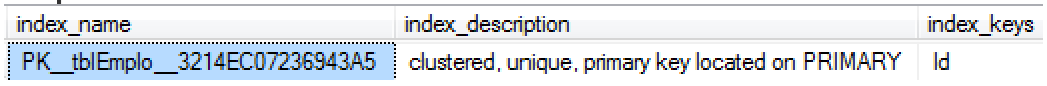
Suggested SQL Server Videos before watching this video
1. Part 9 - Unique Key Constraint
2. Part 35 - Index basics
3. Part 36 - Clustered and Nonclustered indexes

Unique index is used to enforce uniqueness of key values in the index. Let's understand this with an example.

Create the Employee table using the script below
```
CREATE TABLE [tblEmployee]
(
 [Id] int Primary Key,
 [FirstName] nvarchar(50),
 [LastName] nvarchar(50),
 [Salary] int,
 [Gender] nvarchar(10),
 [City] nvarchar(50)
)
```

Since, we have marked Id column, as the Primary key for this table, a UNIQUE CLUSTERED INDEX gets created on the Id column, with Id as the index key.

We can verify this by executing the sp_helpindex system stored procedure as shown below.
```
Execute sp_helpindex tblEmployee
```

Output:


index_name	index_description	index_keys
PK__tblEmplo__3214EC07236943A5	clustered, unique, primary key located on PRIMARY	Id

Since, we now have a UNIQUE CLUSTERED INDEX on the Id column, any attempt to duplicate the key values, will throw an error stating 'Violation of PRIMARY KEY constraint 'PK__tblEmplo__3214EC07236943A5'. Cannot insert duplicate key in object dbo.tblEmployee'

Example: The following insert queries will fail
```
Insert into tblEmployee Values(1,'Mike', 'Sandoz',4500,'Male','New York')
Insert into tblEmployee Values(1,'John', 'Menco',2500,'Male','London')
```

Now let's try to drop the Unique Clustered index on the Id column. This will raise an error stating - 'An explicit DROP INDEX is not allowed on index tblEmployee.PK__tblEmplo__3214EC07236943A5. It is being used for PRIMARY KEY constraint enforcement.'
```
Drop index tblEmployee.PK__tblEmplo__3214EC07236943A5
```

So this error message proves that, SQL server internally, uses the UNIQUE index to enforce the uniqueness of values and primary key.

Expand keys folder in the object explorer window, and you can see a primary key constraint. Now, expand the indexes folder and you should see a unique clustered index. In the object explorer it just shows the '**CLUSTERED**' word. To, confirm, this is infact an UNIQUE index, right click and select properties. The properties window, shows the UNIQUE checkbox being selected.

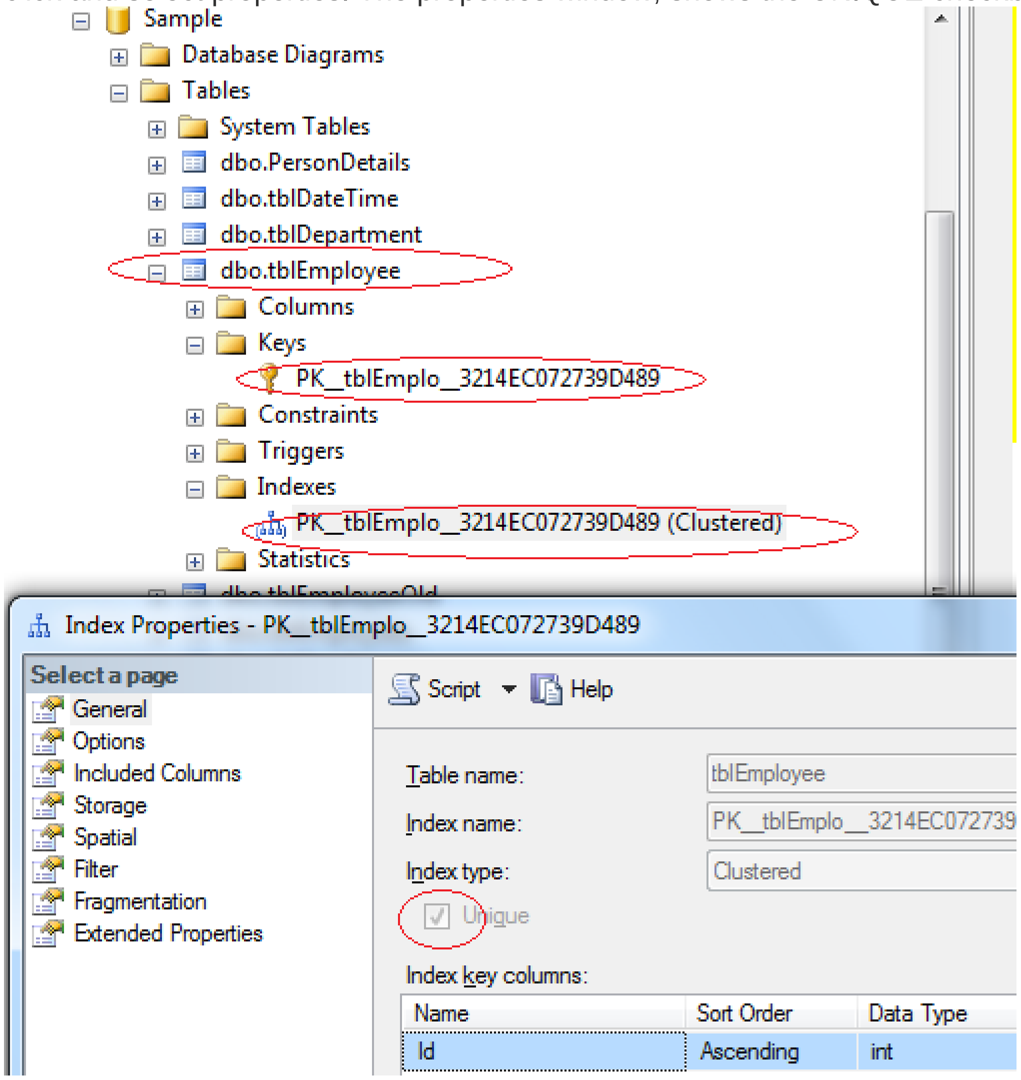


Table name: tblEmployee

Index name: PK__tblEmplo__3214EC072739

Index type: Clustered

[✓ Unique]

Index key columns:

Name	Sort Order	Data Type
Id	Ascending	int

SQL Server allows us to delete this UNIQUE CLUSTERED INDEX from the object explorer. so, Right click on the index, and select DELETE and finally, click OK. Along with the UNIQUE index, the primary key constraint is also deleted.

Now, let's try to insert duplicate values for the ID column. The rows should be accepted, without any primary key violation error.

```
Insert into tblEmployee Values(1,'Mike', 'Sandoz',4500,'Male','New York')
Insert into tblEmployee Values(1,'John', 'Menco',2500,'Male','London')
```

So, the UNIQUE index is used to enforce the uniqueness of values and primary key constraint.

UNIQUENESS is a property of an Index, and both CLUSTERED and NON-CLUSTERED indexes can be UNIQUE.

Creating a UNIQUE NON CLUSTERED index on the FirstName and LastName columns.

```
Create Unique NonClustered Index UIX_tblEmployee_FirstName_LastName
On tblEmployee(FirstName, LastName)
```

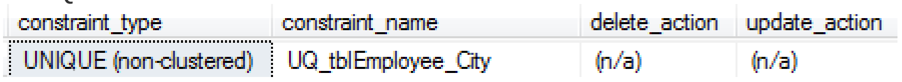
This unique non clustered index, ensures that no 2 entires in the index has the same first and last names. In Part 9, of this video series, we have learnt that, a Unique Constraint, can be used to enforce the uniqueness of values, across one or more columns. There are no major differences between a unique constraint and a unique index.

In fact, when you add a unique constraint, a unique index gets created behind the scenes. To prove this, let's add a unique constraint on the city column of the tblEmployee table.

```
ALTER TABLE tblEmployee
ADD CONSTRAINT UQ_tblEmployee_City
UNIQUE NONCLUSTERED (City)
```

At this point, we expect a unique constraint to be created. Refresh and Expand the constraints folder in the object explorer window. The constraint is not present in this folder. Now, refresh and expand the 'indexes' folder. In the indexes folder, you will see a UNIQUE NONCLUSTERED index with name UQ_tblEmployee_City.

Also, executing EXECUTE SP_HELPCONSTRAINT tblEmployee, lists the constraint as a UNIQUE NONCLUSTERED index.



constraint_type	constraint_name	delete_action	update_action
UNIQUE (non-clustered)	UQ_tblEmployee_City	(n/a)	(n/a)

So creating a UNIQUE constraint, actually creates a UNIQUE index. So a UNIQUE index can be created explicitly, using CREATE INDEX statement or indirectly using a UNIQUE constraint. So, when should you be creating a Unique constraint over a unique index.To make our intentions clear, create a unique constraint, when data integrity is the objective. This makes the objective of the index very clear. In either cases, data is validated in the same manner, and the query optimizer does not differentiate between a unique index created by a unique constraint or manually created.

Note:
1. By default, a PRIMARY KEY constraint, creates a unique clustered index, where as a UNIQUE constraint creates a unique nonclustered index. These defaults can be changed if you wish to.

2. A UNIQUE constraint or a UNIQUE index cannot be created on an existing table, if the table contains duplicate values in the key columns. Obviously, to solve this,remove the key columns from the index definition or delete or update the duplicate values.

3. By default, duplicate values are not allowed on key columns, when you have a unique index or constraint. For, example, if I try to insert 10 rows, out of which 5 rows contain duplicates, then all the 10 rows are rejected. However, if I want only the 5 duplicate rows to be rejected and accept the non-duplicate 5 rows, then I can use IGNORE_DUP_KEY option. An example of using IGNORE_DUP_KEY option is shown below.

```
CREATE UNIQUE INDEX IX_tblEmployee_City
ON tblEmployee(City)
WITH IGNORE_DUP_KEY
```

Advantages and disadvantages of indexes - Part 38

Suggested SQL Server Videos before watching this video
Part 35 - Index basics
Part 36 - Clustered and Nonclustered indexes
Part 37 - Unique and Non-Unique Indexes

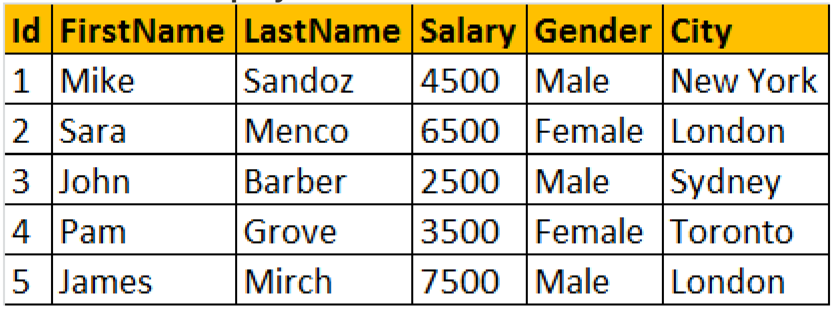
In this video session, we talk about the advantages and disadvantages of indexes. We wil also talk about a concept called **covering queries**.

In Part 35, we have learnt that, Indexes are used by queries to find data quickly. In this part, we will learn about the different queries that can benefit from indexes.

Create Employees table
```
CREATE TABLE [tblEmployee]
(
 [Id] int Primary Key,
 [FirstName] nvarchar(50),
 [LastName] nvarchar(50),
 [Salary] int,
 [Gender] nvarchar(10),
 [City] nvarchar(50)
)
```

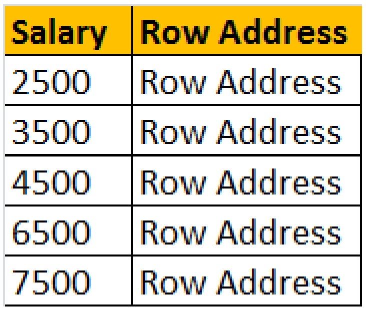
Insert sample data:
```
Insert into tblEmployee Values(1,'Mike', 'Sandoz',4500,'Male','New York')
Insert into tblEmployee Values(2,'Sara', 'Menco',6500,'Female','London')
Insert into tblEmployee Values(3,'John', 'Barber',2500,'Male','Sydney')
Insert into tblEmployee Values(4,'Pam', 'Grove',3500,'Female','Toronto')
Insert into tblEmployee Values(5,'James', 'Mirch',7500,'Male','London')
```

Create a Non-Clustered Index on Salary Column
```
Create NonClustered Index IX_tblEmployee_Salary
On tblEmployee (Salary Asc)
```

Data from tblEmployee table


Id	FirstName	LastName	Salary	Gender	City
1	Mike	Sandoz	4500	Male	New York
2	Sara	Menco	6500	Female	London
3	John	Barber	2500	Male	Sydney
4	Pam	Grove	3500	Female	Toronto
5	James	Mirch	7500	Male	London

NonClustered Index



Salary	Row Address
2500	Row Address
3500	Row Address
4500	Row Address
6500	Row Address
7500	Row Address

The following select query benefits from the index on the Salary column, because the salaries are sorted in ascending order in the index. From the index, it's easy to identify the records where salary is between 4000 and 8000, and using the row address the corresponding records from the table can be fetched quickly.

```
Select * from tblEmployee where Salary > 4000 and Salary < 8000
```

Not only, the SELECT statement, even the following DELETE and UPDATE statements can also benefit from the index. To update or delete a row, SQL server needs to first find that row, and the index can help in searching and finding that specific row quickly.

```
Delete from tblEmployee where Salary = 2500
Update tblEmployee Set Salary = 9000 where Salary = 7500
```

Indexes can also help queries, that ask for sorted results. Since the Salaries are already sorted, the database engine, simply scans the index from the first entry to the last entry and retrieve the rows in sorted order. This avoids, sorting of rows during query execution, which can significantly imrpove the processing time.

```
Select * from tblEmployee order by Salary
```

The index on the Salary column, can also help the query below, by scanning the index in reverse order.

```
Select * from tblEmployee order by Salary Desc
```

GROUP BY queries can also benefit from indexes. To group the Employees with the same salary, the query engine, can use the index on Salary column, to retrieve the already sorted salaries. Since matching salaries are present in consecutive index entries, it is to count the total number of Employees at each Salary quickly.

```
Select Salary, COUNT(Salary) as Total
from tblEmployee
Group By Salary
```

Diadvantages of Indexes:
Additional Disk Space: Clustered Index does not, require any additional storage. Every Non-Clustered index requires additional space as it is stored separately from the table.The amount of space required will depend on the size of the table, and the number and types of columns used in the index.

Insert Update and Delete statements can become slow: When **DML** (Data Manipulation Language) statements (**INSERT, UPDATE, DELETE**) modifies data in a table, the data in all the indexes also needs to be updated. Indexes can help, to search and locate the rows, that we want to delete, but too many indexes to update can actually hurt the performance of data modifications.

What is a covering query?

If all the columns that you have requested in the SELECT clause of query, are present in the index, then there is no need to lookup in the table again. The requested columns data can simply be returned from the index.

A clustered index, always covers a query, since it contains all of the data in a table. A composite index is an index on two or more columns. Both clustered and nonclustered indexes can be composite indexes. To a certain extent, a composite index, can cover a query.

Views in sql server - Part 39

What is a View?
A view is nothing more than a **saved SQL query**. A view can also be considered as a **virtual table**.

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

SQL Script to create tblEmployee table:
```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Salary int,
  Gender nvarchar(10),
  DepartmentId int
)
```

SQL Script to create tblDepartment table:
```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

Insert data into tblDepartment table
```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

Insert data into tblEmployee table
```
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)
Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
Insert into tblEmployee values (4,'Todd', 4800, 'Male', 4)
Insert into tblEmployee values (5,'Sara', 3200, 'Female', 1)
Insert into tblEmployee values (6,'Ben', 4800, 'Male', 3)
```

At this point Employees and Departments table should look like this.
Employees Table:

Id	Name	Salary	Gender	DepartmentId
1	John	5000	Male	3
2	Mike	3400	Male	2
3	Pam	6000	Female	1
4	Todd	4800	Male	4
5	Sara	3200	Female	1
6	Ben	4800	Male	3

Departments Table:

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

Now, let's write a Query which returns the output as shown below:

Id	Name	Salary	Gender	DeptName
1	John	5000	Male	HR
2	Mike	3400	Male	Payroll
3	Pam	6000	Female	IT
4	Todd	4800	Male	Admin
5	Sara	3200	Female	IT
6	Ben	4800	Male	HR

To get the expected output, we need to join **tblEmployees** table with **tblDepartments** table. If you are new to joins, please click here to view the video on Joins in SQL Server.

```
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

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

```
Create View vWEmployeesByDepartment
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

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

```
SELECT * from vWEmployeesByDepartment
```

When this query is executed, the database engine actually retrieves the data from the underlying base tables, **tblEmployees and tblDepartments**. The View itself, doesnot store any data by default. However, we can change this default behaviour, which we will talk about in a later session. So, this is the reason, a view is considered, as just, a stored query or a virtual table.

Advantages of using views:
1. Views can be used to reduce the **complexity of the database schema**, for non IT users. The sample view, **vWEmployeesByDepartment**, hides the complexity of joins. Non-IT users, finds it easy to query the view, rather than writing complex joins.

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

View that returns only IT department employees:
```
Create View vWITDepartment_Employees
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
where tblDepartment.DeptName = 'IT'
```

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

View that returns all columns except Salary column:
```
Create View vWEmployeesNonConfidentialData
as
Select Id, Name, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

3. Views can be used to present **only aggregated data** and **hide detailed data**.

View that returns summarized data, Total number of employees by Department.
```
Create View vWEmployeesCountByDepartment
as
Select DeptName, COUNT(Id) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
Group By DeptName
```

To look at view definition - sp_helptext vWName
To modify a view - ALTER VIEW statement
To Drop a view - DROP VIEW vWName

Updateable Views - Part 40

In Part 39, we have discussed the basics of views. In this session we will learn about Updateable Views. Let's create **tblEmployees** table and populate it with some sample data.

Create Table tblEmployee Script:

```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Salary int,
  Gender nvarchar(10),
  DepartmentId int
)
```

Script to insert data:

```
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)
Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
Insert into tblEmployee values (4,'Todd', 4800, 'Male', 4)
Insert into tblEmployee values (5,'Sara', 3200, 'Female', 1)
Insert into tblEmployee values (6,'Ben', 4800, 'Male', 3)
```

Let's create a view, which returns all the columns from the tblEmployees table, except Salary column.

```
Create view vWEmployeesDataExceptSalary
as
Select Id, Name, Gender, DepartmentId
from tblEmployee
```

Select data from the view: A view does not store any data. So, when this query is executed, the database engine actually retrieves data, from the underlying tblEmployee base table.

```
Select * from vWEmployeesDataExceptSalary
```

Is it possible to Insert, Update and delete rows, from the underlying tblEmployees table, using view vWEmployeesDataExceptSalary?
Yes, SQL server views are updateable.

The following query updates, Name column from Mike to Mikey. Though, we are updating the view, SQL server, correctly updates the base table tblEmployee. To verify, execute, SELECT statement, on tblEmployee table.

```
Update vWEmployeesDataExceptSalary
Set Name = 'Mikey' Where Id = 2
```

Along the same lines, it is also possible to insert and delete rows from the base table using views.

```
Delete from vWEmployeesDataExceptSalary where Id = 2
Insert into vWEmployeesDataExceptSalary values (2, 'Mikey', 'Male', 2)
```

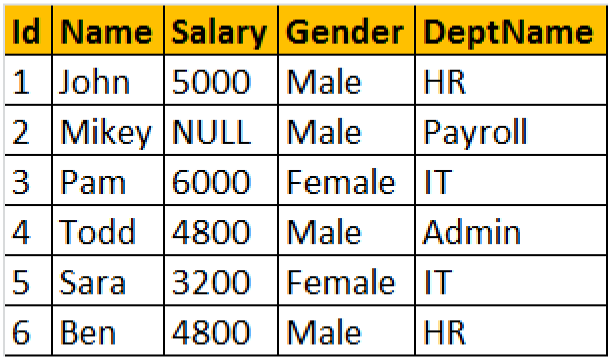
Now, let us see, what happens if our view is based on multiple base tables. For this purpose, let's create tblDepartment table and populate with some sample data.
SQL Script to create tblDepartment table

```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

Insert data into tblDepartment table

```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

Create a view which joins tblEmployee and tblDepartment tables, and return the result as shown below.



Id	Name	Salary	Gender	DeptName
1	John	5000	Male	HR
2	Mikey	NULL	Male	Payroll
3	Pam	6000	Female	IT
4	Todd	4800	Male	Admin
5	Sara	3200	Female	IT
6	Ben	4800	Male	HR

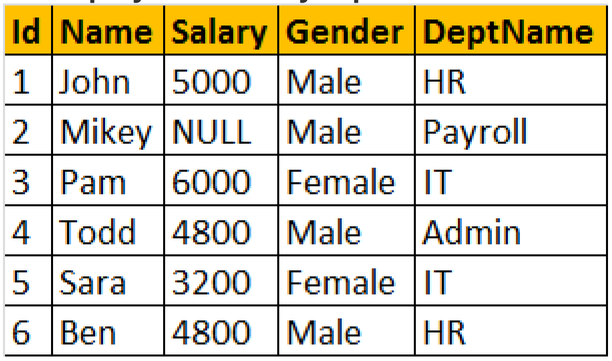
View that joins tblEmployee and tblDepartment

```
Create view vwEmployeeDetailsByDepartment
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

Select Data from view vwEmployeeDetailsByDepartment

```
Select * from vwEmployeeDetailsByDepartment
```

vwEmployeeDetailsByDepartment Data:



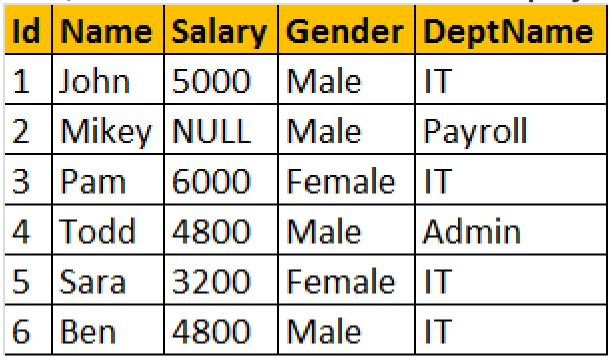
Id	Name	Salary	Gender	DeptName
1	John	5000	Male	HR
2	Mikey	NULL	Male	Payroll
3	Pam	6000	Female	IT
4	Todd	4800	Male	Admin
5	Sara	3200	Female	IT
6	Ben	4800	Male	HR

Now, let's update, John's department, from HR to IT. At the moment, there are 2 employees (Ben, and John) in the HR department.

```
Update vwEmployeeDetailsByDepartment
```

```
set DeptName='IT' where Name = 'John'
```

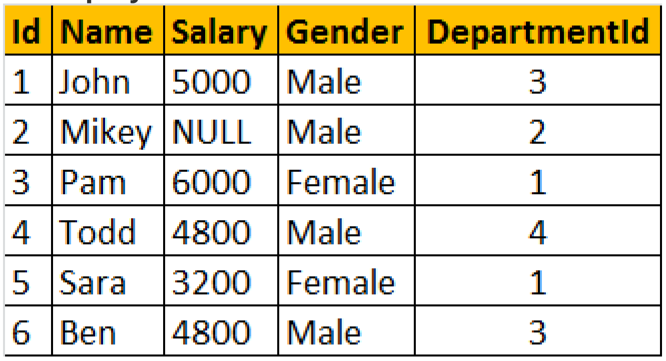
Now, Select data from the view vwEmployeeDetailsByDepartment:



Id	Name	Salary	Gender	DeptName
1	John	5000	Male	IT
2	Mikey	NULL	Male	Payroll
3	Pam	6000	Female	IT
4	Todd	4800	Male	Admin
5	Sara	3200	Female	IT
6	Ben	4800	Male	IT

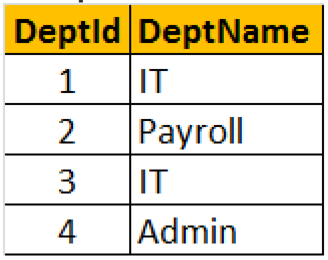
Notice, that Ben's department is also changed to IT. To understand the reasons for incorrect UPDATE, select Data from tblDepartment and tblEmployee base tables.

tblEmployee Table



Id	Name	Salary	Gender	DepartmentId
1	John	5000	Male	3
2	Mikey	NULL	Male	2
3	Pam	6000	Female	1
4	Todd	4800	Male	4
5	Sara	3200	Female	1
6	Ben	4800	Male	3

tblDepartment



DeptId	DeptName
1	IT
2	Payroll
3	IT
4	Admin

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

We will discuss about triggers and correctly updating a view that is based on multiple tables, in a later video session.

Indexed views in sql server - Part 41

Suggested SQL Server Videos before watching this Video
1. Part 39 - Views in sql server
2. Part 40 - Updateable views in sql server

In **Part 39**, we have covered the basics of views and in **Part 40**, we have seen, how to update the underlying base tables thru a view. In this video session, we will learn about INDEXED VIEWS.

What is an Indexed View or What happens when you create an Index on a view?
A **standard** or **Non-indexed** view, is just a stored SQL query. When, we try to retrieve data from the view, the data is actually retrieved from the underlying base tables. So, a view is just a virtual table it does not store any data, by default.

However, when you create an index, on a view, the view gets materialized. This means, the view is now, capable of storing data. In SQL server, we call them Indexed views and in Oracle, Materialized views.

Let's now, look at an example of creating an Indexed view. For the purpose of this video, we will be using **tblProduct** and **tblProductSales** tables.

Script to create table tblProduct
```
Create Table tblProduct
(
 ProductId int primary key,
 Name nvarchar(20),
 UnitPrice int
)
```

Script to pouplate tblProduct, with sample data
```
Insert into tblProduct Values(1, 'Books', 20)
Insert into tblProduct Values(2, 'Pens', 14)
Insert into tblProduct Values(3, 'Pencils', 11)
Insert into tblProduct Values(4, 'Clips', 10)
```

Script to create table tblProductSales
```
Create Table tblProductSales
(
 ProductId int,
 QuantitySold int
)
```

Script to pouplate tblProductSales, with sample data
```
Insert into tblProductSales values(1, 10)
Insert into tblProductSales values(3, 23)
Insert into tblProductSales values(4, 21)
Insert into tblProductSales values(2, 12)
Insert into tblProductSales values(1, 13)
Insert into tblProductSales values(3, 12)
Insert into tblProductSales values(4, 13)
Insert into tblProductSales values(1, 11)
Insert into tblProductSales values(2, 12)
Insert into tblProductSales values(1, 14)
```

tblProduct Table

ProductId	Name	UnitPrice
1	Books	20
2	Pens	14
3	Pencils	11
4	Clips	10

tblProductSales Table

ProductId	QuantitySold
1	10
3	23
4	21
2	12
1	13
3	12
4	13
1	11
2	12
1	14

Create a view which returns Total Sales and Total Transactions by Product. The output should be, as shown below.

Name	TotalSales	TotalTransactions
Books	960	4
Clips	340	2
Pencils	385	2
Pens	336	2

Script to create view vWTotalSalesByProduct

```
Create view vWTotalSalesByProduct
with SchemaBinding
as
Select Name,
SUM(ISNULL((QuantitySold * UnitPrice), 0)) as TotalSales,
COUNT_BIG(*) as TotalTransactions
from dbo.tblProductSales
join dbo.tblProduct
on dbo.tblProduct.ProductId = dbo.tblProductSales.ProductId
group by Name
```

If you want to create an Index, on a view, the following rules should be followed by the view. For the complete list of all rules, please check MSDN.
1. The view should be created with SchemaBinding option

2. If an Aggregate function in the SELECT LIST, references an expression, and if there is a possibility for that expression to become NULL, then, a replacement value should be specified. In this example, we are using, ISNULL() function, to replace NULL values with ZERO.

3. If GROUP BY is specified, the view select list must contain a COUNT_BIG(*) expression

4. The base tables in the view, should be referenced with 2 part name. In this example, tblProduct and tblProductSales are referenced using dbo.tblProduct and dbo.tblProductSales respectively.

Now, let's create an Index on the view:
The first index that you create on a view, must be a unique clustered index. After the unique clustered index has been created, you can create additional nonclustered indexes.
```
Create Unique Clustered Index UIX_vWTotalSalesByProduct_Name
on vWTotalSalesByProduct(Name)
```

Since, we now have an index on the view, the view gets materialized. The data is stored in the view. So when we execute Select * from vWTotalSalesByProduct, the data is retrurned from the view itself, rather than retrieving data from the underlying base tables.

Indexed views, can significantly improve the performance of queries that involves JOINS and Aggeregations. The cost of maintaining an indexed view is much higher than the cost of maintaining a table index.

Indexed views are ideal for scenarios, where the underlying data is not frequently changed. Indexed views are more often used in OLAP systems, because the data is mainly used for reporting and analysis purposes. Indexed views, may not be suitable for OLTP systems, as the data is frequently addedd and changed.

Limitations of views - Part 42

Suggested SQL Server Videos before watching this Video
Part 39 - View basics
Part 40 - Updateable views
Part 41 - Indexed views

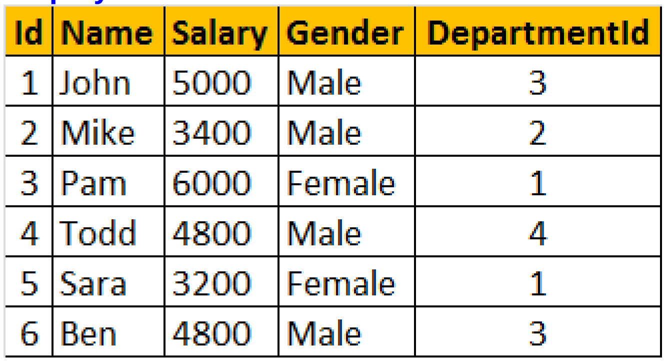
1. **You cannot pass parameters to a view**. Table Valued functions are an excellent replacement for parameterized views.

   We will use tblEmployee table for our examples. SQL Script to create tblEmployee table:
   ```
   CREATE TABLE tblEmployee
   (
     Id int Primary Key,
     Name nvarchar(30),
     Salary int,
     Gender nvarchar(10),
     DepartmentId int
   )
   ```

   Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)
Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
Insert into tblEmployee values (4,'Todd', 4800, 'Male', 4)
Insert into tblEmployee values (5,'Sara', 3200, 'Female', 1)
Insert into tblEmployee values (6,'Ben', 4800, 'Male', 3)
```

Employee Table



Id	Name	Salary	Gender	DepartmentId
1	John	5000	Male	3
2	Mike	3400	Male	2
3	Pam	6000	Female	1
4	Todd	4800	Male	4
5	Sara	3200	Female	1
6	Ben	4800	Male	3

```
-- Error : Cannot pass Parameters to Views
Create View vWEmployeeDetails
@Gender nvarchar(20)
as
Select Id, Name, Gender, DepartmentId
from  tblEmployee
where Gender = @Gender
```

Table Valued functions can be used as a replacement for parameterized views.

```
Create function fnEmployeeDetails(@Gender nvarchar(20))
Returns Table
as
Return
(Select Id, Name, Gender, DepartmentId
from tblEmployee where Gender = @Gender)
```

Calling the function

```
Select * from dbo.fnEmployeeDetails('Male')
```

2. Rules and Defaults cannot be associated with views.

3. The ORDER BY clause is invalid in views unless TOP or FOR XML is also specified.

```
Create View vWEmployeeDetailsSorted
as
Select Id, Name, Gender, DepartmentId
from tblEmployee
order by Id
```

If you use ORDER BY, you will get an error stating - 'The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and common table expressions, unless TOP or FOR XML is also specified.'

4. Views cannot be based on temporary tables.

```
Create Table ##TestTempTable(Id int, Name nvarchar(20), Gender nvarchar(10))

Insert into ##TestTempTable values(101, 'Martin', 'Male')
Insert into ##TestTempTable values(102, 'Joe', 'Female')
Insert into ##TestTempTable values(103, 'Pam', 'Female')
Insert into ##TestTempTable values(104, 'James', 'Male')

-- Error: Cannot create a view on Temp Tables
Create View vwOnTempTable
as
Select Id, Name, Gender
from ##TestTempTable
```

DML Triggers - Part 43

In SQL server there are 3 types of triggers
1. DML triggers
2. DDL triggers
3. Logon trigger

We will discuss about DDL and logon triggers in a later session. In this video, we will learn about DML triggers.

In general, a trigger is a special kind of stored procedure that automatically executes when an event occurs in the database server.

DML stands for Data Manipulation Language. INSERT, UPDATE, and DELETE statements are DML statements. DML triggers are fired, when ever data is modified using INSERT, UPDATE, and DELETE events.

DML triggers can be again classified into 2 types.
1. After triggers (Sometimes called as FOR triggers)
2. Instead of triggers

After triggers, as the name says, fires after the triggering action. The INSERT, UPDATE, and DELETE statements, causes an after trigger to fire after the respective statements complete execution.

On ther hand, as the name says, INSTEAD of triggers, fires instead of the triggering action. The INSERT, UPDATE, and DELETE statements, can cause an INSTEAD OF trigger to fire INSTEAD OF the respective statement execution.

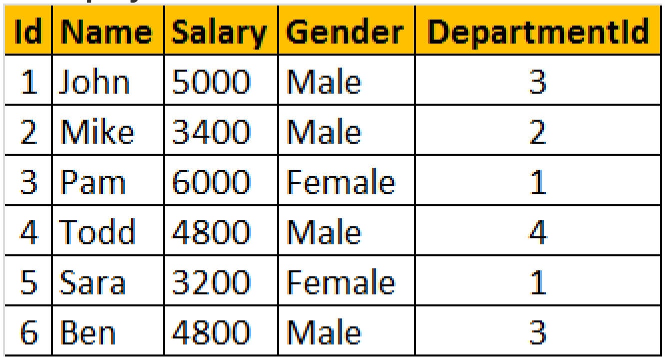
We will use tblEmployee and tblEmployeeAudit tables for our examples

SQL Script to create tblEmployee table:
```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Salary int,
  Gender nvarchar(10),
  DepartmentId int
)
```

Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)
Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
```

tblEmployee



Id	Name	Salary	Gender	DepartmentId
1	John	5000	Male	3
2	Mike	3400	Male	2
3	Pam	6000	Female	1
4	Todd	4800	Male	4
5	Sara	3200	Female	1
6	Ben	4800	Male	3

SQL Script to create tblEmployeeAudit table:

```
CREATE TABLE tblEmployeeAudit
(
  Id int identity(1,1) primary key,
  AuditData nvarchar(1000)
)
```

When ever, a new Employee is added, we want to capture the ID and the date and time, the new employee is added in tblEmployeeAudit table. The easiest way to achieve this, is by having an AFTER TRIGGER for INSERT event.

Example for AFTER TRIGGER for INSERT event on tblEmployee table:

```
CREATE TRIGGER tr_tblEMployee_ForInsert
ON tblEmployee
FOR INSERT
AS
BEGIN
 Declare @Id int
 Select @Id = Id from inserted

 insert into tblEmployeeAudit
 values('New employee with Id  = ' + Cast(@Id as nvarchar(5)) + ' is added at ' + cast(Getdate() as
nvarchar(20)))
END
```

In the trigger, we are getting the id from inserted table. So, what is this inserted table? INSERTED table, is a special table used by DML triggers. When you add a new row into tblEmployee table, a copy of the row will also be made into inserted table, which only a trigger can access. You cannot access this table outside the context of the trigger. The structure of the inserted table will be identical to the structure of tblEmployee table.

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

into tblEmployeeAudit, is also inserted.
Insert into tblEmployee values (7,'Tan', 2300, 'Female', 3)

Along, the same lines, let us now capture audit information, when a row is deleted from the table, tblEmployee.
Example for AFTER TRIGGER for DELETE event on tblEmployee table:
```
CREATE TRIGGER tr_tblEMployee_ForDelete
ON tblEmployee
FOR DELETE
AS
BEGIN
 Declare @Id int
 Select @Id = Id from deleted

 insert into tblEmployeeAudit
 values('An existing employee with Id  = ' + Cast(@Id as nvarchar(5)) + ' is deleted at
' + Cast(Getdate() as nvarchar(20)))
END
```

The only difference here is that, we are specifying, the triggering event as **DELETE** and retrieving the deleted row ID from **DELETED** table. DELETED table, is a special table used by DML triggers. When you delete a row from tblEmployee table, a copy of the deleted row will be made available in DELETED table, which only a trigger can access. Just like INSERTED table, DELETED table cannot be accessed, outside the context of the trigger and, the structure of the DELETED table will be identical to the structure of tblEmployee table.

In the next session, we will talk about AFTER trigger for UPDATE event.

After update trigger - Part 44

This video is a continuation of Part - 43, Please watch Part 43, before watching this video.

Triggers make use of 2 special tables, INSERTED and DELETED. The inserted table contains the updated data and the deleted table contains the old data. The After trigger for UPDATE event, makes use of both inserted and deleted tables.

Create AFTER UPDATE trigger script:
```
Create trigger tr_tblEmployee_ForUpdate
on tblEmployee
for Update
as
Begin
 Select * from deleted
 Select * from inserted
End
```

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

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

The following AFTER UPDATE trigger, audits employee information upon UPDATE, and stores the audit data in tblEmployeeAudit table.

```
Alter trigger tr_tblEmployee_ForUpdate
on tblEmployee
for Update
as
Begin
    -- Declare variables to hold old and updated data
    Declare @Id int
    Declare @OldName nvarchar(20), @NewName nvarchar(20)
    Declare @OldSalary int, @NewSalary int
    Declare @OldGender nvarchar(20), @NewGender nvarchar(20)
    Declare @OldDeptId int, @NewDeptId int

    -- Variable to build the audit string
    Declare @AuditString nvarchar(1000)

    -- Load the updated records into temporary table
    Select *
    into #TempTable
    from inserted

    -- Loop thru the records in temp table
    While(Exists(Select Id from #TempTable))
    Begin
        --Initialize the audit string to empty string
        Set @AuditString = ''

        -- Select first row data from temp table
        Select Top 1 @Id = Id, @NewName = Name,
        @NewGender = Gender, @NewSalary = Salary,
        @NewDeptId = DepartmentId
        from #TempTable

        -- Select the corresponding row from deleted table
        Select @OldName = Name, @OldGender = Gender,
        @OldSalary = Salary, @OldDeptId = DepartmentId
        from deleted where Id = @Id

   -- Build the audit string dynamically
        Set @AuditString = 'Employee with Id = ' + Cast(@Id as nvarchar(4)) + ' changed'
        if(@OldName <> @NewName)
            Set @AuditString = @AuditString + ' NAME from ' + @OldName + ' to ' + @NewName

        if(@OldGender <> @NewGender)
            Set @AuditString = @AuditString + ' GENDER from ' + @OldGender + ' to ' +
@NewGender

        if(@OldSalary <> @NewSalary)
            Set @AuditString = @AuditString + ' SALARY from ' + Cast(@OldSalary as
nvarchar(10))+ ' to ' + Cast(@NewSalary as nvarchar(10))

   if(@OldDeptId <> @NewDeptId)
```

```
                Set @AuditString = @AuditString + ' DepartmentId from ' + Cast(@OldDeptId as
nvarchar(10))+ ' to ' + Cast(@NewDeptId as nvarchar(10))

          insert into tblEmployeeAudit values(@AuditString)

          -- Delete the row from temp table, so we can move to the next row
          Delete from #TempTable where Id = @Id
     End
End
```

Instead of insert trigger - Part 45

Suggested SQL Server Videos before watching this Video
Part 39 - Views
Part 40 - Updateable Views
Part 43 - DML triggers
Part 44 - DML After Update Trigger

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

We will base our demos on Employee and Department tables. So, first, let's create these 2 tables.

SQL Script to create tblEmployee table:
```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Gender nvarchar(10),
  DepartmentId int
)
```

SQL Script to create tblDepartment table
```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

Insert data into tblDepartment table
```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

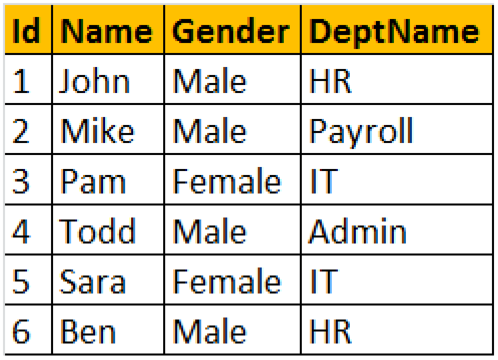
Insert data into tblEmployee table
```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)
Insert into tblEmployee values (4,'Todd', 'Male', 4)
Insert into tblEmployee values (5,'Sara', 'Female', 1)
Insert into tblEmployee values (6,'Ben', 'Male', 3)
```

Since, we now have the required tables, let's create a view based on these tables. The view should return Employee Id, Name, Gender and DepartmentName columns. So, the view is obviously based on multiple tables.

Script to create the view:

```
Create view vWEmployeeDetails
as
Select Id, Name, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

When you execute, **Select * from vWEmployeeDetails**, the data from the view, should be as shown below



Id	Name	Gender	DeptName
1	John	Male	HR
2	Mike	Male	Payroll
3	Pam	Female	IT
4	Todd	Male	Admin
5	Sara	Female	IT
6	Ben	Male	HR

Now, let's try to insert a row into the view, vWEmployeeDetails, by executing the following query. At this point, an error will be raised stating 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables.'

```
Insert into vWEmployeeDetails values(7, 'Valarie', 'Female', 'IT')
```

So, inserting a row into a view that is based on multipe tables, raises an error by default. Now, let's understand, how INSTEAD OF TRIGGERS can help us in this situation. Since, we are getting an error, when we are trying to insert a row into the view, let's create an INSTEAD OF INSERT trigger on the view **vWEmployeeDetails.**

Script to create INSTEAD OF INSERT trigger:

```
Create trigger tr_vWEmployeeDetails_InsteadOfInsert
on vWEmployeeDetails
Instead Of Insert
as
Begin
 Declare @DeptId int

 --Check if there is a valid DepartmentId
 --for the given DepartmentName
 Select @DeptId = DeptId
 from tblDepartment
 join inserted
 on inserted.DeptName = tblDepartment.DeptName

 --If DepartmentId is null throw an error
 --and stop processing
 if(@DeptId is null)
 Begin
  Raiserror('Invalid Department Name. Statement terminated', 16, 1)
  return
 End

 --Finally insert into tblEmployee table
 Insert into tblEmployee(Id, Name, Gender, DepartmentId)
 Select Id, Name, Gender, @DeptId
 from inserted
End
```

Now, let's execute the insert query:

```
Insert into vWEmployeeDetails values(7, 'Valarie', 'Female', 'IT')
```

The instead of trigger correctly inserts, the record into tblEmployee table. Since, we are inserting a row, the **inserted** table, contains the newly added row, where as the **deleted** table will be empty.

In the trigger, we used Raiserror() function, to raise a custom error, when the DepartmentName provided in the insert query, doesnot exist. We are passing 3 parameters to the Raiserror() method. The first parameter is the error message, the second parameter is the severity level. Severity level 16, indicates general errors that can be corrected by the user. The final parameter is the state. We will talk about Raiserror() and exception handling in sql server, in a later video session.

Instead of update triggers - Part 46

Suggested SQL Server Videos before watching this Video

Part 43 - DML triggers
Part 44 - DML After Update Trigger
Part 45 - Instead of Insert Trigger

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

Let's create the required Employee and Department tables, that we will be using for this demo.

SQL Script to create tblEmployee table:

```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Gender nvarchar(10),
  DepartmentId int
)
```

SQL Script to create tblDepartment table

```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

Insert data into tblDepartment table

```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

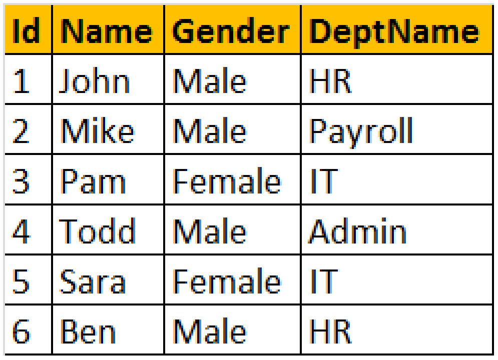
Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)
Insert into tblEmployee values (4,'Todd', 'Male', 4)
Insert into tblEmployee values (5,'Sara', 'Female', 1)
Insert into tblEmployee values (6,'Ben', 'Male', 3)
```

Since, we now have the required tables, let's create a view based on these tables. The view should return Employee Id, Name, Gender and DepartmentName columns. So, the view is obviously based on multiple tables.
Script to create the view:

```
Create view vWEmployeeDetails
as
Select Id, Name, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

When you execute, Select * from vWEmployeeDetails, the data from the view, should be as shown below



Id	Name	Gender	DeptName
1	John	Male	HR
2	Mike	Male	Payroll
3	Pam	Female	IT
4	Todd	Male	Admin
5	Sara	Female	IT
6	Ben	Male	HR

In Part 45, we tried to insert a row into the view, and we got an error stating - 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables.'

Now, let's try to update the view, in such a way that, it affects, both the underlying tables, and see, if we get the same error. The following UPDATE statement changes **Name**

column from **tblEmployee** and **DeptName column** from **tblDepartment**. So, when we execute this query, we get the same error.

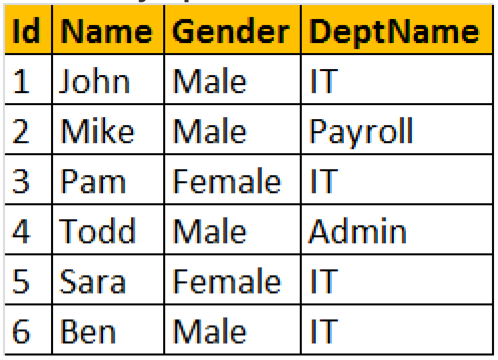
```
Update vWEmployeeDetails
set Name = 'Johny', DeptName = 'IT'
where Id = 1
```

Now, let's try to change, just the department of John from HR to IT. The following UPDATE query, affects only one table, tblDepartment. So, the query should succeed. But, before executing the query, please note that, employees **JOHN** and **BEN** are in **HR** department.

```
Update vWEmployeeDetails
set DeptName = 'IT'
where Id = 1
```

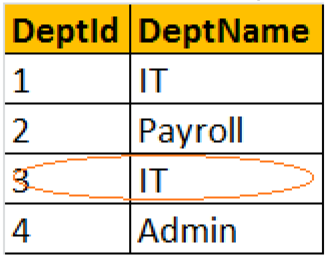
After executing the query, select the data from the view, and notice that **BEN's DeptName** is also changed to **IT**. We intended to just change **JOHN's DeptName**. So, the UPDATE didn't work as expected. This is because, the UPDATE query, updated the **DeptName from HR to IT**, in tblDepartment table. For the UPDATE to work correctly, we should change the **DeptId** of **JOHN** from 3 to 1.

Incorrectly Updated View



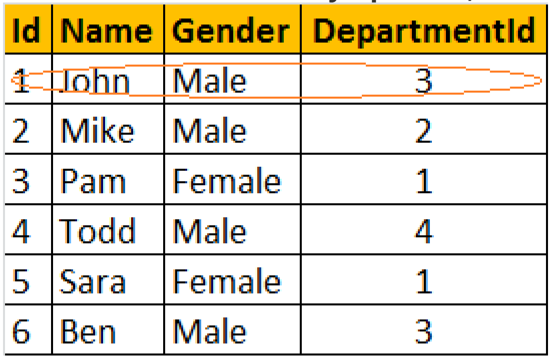
Id	Name	Gender	DeptName
1	John	Male	IT
2	Mike	Male	Payroll
3	Pam	Female	IT
4	Todd	Male	Admin
5	Sara	Female	IT
6	Ben	Male	IT

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



DeptId	DeptName
1	IT
2	Payroll
3	IT
4	Admin

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



Id	Name	Gender	DepartmentId
1	John	Male	3
2	Mike	Male	2
3	Pam	Female	1
4	Todd	Male	4
5	Sara	Female	1
6	Ben	Male	3

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

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

```
Update tblDepartment set DeptName = 'HR' where DeptId = 3
```

Script to create INSTEAD OF UPDATE trigger:

```
Create Trigger tr_vWEmployeeDetails_InsteadOfUpdate
on vWEmployeeDetails
instead of update
as
Begin
 -- if EmployeeId is updated
 if(Update(Id))
 Begin
  Raiserror('Id cannot be changed', 16, 1)
  Return
 End

 -- If DeptName is updated
 if(Update(DeptName))
 Begin
  Declare @DeptId int

  Select @DeptId = DeptId
  from tblDepartment
  join inserted
  on inserted.DeptName = tblDepartment.DeptName

  if(@DeptId is NULL )
  Begin
   Raiserror('Invalid Department Name', 16, 1)
   Return
  End

  Update tblEmployee set DepartmentId = @DeptId
  from inserted
  join tblEmployee
  on tblEmployee.Id = inserted.id
 End

 -- If gender is updated
 if(Update(Gender))
 Begin
  Update tblEmployee set Gender = inserted.Gender
  from inserted
  join tblEmployee
  on tblEmployee.Id = inserted.id
 End
 -- If Name is updated
 if(Update(Name))
 Begin
  Update tblEmployee set Name = inserted.Name
  from inserted
  join tblEmployee
  on tblEmployee.Id = inserted.id
 End
End
```

Now, let's try to update JOHN's Department to IT.

```
Update vWEmployeeDetails
set DeptName = 'IT'
where Id = 1
```

The UPDATE query works as expected. The INSTEAD OF UPDATE trigger, correctly updates, JOHN's DepartmentId to 1, in tblEmployee table.

Now, let's try to update Name, Gender and DeptName. The UPDATE query, works as expected, without raising the error - 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables.'

```
Update vWEmployeeDetails
set Name = 'Johny', Gender = 'Female', DeptName = 'IT'
where Id = 1
```

Update() function used in the trigger, returns true, even if you update with the same value. For this reason, I recomend to compare values between inserted and deleted tables, rather than relying on Update() function. The Update() function does not operate on a per row basis, but across all rows.

Instead of delete trigger - Part 47

Suggested SQL Server Videos before watching this Video

Part 45 - Instead of Insert Trigger
Part 46 - Instead of Update Trigger

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

Let's create the required Employee and Department tables, that we will be using for this demo.

SQL Script to create tblEmployee table:

```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Gender nvarchar(10),
  DepartmentId int
)
```

SQL Script to create tblDepartment table

```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

Insert data into tblDepartment table

```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

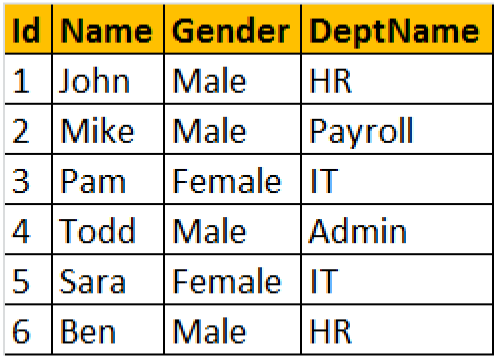
Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)
Insert into tblEmployee values (4,'Todd', 'Male', 4)
Insert into tblEmployee values (5,'Sara', 'Female', 1)
Insert into tblEmployee values (6,'Ben', 'Male', 3)
```

Since, we now have the required tables, let's create a view based on these tables. The view should return Employee Id, Name, Gender and DepartmentName columns. So, the view is obviously based on multiple tables.
Script to create the view:

```
Create view vWEmployeeDetails
as
Select Id, Name, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

When you execute, Select * from vWEmployeeDetails, the data from the view, should be as shown below



Id	Name	Gender	DeptName
1	John	Male	HR
2	Mike	Male	Payroll
3	Pam	Female	IT
4	Todd	Male	Admin
5	Sara	Female	IT
6	Ben	Male	HR

In Part 45, we tried to insert a row into the view, and we got an error stating - 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables'. Along, the same lines, in **Part 46**, when we tried to update a view that is based on multiple tables, we got the same error. To get the error, the UPDATE should affect both the base tables. If the update affects only one base table, we don't get the error, but the UPDATE does not work correctly, if the **DeptName** column is updated.

Now, let's try to delete a row from the view, and we get the same error.

```
Delete from vWEmployeeDetails where Id = 1
```

Script to create INSTEAD OF DELETE trigger:

```
Create Trigger tr_vWEmployeeDetails_InsteadOfDelete
on vWEmployeeDetails
instead of delete
as
Begin
 Delete tblEmployee
 from tblEmployee
 join deleted
 on tblEmployee.Id = deleted.Id

 --Subquery
 --Delete from tblEmployee
 --where Id in (Select Id from deleted)
End
```

Notice that, the trigger tr_vWEmployeeDetails_InsteadOfDelete, makes use of DELETED table. DELETED table contains all the rows, that we tried to DELETE from the view. So, we are joining the DELETED table with tblEmployee, to delete the rows. You can also use sub-queries to do the same. In most cases JOINs are faster than SUB-QUERIEs. However, in cases, where you only need a subset of records from a table that you are joining with, sub-queries can be faster.

Upon executing the following DELETE statement, the row gets DELETED as expected from tblEmployee table

```
Delete from vWEmployeeDetails where Id = 1
```

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

Derived table and CTE in sql server - Part 48

In this video we will learn about, Derived tables and common table expressions (CTE's). We will also explore the differences between Views, Table Variable, Local and Global Temp Tables, Derived tables and common table expressions.

Let's create the required Employee and Department tables, that we will be using for this demo.

SQL Script to create tblEmployee table:

```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Gender nvarchar(10),
  DepartmentId int
)
```

SQL Script to create tblDepartment table

```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

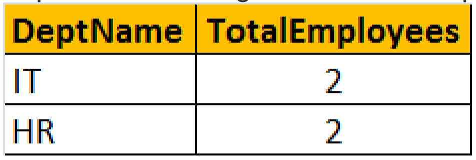
Insert data into tblDepartment table

```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)
Insert into tblEmployee values (4,'Todd', 'Male', 4)
Insert into tblEmployee values (5,'Sara', 'Female', 1)
Insert into tblEmployee values (6,'Ben', 'Male', 3)
```

Now, we want to write a query which would return the following output. The query should return, the Department Name and Total Number of employees, with in the department. The departments with greatar than or equal to 2 employee should only be returned.



DeptName	TotalEmployees
IT	2
HR	2

Obviously, there are severl ways to do this. Let's see how to achieve this, with the help of a view

Script to create the View

```
Create view vWEmployeeCount
as
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId
```

Query using the view:

```
Select DeptName, TotalEmployees
from vWEmployeeCount
where  TotalEmployees >= 2
```

Note: Views get saved in the database, and can be available to other queries and stored procedures. However, if this view is only used at this one place, it can be easily eliminated using other options, like CTE, Derived Tables, Temp Tables, Table Variable etc.

Now, let's see, how to achieve the same using, temporary tables. We are using local temporary tables here.

```
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
into #TempEmployeeCount
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId

Select DeptName, TotalEmployees
From #TempEmployeeCount
where TotalEmployees >= 2

Drop Table #TempEmployeeCount
```

Note: Temporary tables are stored in TempDB. Local temporary tables are visible only in the current session, and can be shared between nested stored procedure calls. Global temporary tables are visible to other sessions and are destroyed, when the last connection referencing the table is closed.

Using Table Variable:

```
Declare @tblEmployeeCount table
(DeptName nvarchar(20),DepartmentId int, TotalEmployees int)

Insert @tblEmployeeCount
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId

Select DeptName, TotalEmployees
From @tblEmployeeCount
where  TotalEmployees >= 2
```

Note: Just like TempTables, a table variable is also created in TempDB. The scope of a table variable is the batch, stored procedure, or statement block in which it is declared. They can be passed as parameters between procedures.

Using Derived Tables

```
Select DeptName, TotalEmployees
from
 (
  Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
  from tblEmployee
  join tblDepartment
  on tblEmployee.DepartmentId = tblDepartment.DeptId
  group by DeptName, DepartmentId
 )
as EmployeeCount
where TotalEmployees >= 2
```

Note: Derived tables are available only in the context of the current query.

Using CTE
```
With EmployeeCount(DeptName, DepartmentId, TotalEmployees)
as
(
 Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
 from tblEmployee
 join tblDepartment
 on tblEmployee.DepartmentId = tblDepartment.DeptId
 group by DeptName, DepartmentId
)

Select DeptName, TotalEmployees
from EmployeeCount
where TotalEmployees >= 2
```

Note: A CTE can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is similar to a derived table in that it is not stored as an object and lasts only for the duration of the query.

Common Table Expressions - Part 49

Common table expression (CTE) is introduced in SQL server 2005. A **CTE** is a temporary result set, that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement, that immediately follows the **CTE.**

Let's create the required Employee and Department tables, that we will be using for this demo.

SQL Script to create tblEmployee table:
```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Gender nvarchar(10),
  DepartmentId int
)
```

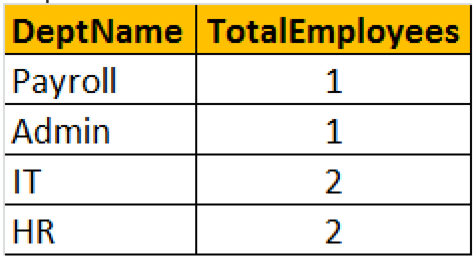
SQL Script to create tblDepartment table
```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

Insert data into tblDepartment table
```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)
Insert into tblEmployee values (4,'Todd', 'Male', 4)
Insert into tblEmployee values (5,'Sara', 'Female', 1)
Insert into tblEmployee values (6,'Ben', 'Male', 3)
```

Write a query using CTE, to display the total number of Employees by Department Name. The output should be as shown below.



DeptName	TotalEmployees
Payroll	1
Admin	1
IT	2
HR	2

Before we write the query, let's look at the syntax for creating a CTE.

```
WITH cte_name (Column1, Column2, ..)
AS
( CTE_query )
```

SQL query using CTE:

```
With EmployeeCount(DepartmentId, TotalEmployees)
as
(
 Select DepartmentId, COUNT(*) as TotalEmployees
 from tblEmployee
 group by DepartmentId
)

Select DeptName, TotalEmployees
from tblDepartment
join EmployeeCount
on tblDepartment.DeptId = EmployeeCount.DepartmentId
order by TotalEmployees
```

We define a CTE, using **WITH** keyword, followed by the name of the CTE. In our example, **EmployeeCount** is the name of the CTE. Within parentheses, we specify the columns that make up the CTE. **DepartmentId** and **TotalEmployees** are the columns of **EmployeeCount** CTE. These 2 columns map to the columns returned by the **SELECT CTE query**. The CTE column names and CTE query column names can be different. Infact, CTE column names are optional. However, if you do specify, the number of **CTE columns** and the **CTE SELECT query** columns should be same. Otherwise you will get an error stating - 'EmployeeCount has fewer columns than were specified in the column list'. The column list, is followed by the as keyword, following which we have the CTE query within a pair of parentheses.

EmployeeCount CTE is being joined with **tblDepartment** table, in the SELECT query, that immediately follows the CTE. Remember, a CTE can only be referenced by a SELECT, INSERT, UPDATE, or DELETE statement, **that immediately follows the CTE**. If you try to do something else in between, we get an error stating - 'Common table expression defined but not used'. The following SQL, raise an error.

```
With EmployeeCount(DepartmentId, TotalEmployees)
as
(
 Select DepartmentId, COUNT(*) as TotalEmployees
 from tblEmployee
 group by DepartmentId
)

Select 'Hello'

Select DeptName, TotalEmployees
from tblDepartment
join EmployeeCount
on tblDepartment.DeptId = EmployeeCount.DepartmentId
order by TotalEmployees
```

It is also, possible to create multiple CTE's using a single WITH clause.

```
With EmployeesCountBy_Payroll_IT_Dept(DepartmentName, Total)
as
(
 Select DeptName, COUNT(Id) as TotalEmployees
 from tblEmployee
 join tblDepartment
 on tblEmployee.DepartmentId = tblDepartment.DeptId
 where DeptName IN ('Payroll','IT')
 group by DeptName
),
EmployeesCountBy_HR_Admin_Dept(DepartmentName, Total)
as
(
 Select DeptName, COUNT(Id) as TotalEmployees
 from tblEmployee
 join tblDepartment
 on tblEmployee.DepartmentId = tblDepartment.DeptId
 group by DeptName
)
Select * from EmployeesCountBy_HR_Admin_Dept
UNION
Select * from EmployeesCountBy_Payroll_IT_Dept
```

Updatable CTE - Part 50

Is it possible to UPDATE a CTE?

Yes & No, depending on the number of base tables, the CTE is created upon, and the number of base tables affected by the UPDATE statement. If this is not clear at the moment, don't worry. We will try to understand this with an example.

Let's create the required tblEmployee and tblDepartment tables, that we will be using for this demo.

SQL Script to create tblEmployee table:

```
CREATE TABLE tblEmployee
```

```
(
  Id int Primary Key,
  Name nvarchar(30),
  Gender nvarchar(10),
  DepartmentId int
)
```

SQL Script to create tblDepartment table

```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

Insert data into tblDepartment table

```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)
Insert into tblEmployee values (4,'Todd', 'Male', 4)
Insert into tblEmployee values (5,'Sara', 'Female', 1)
Insert into tblEmployee values (6,'Ben', 'Male', 3)
```

Let's create a simple common table expression, based on tblEmployee table. **Employees_Name_Gender** CTE is getting all the required columns from one base table tblEmployee.

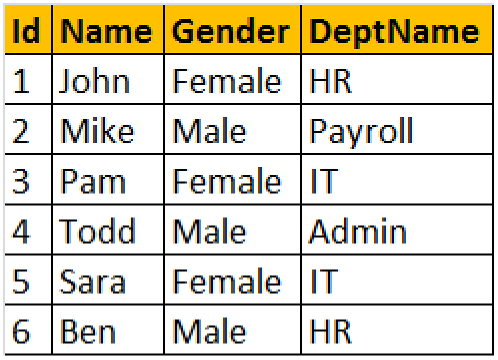
```
With Employees_Name_Gender
as
(
 Select Id, Name, Gender from tblEmployee
)
Select * from Employees_Name_Gender
```

Let's now, UPDATE JOHN's gender from Male to Female, using the **Employees_Name_Gender CTE**

```
With Employees_Name_Gender
as
(
 Select Id, Name, Gender from tblEmployee
)
Update Employees_Name_Gender Set Gender = 'Female' where Id = 1
```

Now, query the tblEmployee table. JOHN's gender is actually UPDATED. So, if a CTE is created on one base table, then it is possible to UPDATE the CTE, which in turn will update the underlying base table. In this case, UPDATING **Employees_Name_Gender** CTE, updates **tblEmployee** table.

Now, let's create a CTE, on both the tables - tblEmployee and tblDepartment. The CTE should return, Employee Id, Name, Gender and Department. In short the output should be as shown below.



Id	Name	Gender	DeptName
1	John	Female	HR
2	Mike	Male	Payroll
3	Pam	Female	IT
4	Todd	Male	Admin
5	Sara	Female	IT
6	Ben	Male	HR

CTE, that returns Employees by Department

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Select * from EmployeesByDepartment
```

Let's update this CTE. Let's change JOHN's Gender from **Female to Male**. Here, the CTE is based on 2 tables, but the UPDATE statement affects only one base table **tblEmployee**. So the UPDATE succeeds. So, if a CTE is based on more than one table, and if the UPDATE affects only one base table, then the UPDATE is allowed.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set Gender = 'Male' where Id = 1
```

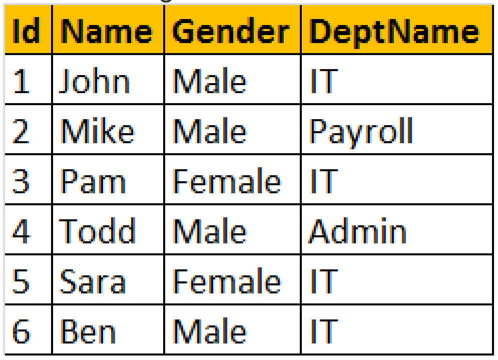
Now, let's try to **UPDATE the CTE**, in such a way, that the update affects both the tables - **tblEmployee and tblDepartment**. This UPDATE statement changes **Gender** from **tblEmployee** table and **DeptName** from **tblDepartment** table. When you execute this UPDATE, you get an error stating - 'View or function EmployeesByDepartment is not updatable because the modification affects multiple base tables'. So, if a CTE is based on multiple tables, and if the UPDATE statement affects more than 1 base table, then the UPDATE is not allowed.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set
Gender = 'Female', DeptName = 'IT'
where Id = 1
```

Finally, let's try to UPDATE just the DeptName. Let's change JOHN's DeptName from HR to IT. Before, you execute the UPDATE statement, notice that BEN is also currently in HR department.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set
DeptName = 'IT' where Id = 1
```

After you execute the UPDATE. Select data from the CTE, and you will see that BEN's DeptName is also changed to IT.



Id	Name	Gender	DeptName
1	John	Male	IT
2	Mike	Male	Payroll
3	Pam	Female	IT
4	Todd	Male	Admin
5	Sara	Female	IT
6	Ben	Male	IT

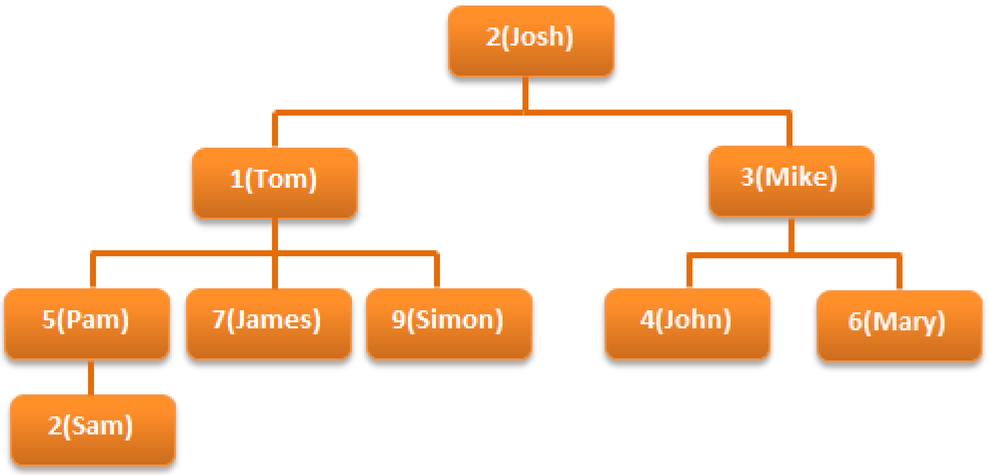
This is because, when we updated the **CTE**, the UPDATE has actually changed the **DeptName** from **HR** to **IT**, in **tblDepartment** table, instead of changing the **DepartmentId** column (from 3 to 1) in **tblEmployee** table. So, if a CTE is based on multiple tables, and if the UPDATE statement affects only one base table, the update succeeds. But the update may not work as you expect.

So in short if,
1. A CTE is based on a single base table, then the UPDATE suceeds and works as expected.
2. A CTE is based on more than one base table, and if the UPDATE affects multiple base tables, the update is not allowed and the statement terminates with an error.
3. A CTE is based on more than one base table, and if the UPDATE affects only one base table, the UPDATE succeeds(but not as expected always)

Recursive CTE - Part 51

Suggested SQL Server videos
Part 49 - Common Table Expressions (CTE)
Part 50 - Updatable CTE

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.

Database Normalization - Part 52

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**.

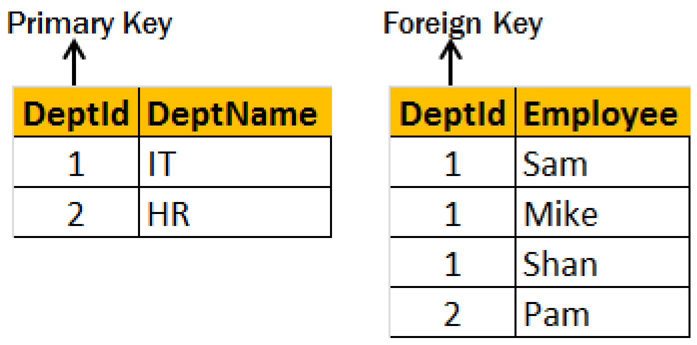
In the table below, data in Employee column is not atomic. It contains multiple employees seperated by comma. From the data you can see that in the IT department, we have 3 employees - Sam, Mike, Shan. Now, let's say I want to change just, SHAN name. **It is not possible, we have to update the entire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


Primary Key
Foreign Key

DeptId	DeptName
1	IT
2	HR

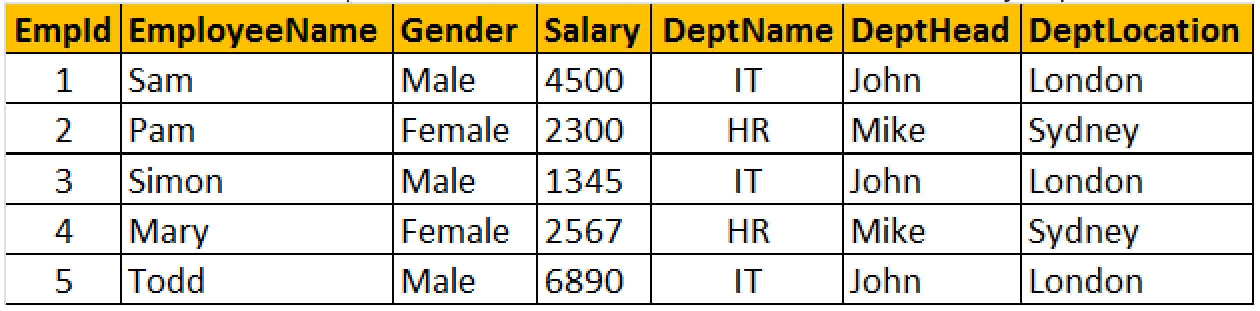
DeptId	Employee
1	Sam
1	Mike
1	Shan
2	Pam

Second Normal Form and Third Normal Form - Part 53

Suggested sql server videos
Part 52 - Database Normalization & First Normal Form

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.

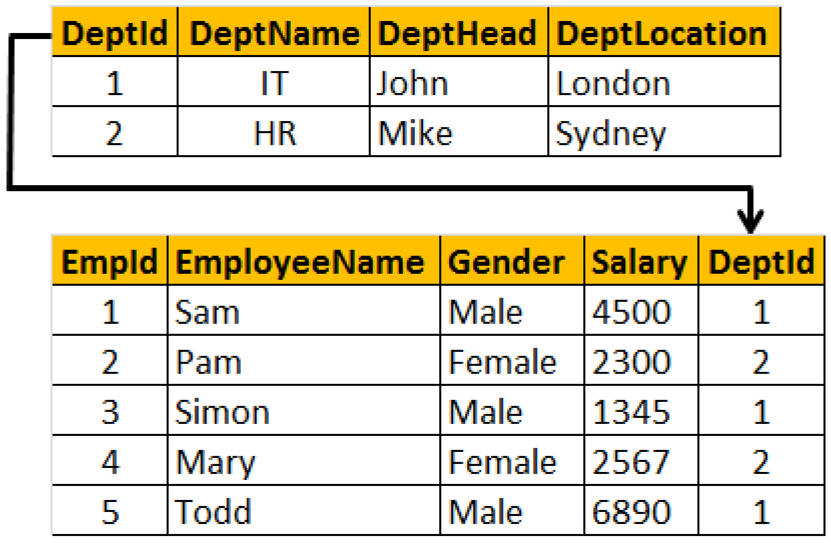


EmpId	EmployeeName	Gender	Salary	DeptName	DeptHead	DeptLocation
1	Sam	Male	4500	IT	John	London
2	Pam	Female	2300	HR	Mike	Sydney
3	Simon	Male	1345	IT	John	London
4	Mary	Female	2567	HR	Mike	Sydney
5	Todd	Male	6890	IT	John	London

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.

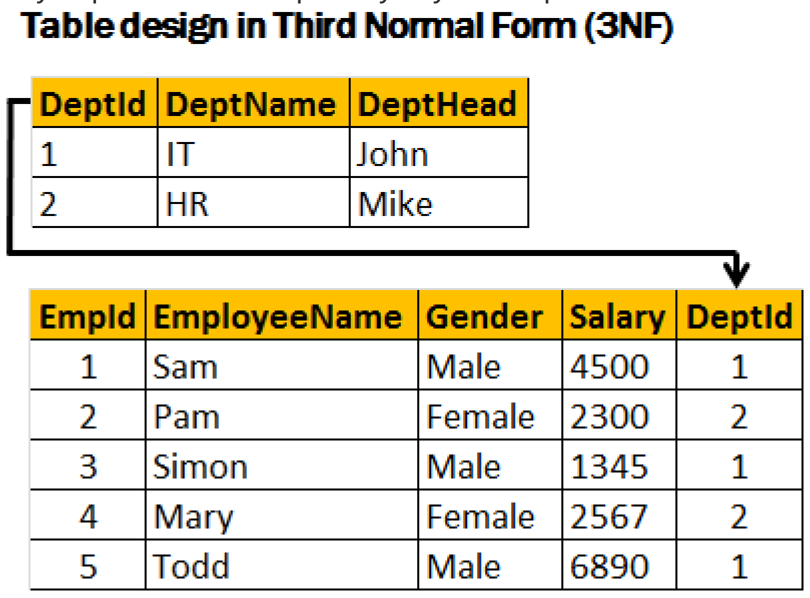


Table design in Third Normal Form (3NF)

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

Suggested SQL Server Videos:
Part 11 - Group By
Part 48 - Derived table and CTE in sql server

One of my youtube channel subscribers, has asked me to make a video on **PIVOT** operator. So here we are with another sql server video.

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

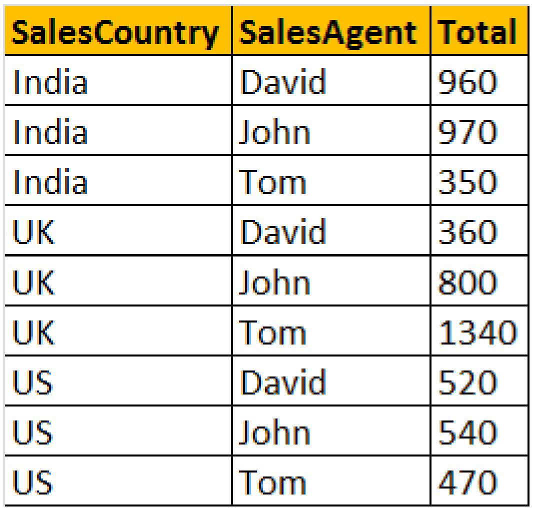
```
Create Table tblProductSales
(
 SalesAgent nvarchar(50),
 SalesCountry nvarchar(50),
 SalesAmount int
)

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)
```

```sql
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.

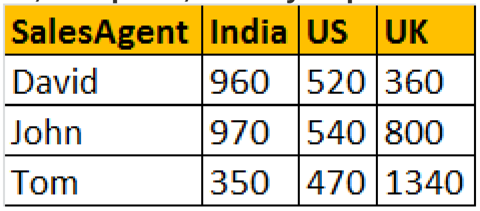


SalesCountry	SalesAgent	Total
India	David	960
India	John	970
India	Tom	350
UK	David	360
UK	John	800
UK	Tom	1340
US	David	520
US	John	540
US	Tom	470

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.

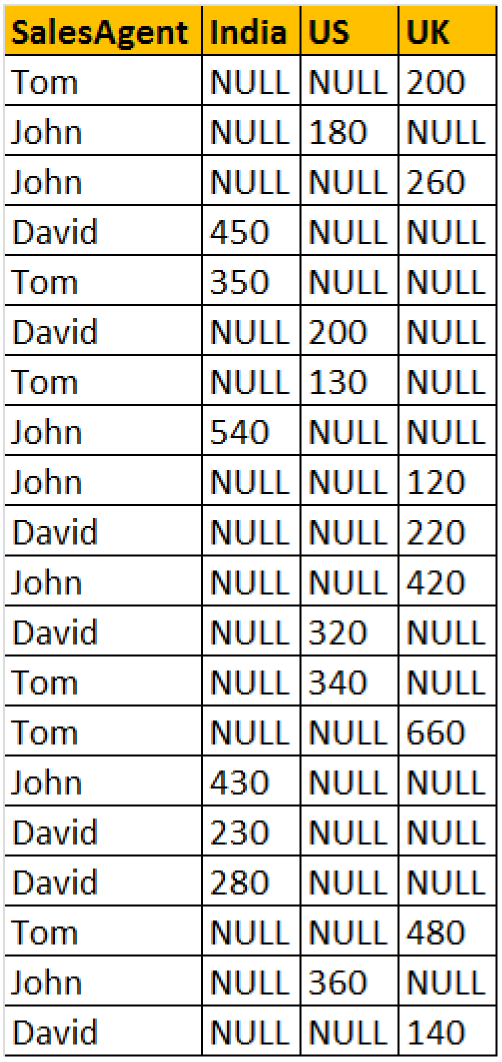
```
Create Table tblProductsSale
(
   Id int primary key,
   SalesAgent nvarchar(50),
   SalesCountry nvarchar(50),
   SalesAmount int
)
```

```
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>;
```

Error handling in sql server 2000 - Part 55

Suggested SQL Server videos
Part 18 - Stored procedures

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

Now let's create **tblProduct** and **tblProductSales**, that we will be using for the rest of this demo.

SQL script to create tblProduct
```
Create Table tblProduct
(
 ProductId int NOT NULL primary key,
 Name nvarchar(50),
 UnitPrice int,
 QtyAvailable int
)
```

SQL script to load data into tblProduct
```
Insert into tblProduct values(1, 'Laptops', 2340, 100)
Insert into tblProduct values(2, 'Desktops', 3467, 50)
```

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

```
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 Tran
         -- First reduce the quantity available
  Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)
  where ProductId = @ProductId

  Declare @MaxProductSalesId int
  -- Calculate MAX ProductSalesId
  Select @MaxProductSalesId = Case When
          MAX(ProductSalesId) IS NULL
          Then 0 else MAX(ProductSalesId) end
         from tblProductSales
  -- Increment @MaxProductSalesId by 1, so we don't get a primary key violation
  Set @MaxProductSalesId = @MaxProductSalesId + 1
  Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)
    Commit Tran
   End
End
```

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

2. Sections of the stored procedure is commented, and is self explanatory.

3. In the procedure, we are using **Raiserror**() function to return an error message back to the calling application, if the stock available is less than the quantity we are trying to sell. We have to pass atleast 3 parameters to the Raiserror() function.
RAISERROR('Error Message', ErrorSeverity, ErrorState)
Severity and State are integers. In most cases, when you are returning custom errors, the severity level is 16, which indicates general errors that can be corrected by the user. In this case, the error can be corrected, by adjusting the **@QuantityToSell**, to be less than or equal to the stock available. ErrorState is also an integer between 1 and 255. RAISERROR only generates errors with state from 1 through 127.

4. The problem with this procedure is that, the **transaction is always committed**. Even, if there is an error somewhere, between updating **tblProduct** and **tblProductSales** table. In fact, the main purpose of wrapping these 2 statments (Update tblProduct Statement & Insert into tblProductSales statement) in a transaction is to ensure that, both of the statements are treated as a single unit. For example, if we have an error when executing the second statement, then the first statement should also be rolledback.

In SQL server 2000, to detect errors, we can use @@**Error** system function. @@Error returns a NON-ZERO value, if there is an error, otherwise ZERO, indicating that the previous sql statement encountered no errors. The stored procedure **spSellProductCorrected**, makes use of @@ERROR system function to detect any errors that may have occurred. If there are errors, roll back the transaction, else commit the transaction. If you comment the line (Set @MaxProductSalesId = @MaxProductSalesId + 1), and then execute the stored procedure there will be a primary key violation error, when trying to insert into **tblProductSales**. As a result of this the entire transaction will be rolled back.

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

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

  Declare @MaxProductSalesId int
  -- Calculate MAX ProductSalesId
  Select @MaxProductSalesId = Case When
          MAX(ProductSalesId) IS NULL
          Then 0 else MAX(ProductSalesId) end
         from tblProductSales
  -- Increment @MaxProductSalesId by 1, so we don't get a primary key violation
  Set @MaxProductSalesId = @MaxProductSalesId + 1
  Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)
  if(@@ERROR <> 0)
  Begin
   Rollback Tran
   Print 'Rolled Back Transaction'
  End
  Else
  Begin
   Commit Tran
   Print 'Committed Transaction'
  End
   End
End
```

Note: @@ERROR is cleared and reset on each statement execution. Check it immediately following the statement being verified, or save it to a local variable that can be checked later.

In **tblProduct** table, we already have a record with **ProductId = 2**. So the insert statement causes a primary key violation error. @@ERROR retains the error number, as we are checking for it immediately after the statement that cause the error.

```
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
if(@@ERROR <> 0)
 Print 'Error Occurred'
Else
 Print 'No Errors'
```

On the other hand, when you execute the code below, you get message **'No Errors'** printed. This is because the @@ERROR is cleared and reset on each statement execution.

```
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
--At this point @@ERROR will have a NON ZERO value
Select * from tblProduct
--At this point @@ERROR gets reset to ZERO, because the
--select statement successfullyexecuted
if(@@ERROR <> 0)
 Print 'Error Occurred'
Else
 Print 'No Errors'
```

In this example, we are storing the value of @@Error function to a local variable, which is then used later.

```
Declare @Error int
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
Set @Error = @@ERROR
Select * from tblProduct
if(@Error <> 0)
 Print 'Error Occurred'
Else
 Print 'No Errors'
```

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

Suggested SQL Server Videos
Part 18 - Stored Procedures
Part 55 - Error handling in SQL Server 2000

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
```

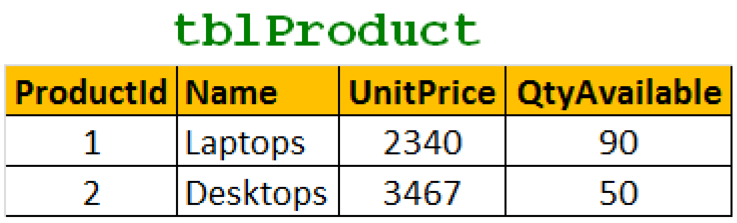
Now, if we execute spUpdateAddress, the first **UPDATE** statements succeeds, but the second **UPDATE** statement fails. As, soon as the second UPDATE statement fails, the control is immediately transferred to the CATCH block. The CATCH block rolls the transaction back. So, the change made by the first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

Transaction Acid Tests - Part 58

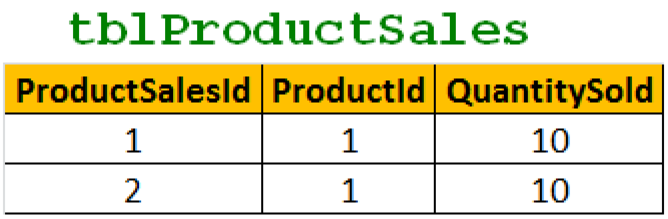
Suggested SQL Server Videos
Part 57 - Transactions in SQL Server

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

Suggested Videos
Part 56 - Error handling in sql server 2005, 2008
Part 57 - Transactions
Part 58 - Transaction Acid Tests

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.

Next Video:
What to choose for performance? Queries that involve a subquery or a join

Correlated subquery in sql - Part 60

Suggested Videos
Part 57 - Transactions
Part 58 - Transaction Acid Tests
Part 59 - Subqueries

In this video we will discuss about Corelated Subqueries

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

Suggested Videos
Part 58 - Transaction Acid Tests
Part 59 - Subqueries
Part 60 - Correlated subquery

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
```

In our next video, we will be using these tables, for performance testing of queries that uses subqueries and joins.

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

Suggested Videos
Part 59 - Subqueries
Part 60 - Correlated subquery
Part 61 - Creating a large table with random data for performance testing

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.

In a later video session we will discuss about client statistics and execution plans in detail.

Cursors in sql server - Part 63

Suggested Videos
Part 60 - Correlated subquery
Part 61 - Creating a large table with random data for performance testing
Part 62 - What to choose for performance - SubQuery or Joins

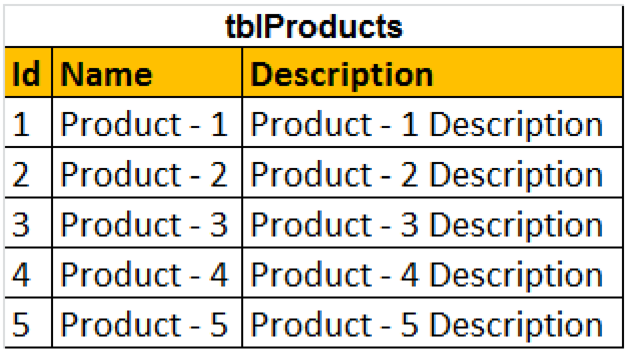
Relational Database Management Systems, including sql server are very good at handling data in SETS. For example, the following "UPDATE" query, updates a set of rows that matches the condition in the "WHERE" clause at the same time.

```
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.

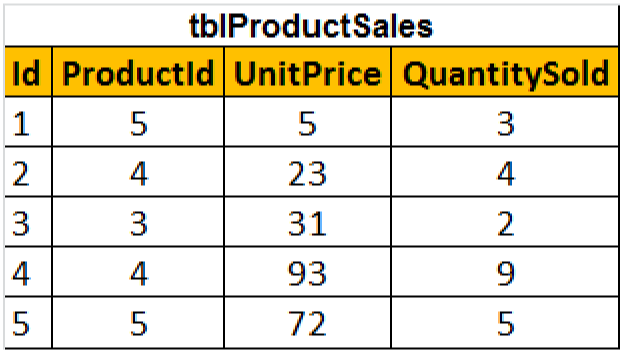
There are different types of cursors in sql server as listed below. We will talk about the differences between these cursor types in a later video session.
1. Forward-Only
2. Static
3. Keyset
4. Dynamic

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

Suggested Videos
Part 61 - Creating a large table with random data for performance testing
Part 62 - What to choose for performance - SubQuery or Joins
Part 63 - Cursors in sql server

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

Suggested Videos
Part 62 - What to choose for performance - SubQuery or Joins
Part 63 - Cursors in sql server
Part 64 - Replacing cursors using joins in sql server

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

Suggested Videos
Part 63 - Cursors in sql server
Part 64 - Replacing cursors using joins in sql server
Part 65 - List all tables in a sql server database using a query

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='EmailAddres
s' 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

Suggested Videos

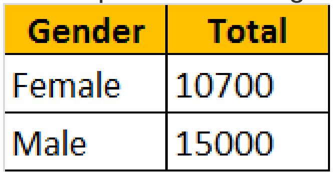
Part 64 - Replacing cursors using joins in sql server
Part 65 - List all tables in a sql server database using a query
Part 66 - Writing re-runnable sql server scripts

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

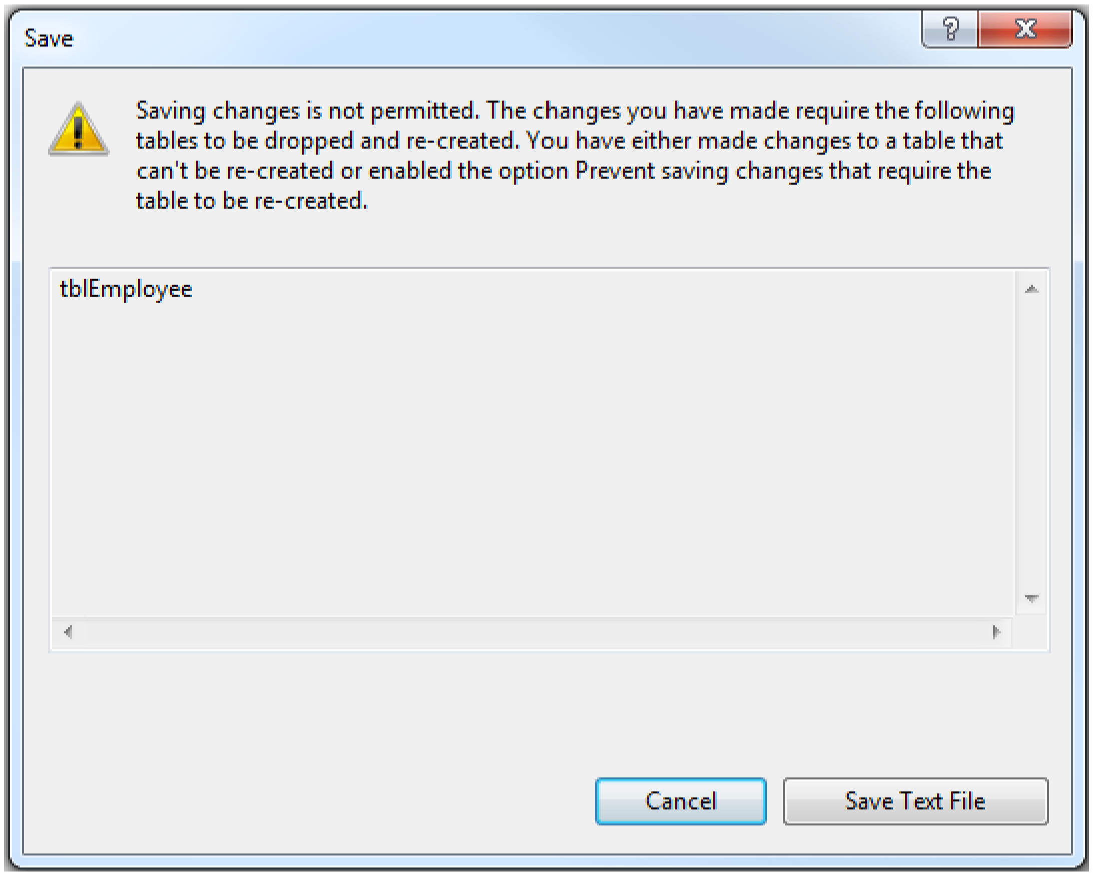
To achieve this we would write a sql query using GROUP BY as shown below.

```
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.


Save

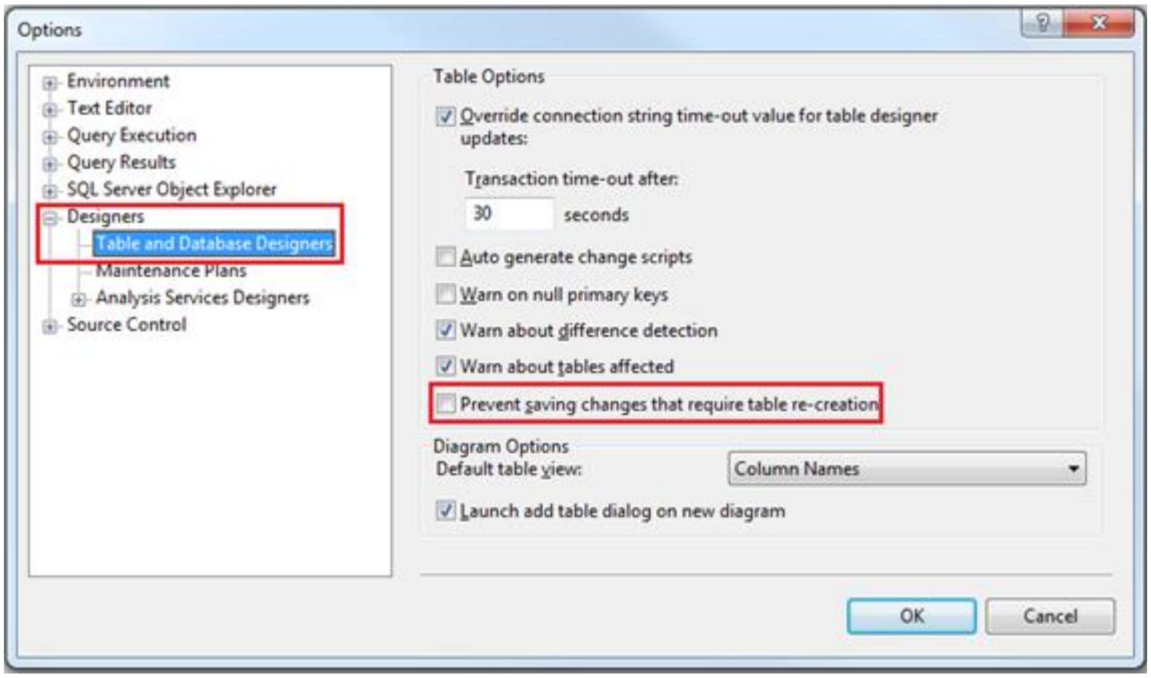
Saving changes is not permitted. The changes you have made require the following tables to be dropped and re-created. You have either made changes to a table that can't be re-created or enabled the option Prevent saving changes that require the table to be re-created.

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

Suggested Videos
Part 65 - List all tables in a sql server database using a query
Part 66 - Writing re-runnable sql server scripts
Part 67 - Alter database table columns without dropping table

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')
```

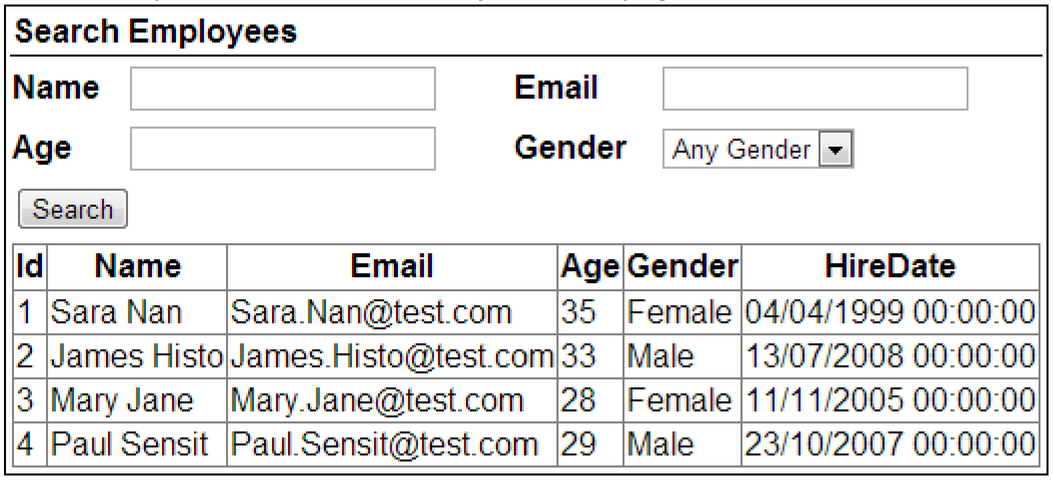
Name, Email, Age and Gender parameters of spSearchEmployees stored procedure are optional. Notice that, we have set defaults for all the parameters, and in the "WHERE" clause we are checking if the respective parameter IS NULL.

```
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.



Search Employees

Name		**Email**	
Age		**Gender**	Any Gender

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

WebForm1.aspx:

```
<table style="font-family:Arial; border:1px solid black">
    <tr>
        <td colspan="4" style="border-bottom: 1px solid black">
            <b>Search Employees</b>
        </td>
    </tr>
    <tr>
        <td>
            <b>Name</b>
        </td>
        <td>
```

```
            <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
        </td>
        <td>
            <b>Email</b>
        </td>
        <td>
            <asp:TextBox ID="txtEmail" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            <b>Age</b>
        </td>
        <td>
            <asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
        </td>
        <td>
            <b>Gender</b>
        </td>
        <td>
            <asp:DropDownList ID="ddlGender" runat="server">
                <asp:ListItem Text="Any Gender" Value="-1"></asp:ListItem>
                <asp:ListItem Text="Male" Value="Male"></asp:ListItem>
                <asp:ListItem Text="Female" Value="Female"></asp:ListItem>
            </asp:DropDownList>
        </td>
    </tr>
    <tr>
        <td colspan="4">
            <asp:Button ID="btnSerach" runat="server" Text="Search"
                onclick="btnSerach_Click" />
        </td>
    </tr>
    <tr>
        <td colspan="4">
            <asp:GridView ID="gvEmployees" runat="server">
            </asp:GridView>
        </td>
    </tr>
</table>
```

WebForm1.aspx.cs:

```
public partial class WebForm1 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            GetData();
        }
    }

    protected void btnSerach_Click(object sender, EventArgs e)
    {
        GetData();
    }

    private void GetData()
    {
        string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
        using (SqlConnection con = new SqlConnection(cs))
        {
            SqlCommand cmd = new SqlCommand("spSearchEmployees", con);
            cmd.CommandType = CommandType.StoredProcedure;

            AttachParameter(cmd, "@Name", txtName);
            AttachParameter(cmd, "@Email", txtEmail);
            AttachParameter(cmd, "@Age", txtAge);
            AttachParameter(cmd, "@Gender", ddlGender);

            con.Open();
            gvEmployees.DataSource = cmd.ExecuteReader();
            gvEmployees.DataBind();
        }
    }

    private void AttachParameter(SqlCommand command, string parameterName, Control control)
    {
        if (control is TextBox && ((TextBox)control).Text != string.Empty)
        {
            SqlParameter parameter = new SqlParameter(parameterName, ((TextBox)control).Text);
            command.Parameters.Add(parameter);
        }
        else if (control is DropDownList && ((DropDownList)control).SelectedValue != "-1")
        {
            SqlParameter parameter = new SqlParameter parameterName,
((DropDownList)control).SelectedValue);
            command.Parameters.Add(parameter);
        }
    }
}
```

Make sure you have the following using statements in your code-behind page

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
```

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)

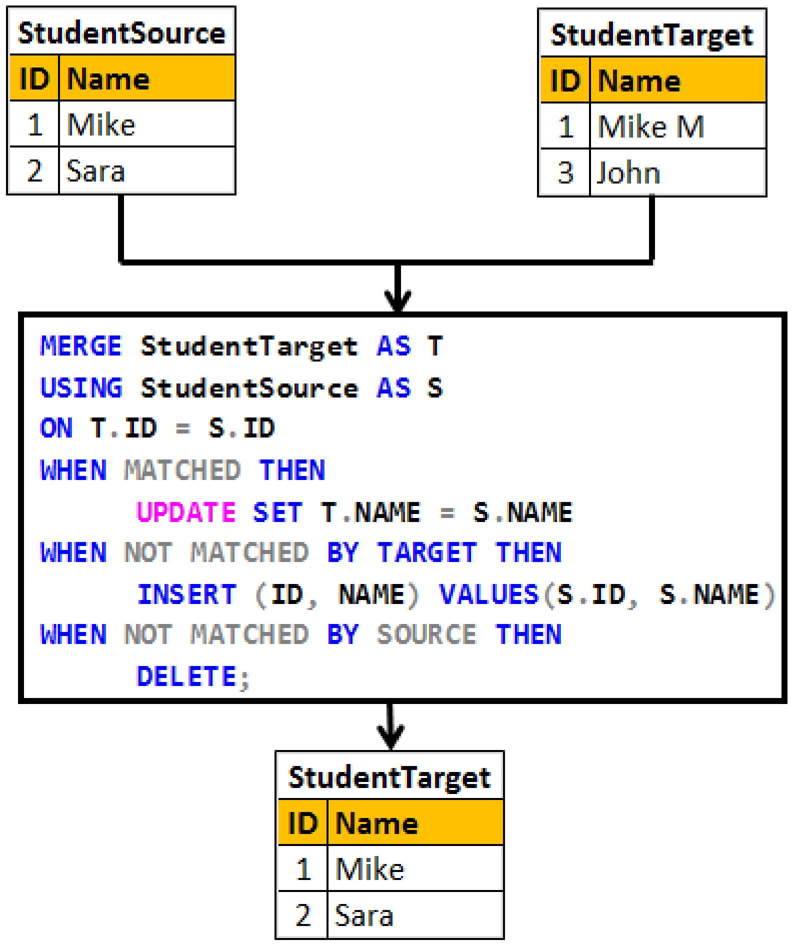
The merge statement joins the target table to the source table by using a common column in both the tables. Based on how the rows match up as a result of the join, we can then perform insert, update, and delete on the target table.

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

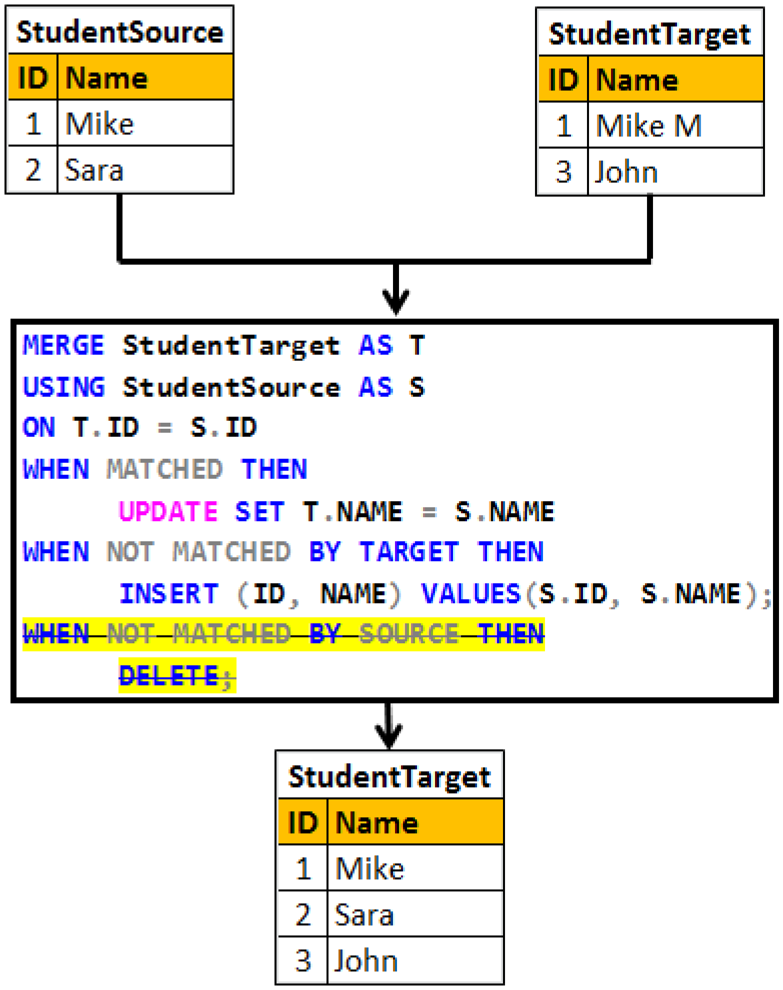
```
Create table StudentSource
(
    ID int primary key,
    Name nvarchar(20)
)
GO

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

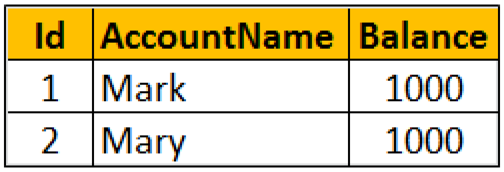
```
Truncate table StudentSource
Truncate table StudentTarget
GO

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

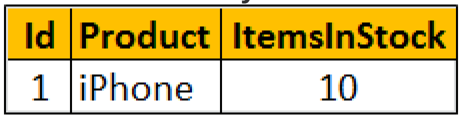
In this video we will discuss, **dirty read concurrency problem** with an example. This is continuation to Part 70. Please watch Part 70 from SQL Server tutorial for beginners.

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

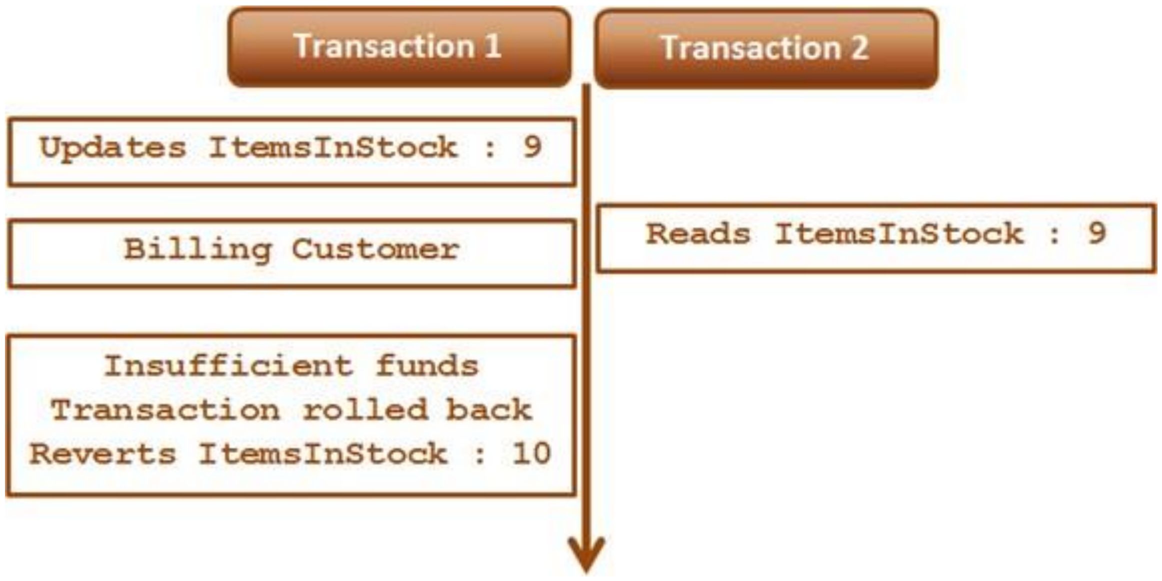
```
Create table tblInventory
(
   Id int identity primary key,
   Product nvarchar(100),
   ItemsInStock int
)
Go

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.

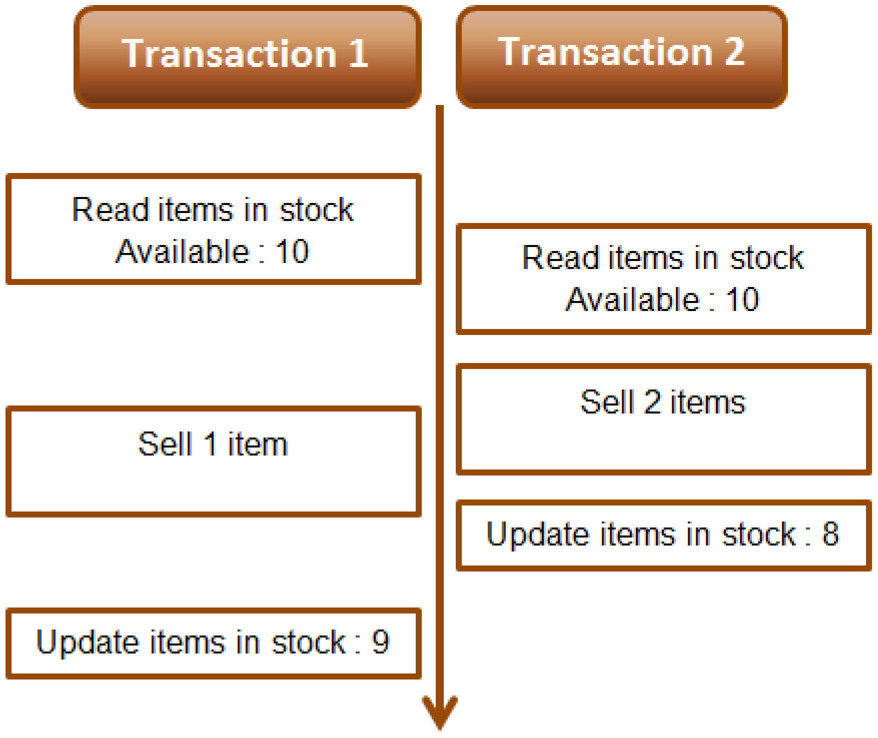
Lost update problem happens when 2 transactions read and update the same data. Let's understand this with an example. We will use the following table **tblInventory** for this example.

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.

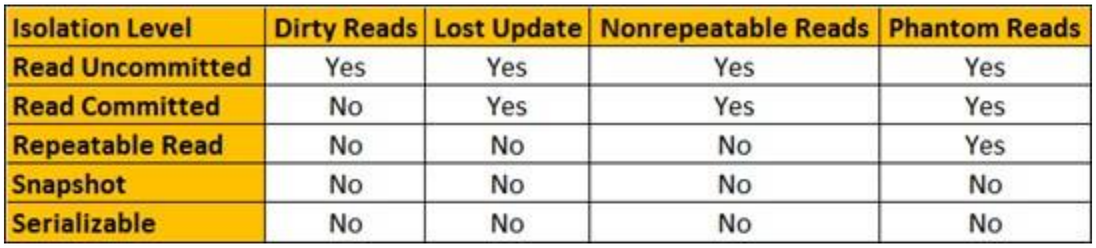

Transaction 1
Transaction 2
Read items in stock Available : 10
Read items in stock Available : 10
Sell 2 items
Sell 1 item
Update items in stock : 8
Update items in stock : 9

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.

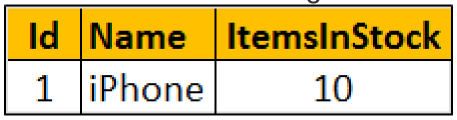
Once you rerun Transaction 2, ItemsInStock will be updated correctly as expected.

Part 73 - Non repeatable read example in sql server

In this video we will discuss **non repeatable read concurrency problem with an example**.

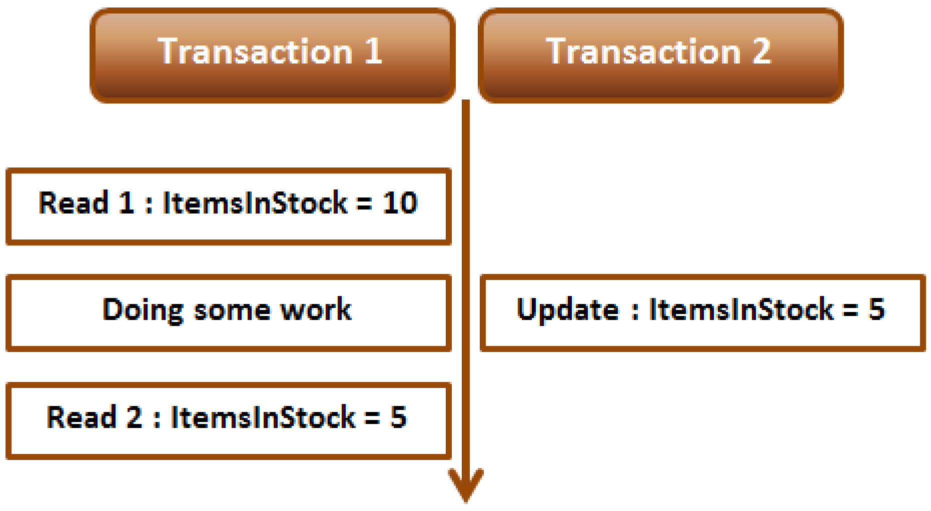
Non repeatable read problem happens when one transaction reads the same data twice and another transaction updates that data in between the first and second read of transaction one.

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



Id	Name	ItemsInStock
1	iPhone	10

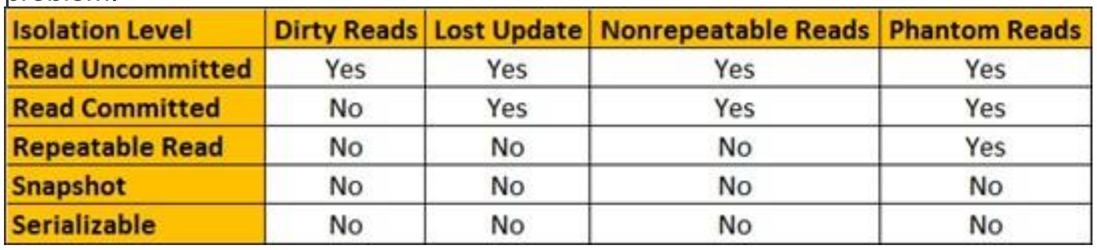
The following diagram explains the problem : Transaction 1 starts first. Reads ItemsInStock. Gets a value of 10 for first read. Transaction 1 is doing some work and at this point Transaction 2 starts and UpdatesItemsInStock to 5. Transaction 1 then makes a second read. At this point Transaction 1 gets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

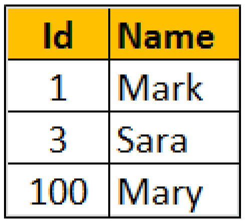
```
-- Transaction 1
Set transaction isolation level repeatable read
Begin Transaction
Select ItemsInStock from tblInventory where Id = 1

-- 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



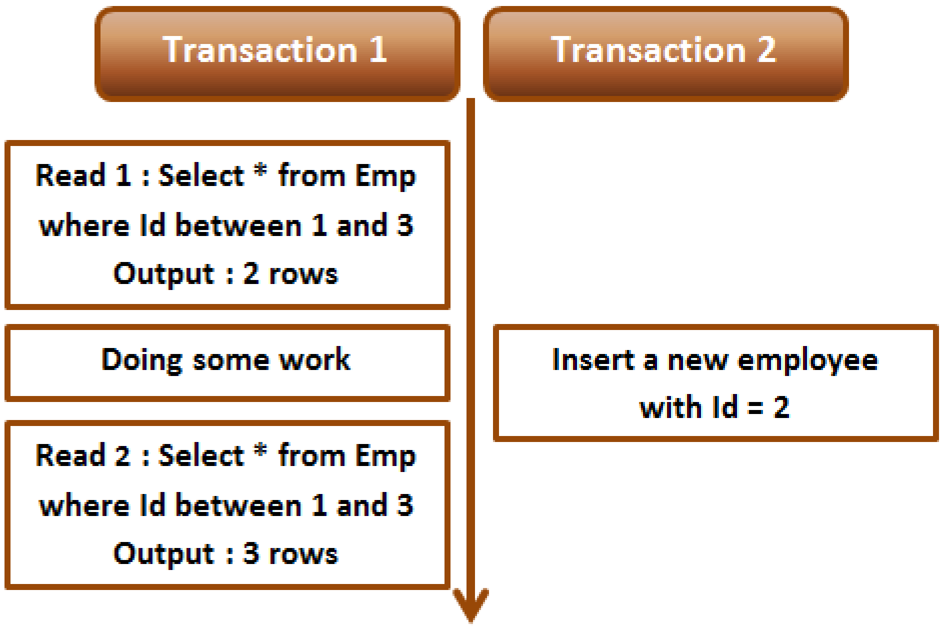
Id	Name
1	Mark
3	Sara
100	Mary

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')
```

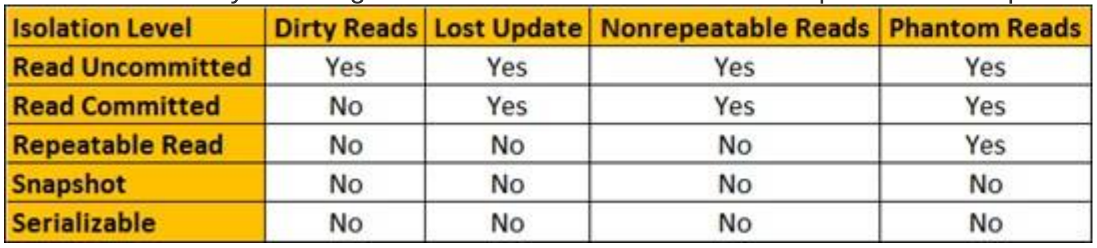
The following diagram explains the problem : Transaction 1 starts first. Reads from Emp table where Id between 1 and 3. 2 rows retrieved for first read. Transaction 1 is doing some work and at this point Transaction 2 starts and inserts a new employee with Id = 2. Transaction 1 then makes a second read.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

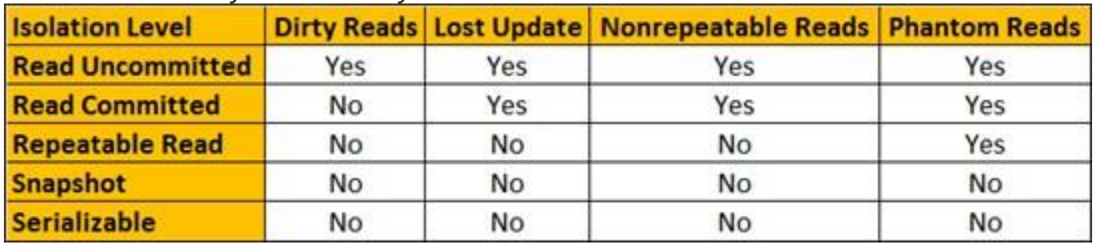
Difference between repeatable read and serializable
Repeatable read prevents only non-repeatable read. Repeatable read isolation level ensures that the data that one transaction has read, will be prevented from being updated or deleted by any other transaction, but it doe not prevent new rows from being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

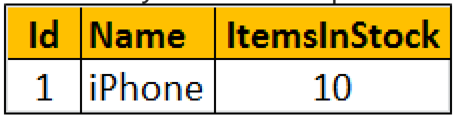


Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

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.
```

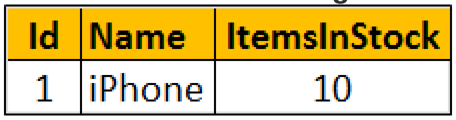
```
--Transaction 1
Set transaction isolation level serializable
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
waitfor delay '00:00:10'
Commit Transaction
```

```
-- 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

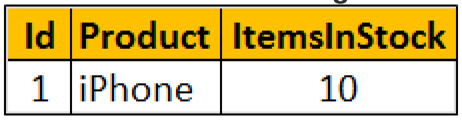
In this video we will discuss the differences between snapshot isolation and read committed snapshot isolation in sql server. This is continuation to Parts 75 and 76. Please watch Part 75 and 76 from SQL Server tutorial before proceeding.

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.

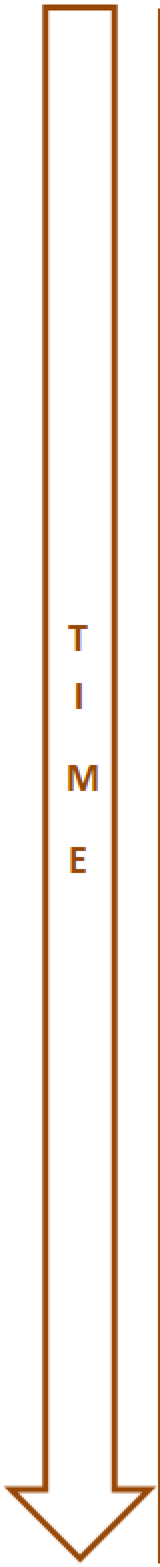
```
--Transaction 1
Set transaction isolation level read committed
Begin Transaction
Update tblInventory set ItemsInStock = 8 where Id = 1
waitfor delay '00:00:10'
Commit Transaction

-- 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.

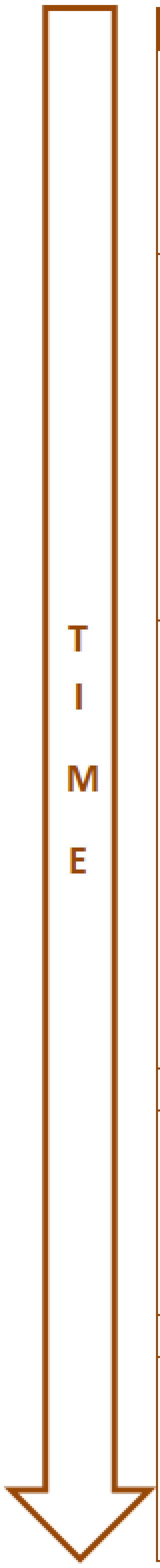
Read consistency : Read Committed Snapshot Isolation provides statement-level read consistency where as Snapshot Isolation provides transaction-level read consistency. The following diagrams explain this.

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`

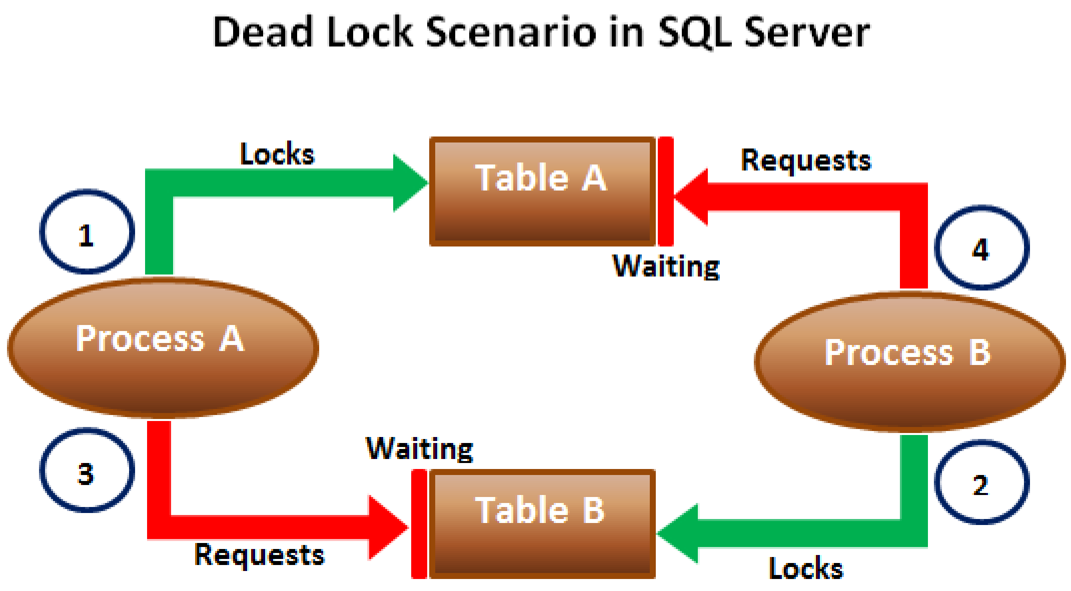
	`Commit Transaction`

`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.



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.

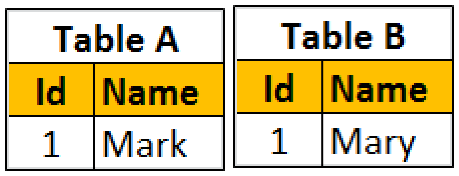


Table A	
Id	Name
1	Mark

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

n 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
```

Part 80 - 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 inPart 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

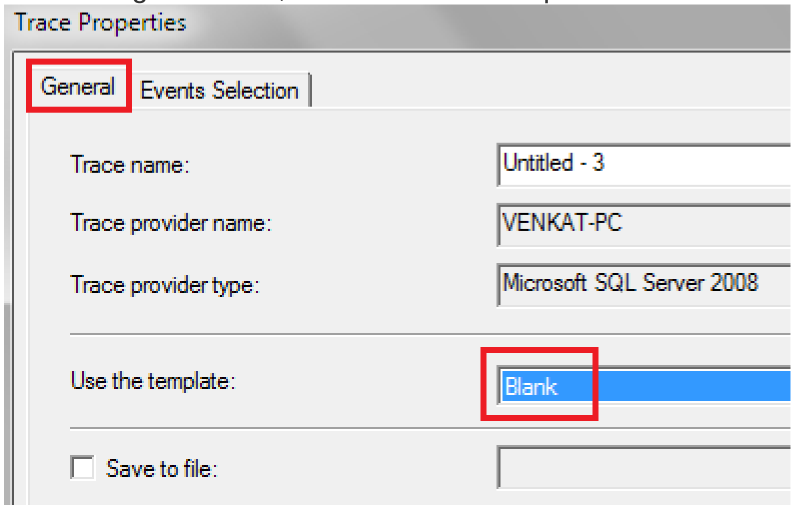
To prevent the deadlock that we have in our case, we need to ensure that database objects (Table A & Table B) are accessed in the same order every time.

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

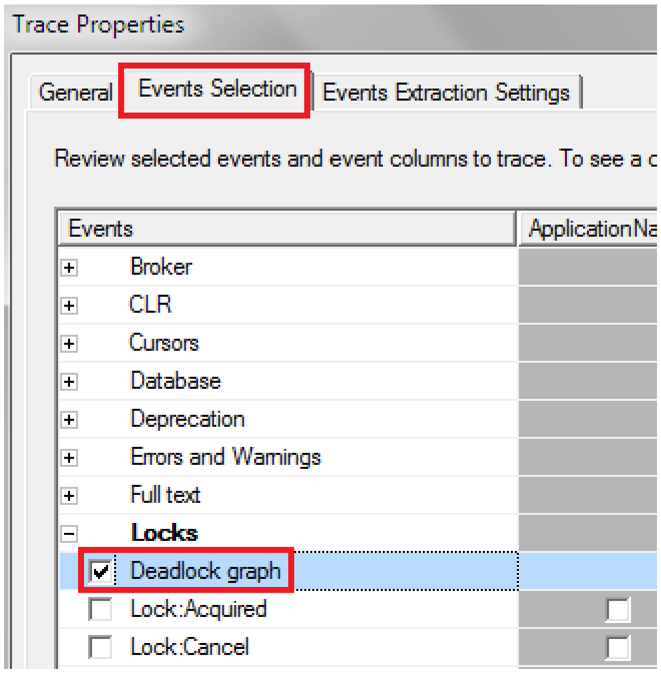


Trace Properties

General | Events Selection

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

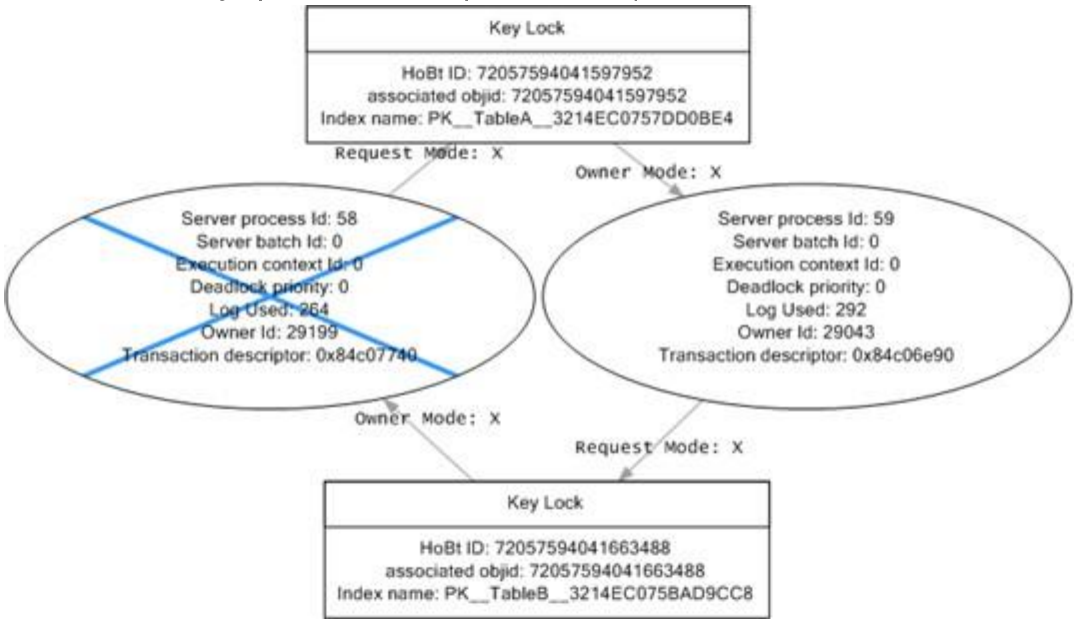
4. On the **"Events Selection"** tab, expand **"Locks"** section and select **"Deadlock graph"** event



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

In this video we will discuss **how to handle deadlock errors in an ADO.NET application**.

To handle deadlock errors in ADO.NET
1. Catch the SqlException object
2. Check if the error is deadlock error using the Number property of the SqlException object

Stored Procedure 1 Code

```
Alter 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
```

Stored Procedure 2 Code

```
Alter 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
```

WebForm1.aspx HTML

```
<table>
   <tr>
      <td>
         <asp:Button ID="Button1" runat="server"
            Text="Update Table A and then Table B"
            OnClick="Button1_Click" />
        </td>
    </tr>
    <tr>
        <td>
            <asp:Label ID="Label1" runat="server"></asp:Label>
        </td>
    </tr>
</table>
```

WebForm1.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }

        protected void Button1_Click(object sender, EventArgs e)
        {
            try
            {
                string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction1", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    Label1.Text = "Transaction successful";
                    Label1.ForeColor = System.Drawing.Color.Green;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    Label1.Text = "Deadlock. Please retry";
                }
                else
                {
                    Label1.Text = ex.Message;
                }
                Label1.ForeColor = System.Drawing.Color.Red;
            }
        }
    }
}
```

WebForm2.aspx HTML

```
<table>
    <tr>
        <td>
            <asp:Button ID="Button1" runat="server"
                Text="Update Table B and then Table A"
                OnClick="Button1_Click" />
        </td>
    </tr>
    <tr>
        <td>
            <asp:Label ID="Label1" runat="server"></asp:Label>
        </td>
    </tr>
</table>
```

WebForm2.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }
        protected void Button1_Click(object sender, EventArgs e)
        {
            try
            {
                string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction1", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    Label1.Text = "Transaction successful";
                    Label1.ForeColor = System.Drawing.Color.Green;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    Label1.Text = "Deadlock. Please retry";
                }
                else
                {
                    Label1.Text = ex.Message;
                }
                Label1.ForeColor = System.Drawing.Color.Red;
            }
        }
    }
}
```

Part 85 - Retry logic for deadlock exceptions

In this video we will discuss implementing **retry logic for deadlock exceptions**.

This is continuation to Part 84. Please watch Part 84, before proceeding.

When a transaction fails due to deadlock, we can write some logic so the system can resubmit the transaction. The deadlocks usually last for a very short duration. So upon resubmitting the transaction it may complete successfully. This is much better from user experience standpoint.

To achieve this we will be using the following technologies
C#
ASP.NET
SQL Server
jQuery AJAX

Result.cs

```
public class Result
{
    public int AttemptsLeft { get; set; }
    public string Message { get; set; }
    public bool Success { get; set; }
}
```

WebForm1.aspx HTML and jQuery code

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <script src="jquery-1.11.2.js"></script>
    <script type="text/javascript">
        $(document).ready(function () {
            var lblMessage = $('#Label1');
            var attemptsLeft;

            function updateData() {
                $.ajax({
                    url: 'WebForm1.aspx/CallStoredProcedure',
                    method: 'post',
                    contentType: 'application/json',
                    data: '{attemptsLeft:' + attemptsLeft + '}',
                    dataType: 'json',
                    success: function (data) {
                        lblMessage.text(data.d.Message);
                        attemptsLeft = data.d.AttemptsLeft;
                        if (data.d.Success) {
                            $('#btn').prop('disabled', false);
                            lblMessage.css('color','green');
                        }
                        else if(attemptsLeft > 0){
                            lblMessage.css('color', 'red');
                            updateData();
                    }
                    else {
                        lblMessage.css('color', 'red');
                        lblMessage.text('Deadlock Occurred. ZERO attempts left. Please try later');
                    }
                },
                error: function (err) {
                    lblMessage.css('color', 'red');
                    lblMessage.text(err.responseText);
                }
            });
        }

        $('#btn').click(function () {
            $(this).prop('disabled', true);
            lblMessage.text('Updating ...');
            attemptsLeft = 5;
            updateData();
        });
    });
  </script>
</head>
<body style="font-family: Arial">
  <form id="form1" runat="server">
    <input id="btn" type="button"
        value="Update Table A and then Table B" />
    <br />
    <asp:Label ID="Label1" runat="server"></asp:Label>
  </form>
</body>
</html>
```

WebForm1.aspx.cs code

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
```

```
{
    protected void Page_Load(object sender, EventArgs e)
    { }

    [System.Web.Services.WebMethod]
    public static Result CallStoredProcedure(int attemptsLeft)
    {
        Result _result = new Result();
        if (attemptsLeft > 0)
        {
            try
            {
                string cs =ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
                using (SqlConnection con = new SqlConnection(cs))
                {
                    SqlCommand cmd = new SqlCommand("spTransaction15", con);
                    cmd.CommandType = CommandType.StoredProcedure;
                    con.Open();
                    cmd.ExecuteNonQuery();
                    _result.Message = "Transaction successful";
                    _result.AttemptsLeft = 0;
                    _result.Success = true;
                }
            }
            catch (SqlException ex)
            {
                if (ex.Number == 1205)
                {
                    _result.AttemptsLeft = attemptsLeft - 1;
                    _result.Message = "Deadlock occurred. Retrying. Attempts left : "
                        + _result.AttemptsLeft.ToString();
                }
                else
                {
                    throw;
                }
                _result.Success = false;
            }
        }
        return _result;
    }
    }
}
```

Copy and paste the above code in WebForm2.aspx and make the required changes as described in the video.

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.

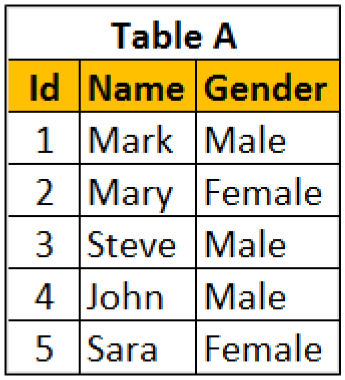


Table A		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
4	John	Male
5	Sara	Female

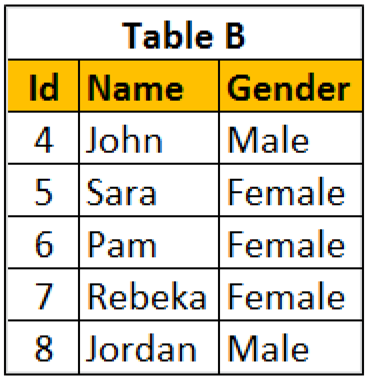


Table B		
Id	Name	Gender
4	John	Male
5	Sara	Female
6	Pam	Female
7	Rebeka	Female
8	Jordan	Male

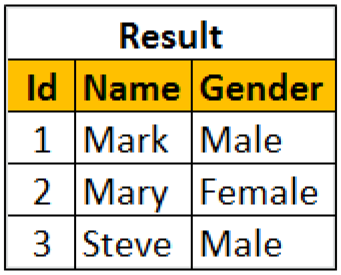
SQL Script to create the tables

```
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 :

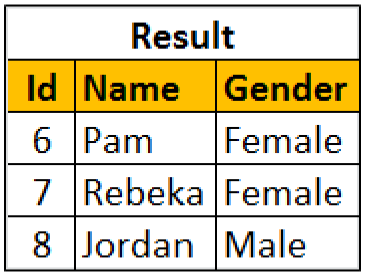


Result		
Id	**Name**	**Gender**
1	Mark	Male
2	Mary	Female
3	Steve	Male

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

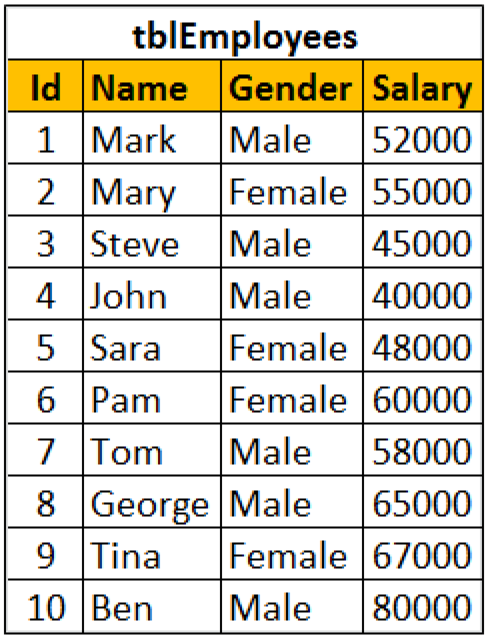
```
Select Id, Name, Gender
From TableB
Except
Select Id, Name, Gender
From TableA
```

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.

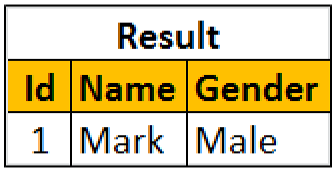
Table A		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male

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.

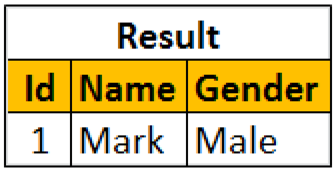
Insert the following row into TableA

```
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:

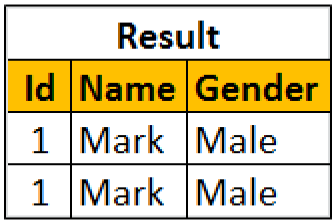


Result		
Id	Name	Gender
1	Mark	Male

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.

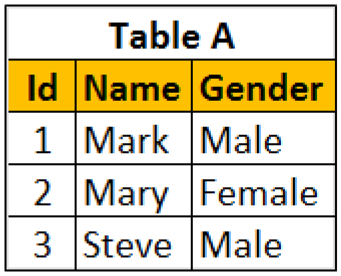


Table A

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

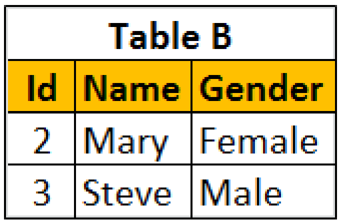


Table B

Id	Name	Gender
2	Mary	Female
3	Steve	Male

SQL Script to create the tables and populate with test data

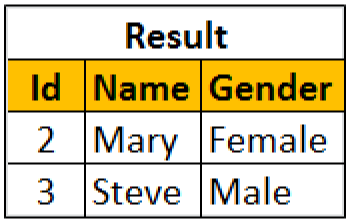
```
Create Table TableA
```

```
(
    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.

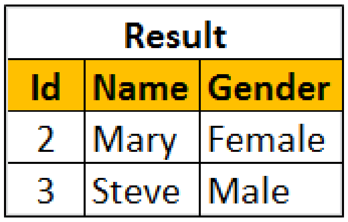
To understand this difference, insert the following row into TableA

```
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 :

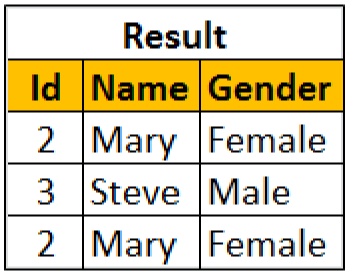


Result		
Id	Name	Gender
2	Mary	Female
3	Steve	Male

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 :

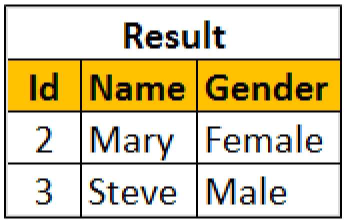


Result		
Id	Name	Gender
2	Mary	Female
3	Steve	Male
2	Mary	Female

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.

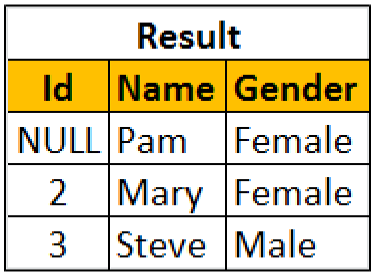
To understand this difference, execute the following 2 insert statements

```
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 :

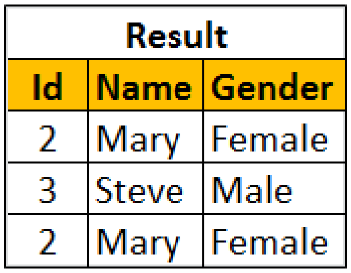


Result		
Id	**Name**	**Gender**
NULL	Pam	Female
2	Mary	Female
3	Steve	Male

INNER JOIN query

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

Result :

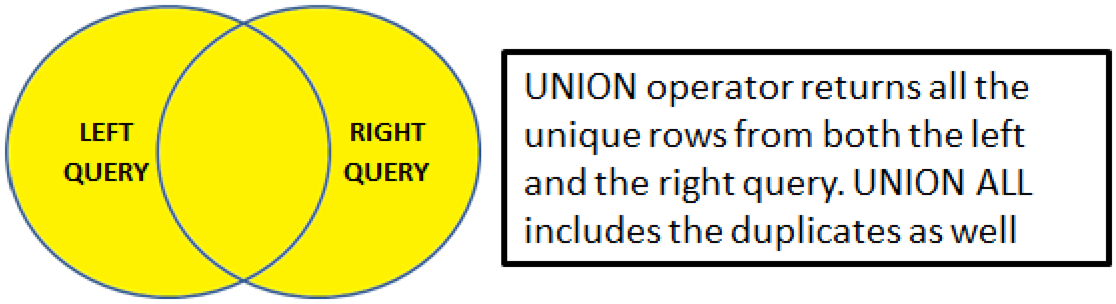


Result		
Id	**Name**	**Gender**
2	Mary	Female
3	Steve	Male
2	Mary	Female

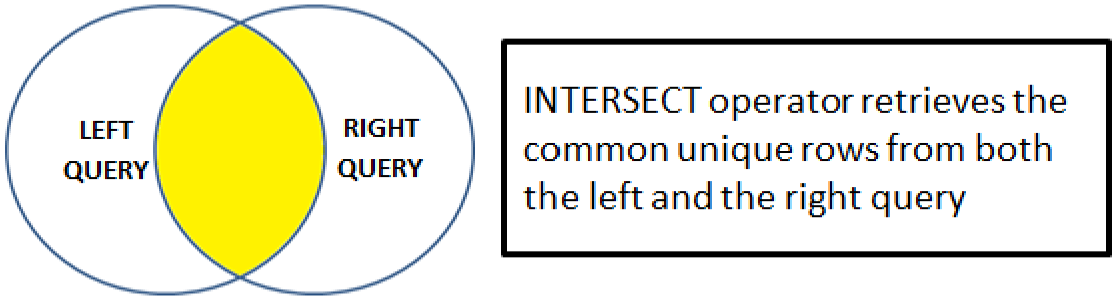
Part 90 - Difference between union intersect and except in sql server

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

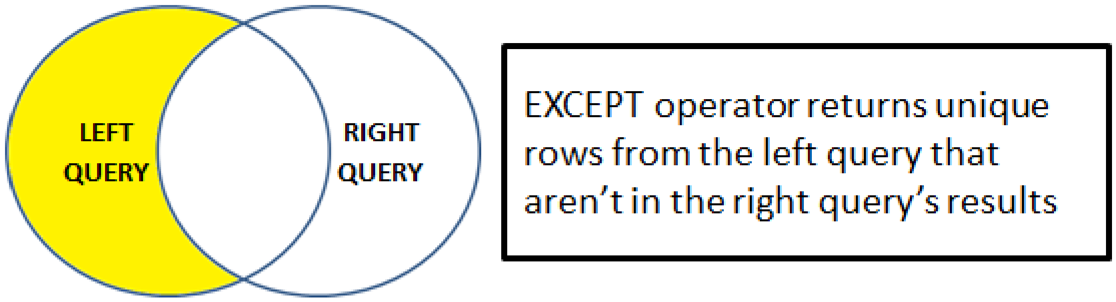
The following diagram explains the difference graphically



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.

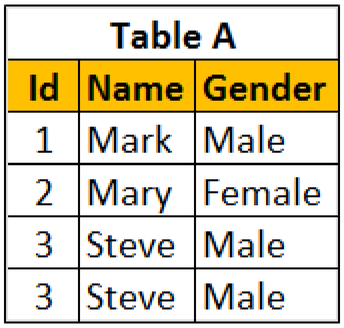


Table A		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
3	Steve	Male

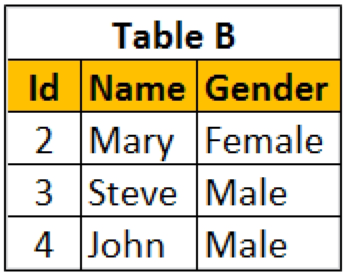


Table B		
Id	Name	Gender
2	Mary	Female
3	Steve	Male
4	John	Male

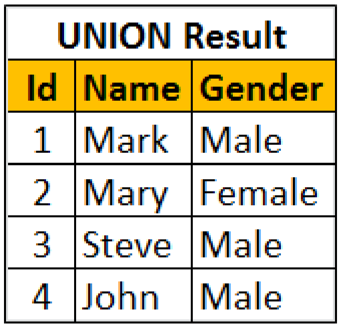
SQL Script to create the tables

```
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 :

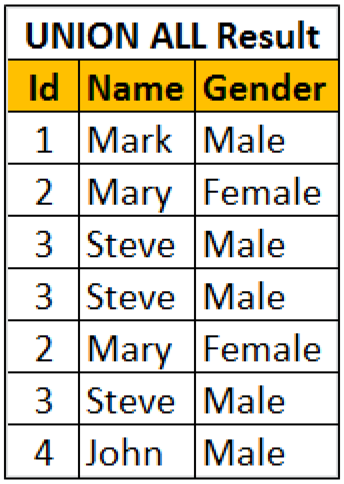


UNION Result		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
4	John	Male

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 :

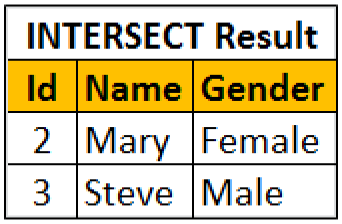


UNION ALL Result		
Id	Name	Gender
1	Mark	Male
2	Mary	Female
3	Steve	Male
3	Steve	Male
2	Mary	Female
3	Steve	Male
4	John	Male

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 :



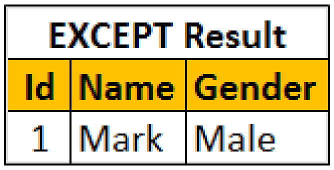
INTERSECT Result

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 :



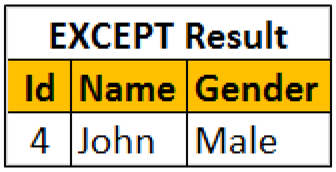
EXCEPT Result

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

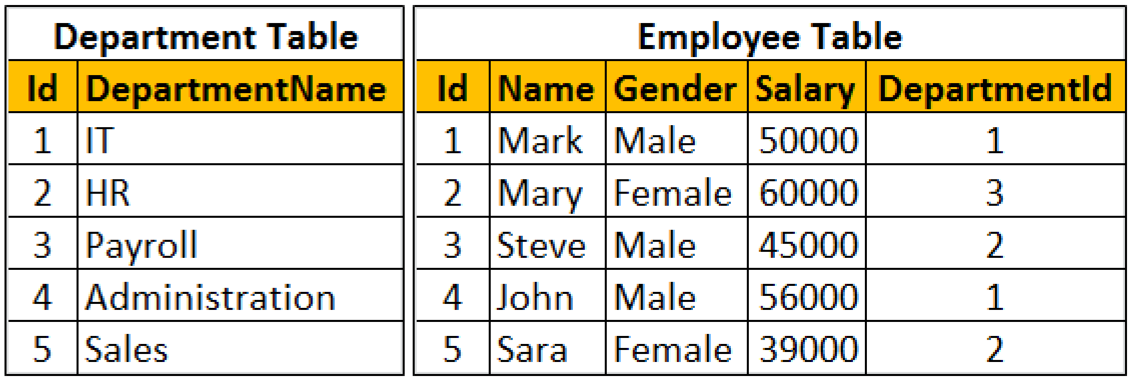
For example, if the number of columns are different, you will get the following error

```
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

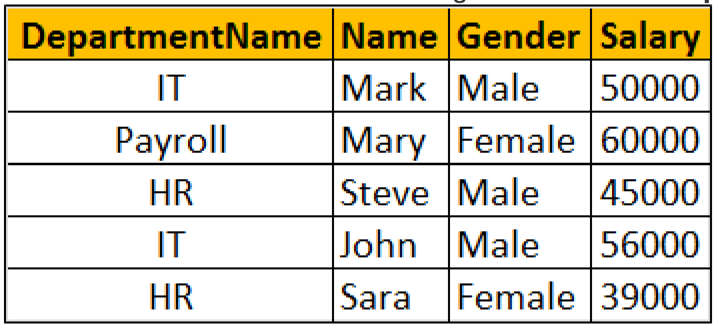
Employee Table

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.

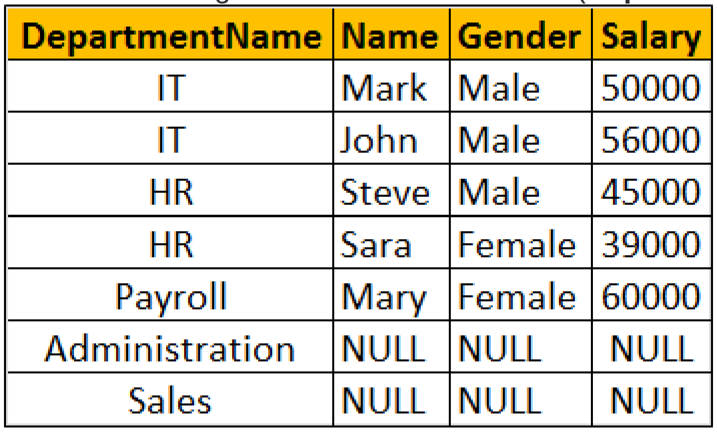


DepartmentName	Name	Gender	Salary
IT	Mark	Male	50000
Payroll	Mary	Female	60000
HR	Steve	Male	45000
IT	John	Male	56000
HR	Sara	Female	39000

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
```

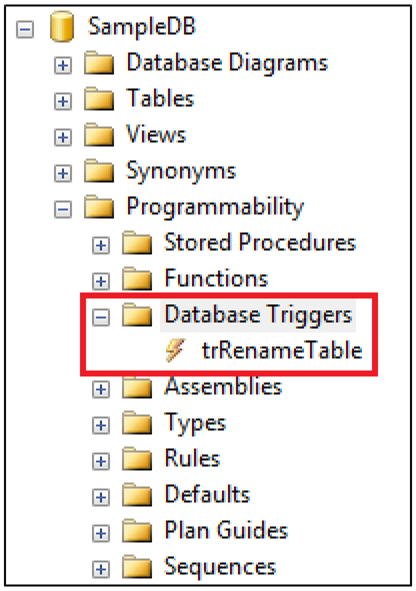
DDL triggers scope : DDL triggers can be created in a specific database or at the server level.

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 usingsp_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.

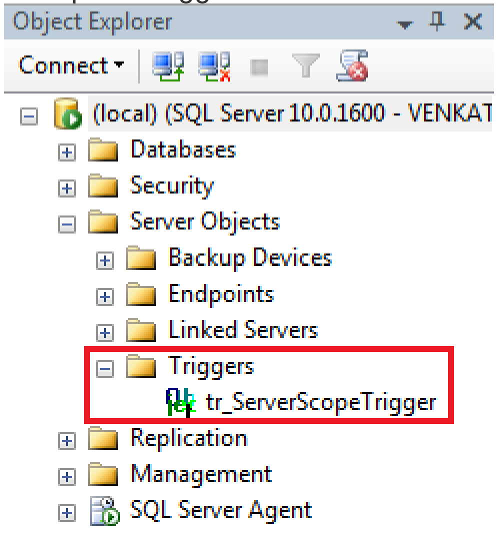
This is where server-scoped DDL triggers come in handy. When you create a server scoped DDL trigger, it will fire in response to the DDL events happening in all of the databases on that server.

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**.

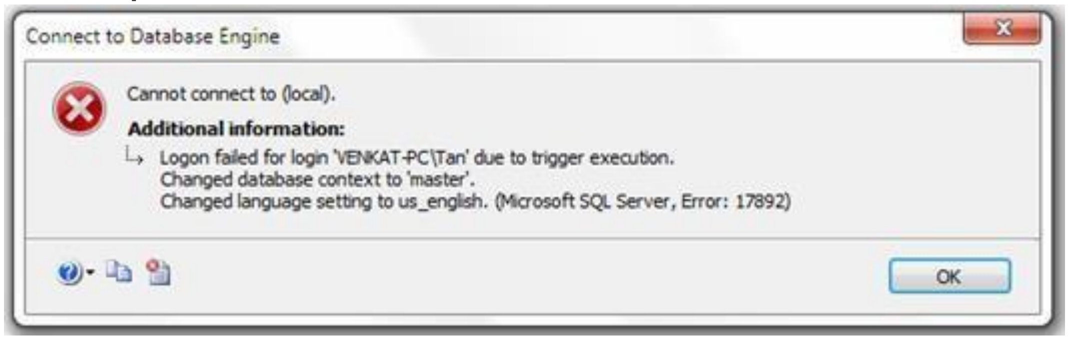
As the name implies **Logon triggers fire in response to a LOGON event**. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established.

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

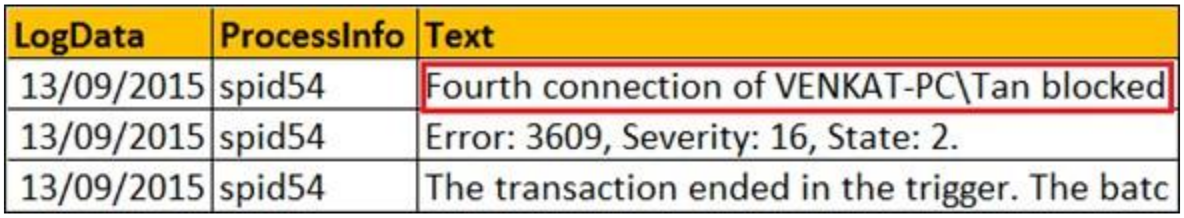
Cannot connect to (local).

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)

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

```
Execute sp_readerrorlog
```

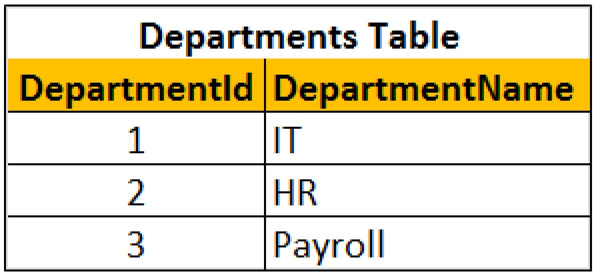


LogData	ProcessInfo	Text
13/09/2015	spid54	Fourth connection of VENKAT-PC\Tan blocked
13/09/2015	spid54	Error: 3609, Severity: 16, State: 2.
13/09/2015	spid54	The transaction ended in the trigger. The batc

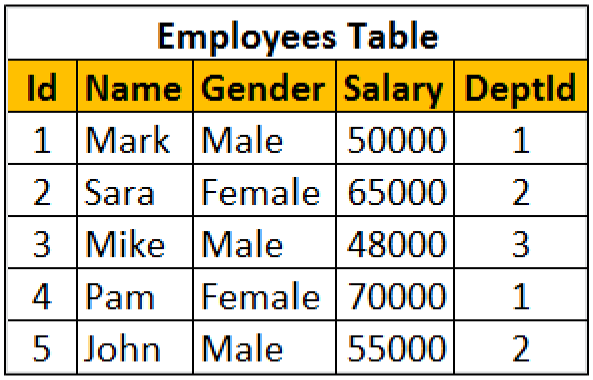
Part 97 - Select into in sql server

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
```

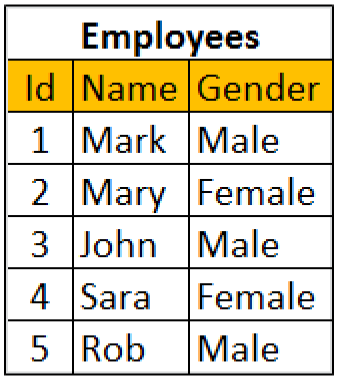
Step 2 : Use the User-defined Table Type as a parameter in the stored procedure. Table valued parameters must be passed as read-only to stored procedures, functions etc. This means you cannot perform DML operations like INSERT, UPDATE or DELETE on a table-valued parameter in the body of a function, stored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

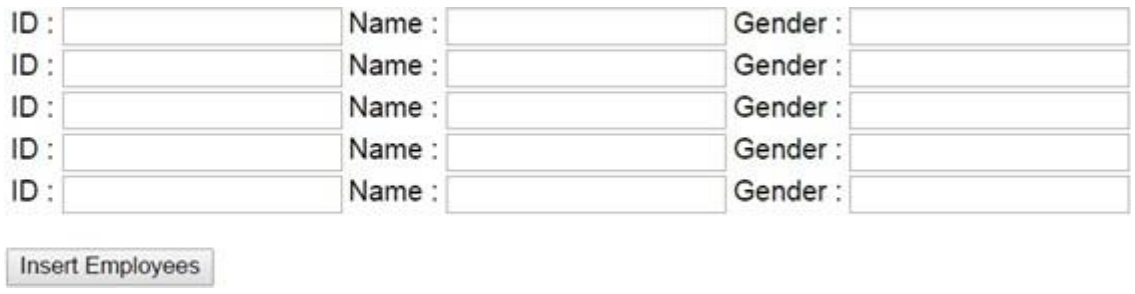
In our next video, we will discuss **how to pass table as a parameter to the stored procedure from an ADO.NET application**

Part 100 - Send datatable as parameter to stored procedure

In this video we will discuss **how to send datatable as parameter to stored procedure**. This is continuation to Part 99. Please watch Part 99 from SQL Server tutorial before proceeding.

In Part 99, we discussed creating a stored procedure that accepts a table as a parameter. In this video we will discuss **how to pass a datatable from a web application to the SQL Server stored procedure**.

Here is what we want to do.
1. Design a webform that looks as shown below. This form allows us to insert 5 employees at a time into the database table.



ID :		Name :		Gender :	
ID :		Name :		Gender :	
ID :		Name :		Gender :	
ID :		Name :		Gender :	
ID :		Name :		Gender :	

2. When "Insert Employees" button is clicked, retrieve the from data into a datatabe and then pass the datatable as a parameter to the stored procedure.

3. The stored procedure will then insert all the rows into the Employees table in the database.

Here are the steps to achieve this.

Step 1 : Create new asp.net web application project. Name it Demo.

Step 2 : Include a connection string in the web.config file to your database.

```
<add name="DBCS"
     connectionString="server=.;database=SampleDB;integrated security=SSPI"/>
```

Step 3 : Copy and paste the following HTML in WebForm1.aspx

```
<asp:Button ID="btnFillDummyData" runat="server" Text="Fill Dummy Data"
    OnClick="btnFillDummyData_Click" />
<br /><br />
<table>
    <tr>
        <td>
            ID : <asp:TextBox ID="txtId1" runat="server"></asp:TextBox>
        </td>
        <td>
            Name : <asp:TextBox ID="txtName1" runat="server"></asp:TextBox>
        </td>
        <td>
            Gender : <asp:TextBox ID="txtGender1" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            ID : <asp:TextBox ID="txtId2" runat="server"></asp:TextBox>
        </td>
        <td>
            Name : <asp:TextBox ID="txtName2" runat="server"></asp:TextBox>
        </td>
        <td>
            Gender : <asp:TextBox ID="txtGender2" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            ID : <asp:TextBox ID="txtId3" runat="server"></asp:TextBox>
        </td>
        <td>
            Name : <asp:TextBox ID="txtName3" runat="server"></asp:TextBox>
        </td>
        <td>
            Gender : <asp:TextBox ID="txtGender3" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            ID : <asp:TextBox ID="txtId4" runat="server"></asp:TextBox>
        </td>
        <td>
            Name : <asp:TextBox ID="txtName4" runat="server"></asp:TextBox>
        </td>
        <td>
            Gender : <asp:TextBox ID="txtGender4" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            ID : <asp:TextBox ID="txtId5" runat="server"></asp:TextBox>
        </td>
        <td>
            Name : <asp:TextBox ID="txtName5" runat="server"></asp:TextBox>
        </td>
        <td>
            Gender : <asp:TextBox ID="txtGender5" runat="server"></asp:TextBox>
        </td>
    </tr>
</table>
<br />
<asp:Button ID="btnInsert" runat="server" Text="Insert Employees"
    OnClick="btnInsert_Click" />
```

Step 4 : Copy and paste the following code in the code-behind file

```
using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;

namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        { }
```

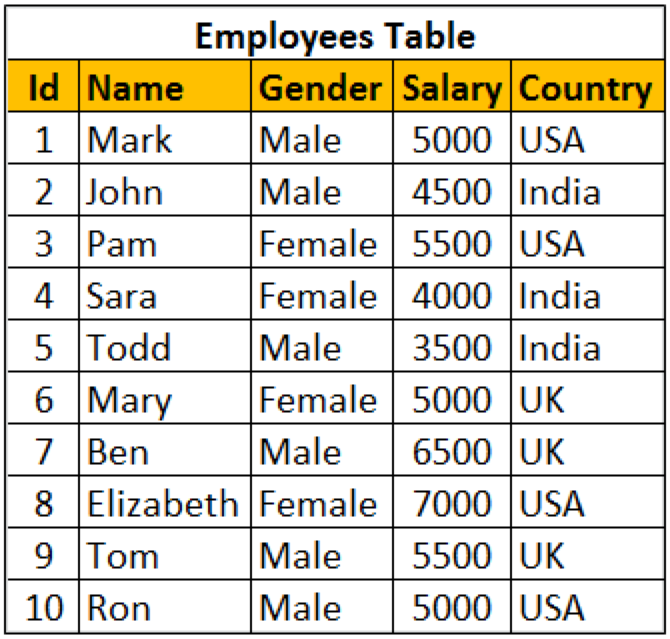
```
private DataTable GetEmployeeData()
{
    DataTable dt = new DataTable();
    dt.Columns.Add("Id");
    dt.Columns.Add("Name");
    dt.Columns.Add("Gender");

    dt.Rows.Add(txtId1.Text, txtName1.Text, txtGender1.Text);
    dt.Rows.Add(txtId2.Text, txtName2.Text, txtGender2.Text);
    dt.Rows.Add(txtId3.Text, txtName3.Text, txtGender3.Text);
    dt.Rows.Add(txtId4.Text, txtName4.Text, txtGender4.Text);
    dt.Rows.Add(txtId5.Text, txtName5.Text, txtGender5.Text);

    return dt;
}

protected void btnInsert_Click(object sender, EventArgs e)
{
    string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
    using (SqlConnection con = new SqlConnection(cs))
    {
        SqlCommand cmd = new SqlCommand("spInsertEmployees", con);
        cmd.CommandType = CommandType.StoredProcedure;

        SqlParameter paramTVP = new SqlParameter()
        {
            ParameterName = "@EmpTableType",
            Value = GetEmployeeData()
        };
        cmd.Parameters.Add(paramTVP);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
    }
}

protected void btnFillDummyData_Click(object sender, EventArgs e)
{
    txtId1.Text = "1";
```

```
            txtId2.Text = "2";
            txtId3.Text = "3";
            txtId4.Text = "4";
            txtId5.Text = "5";

            txtName1.Text = "John";
            txtName2.Text = "Mike";
            txtName3.Text = "Sara";
            txtName4.Text = "Pam";
            txtName5.Text = "Todd";

            txtGender1.Text = "Male";
            txtGender2.Text = "Male";
            txtGender3.Text = "Female";
            txtGender4.Text = "Female";
            txtGender5.Text = "Male";
        }
    }
}
```

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.

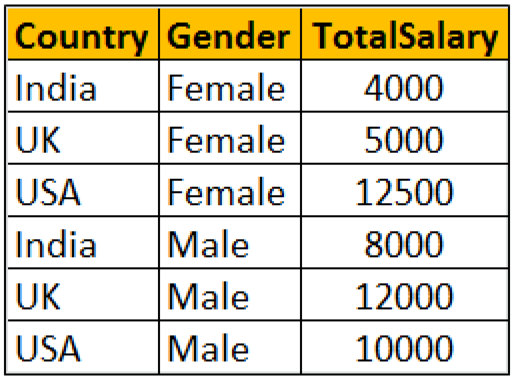


Employees Table				
Id	Name	Gender	Salary	Country
1	Mark	Male	5000	USA
2	John	Male	4500	India
3	Pam	Female	5500	USA
4	Sara	Female	4000	India
5	Todd	Male	3500	India
6	Mary	Female	5000	UK
7	Ben	Male	6500	UK
8	Elizabeth	Female	7000	USA
9	Tom	Male	5500	UK
10	Ron	Male	5000	USA

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.

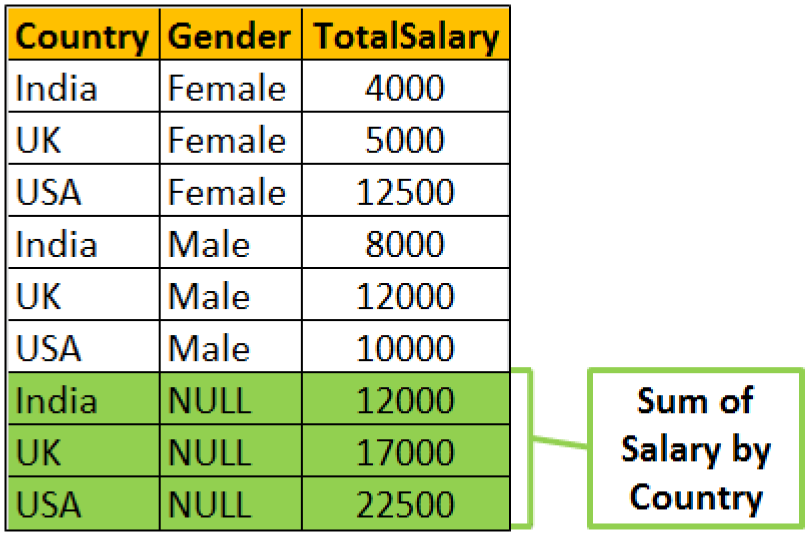


Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000

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

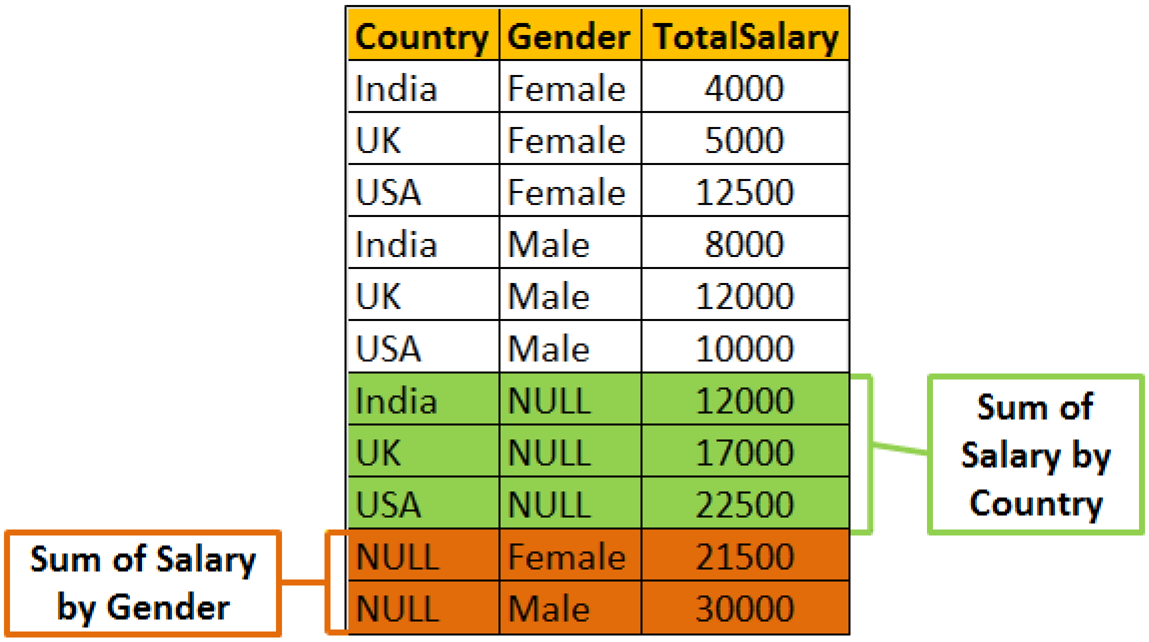
Sum of Salary by Country

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.



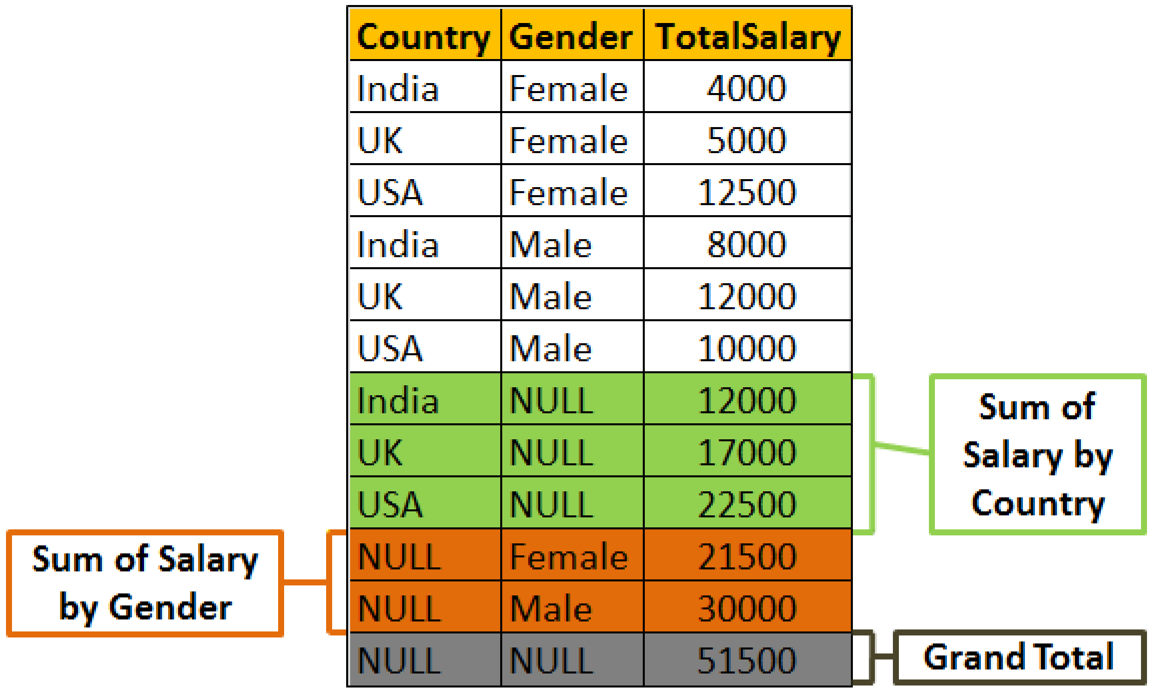
Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000
India	NULL	12000
UK	NULL	17000
USA	NULL	22500
NULL	Female	21500
NULL	Male	30000

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.


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

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

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
```

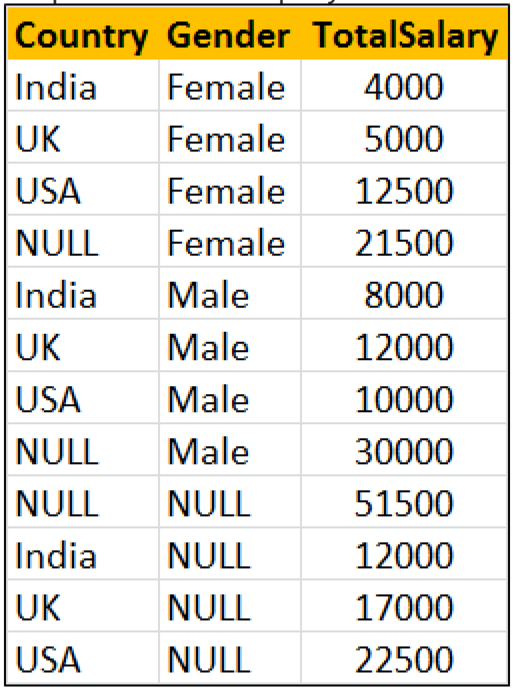
There are 2 problems with the above approach.
1. The query is huge as we have combined different Group By queries using UNION ALL

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

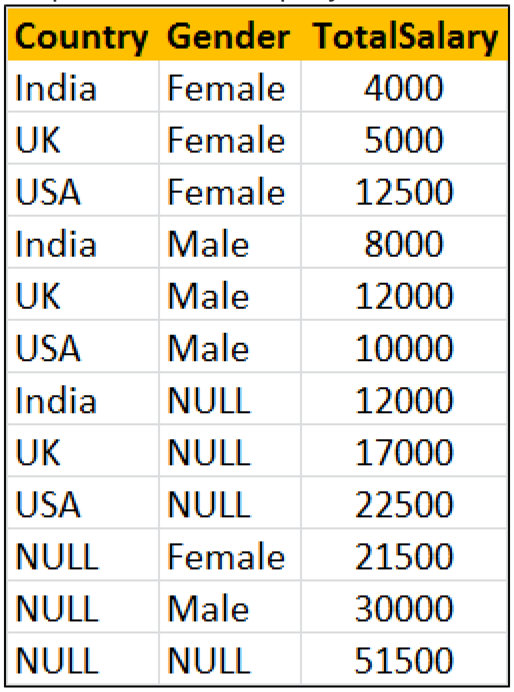


Country	Gender	TotalSalary
India	Female	4000
UK	Female	5000
USA	Female	12500
NULL	Female	21500
India	Male	8000
UK	Male	12000
USA	Male	10000
NULL	Male	30000
NULL	NULL	51500
India	NULL	12000
UK	NULL	17000
USA	NULL	22500

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 - Rollup 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
```

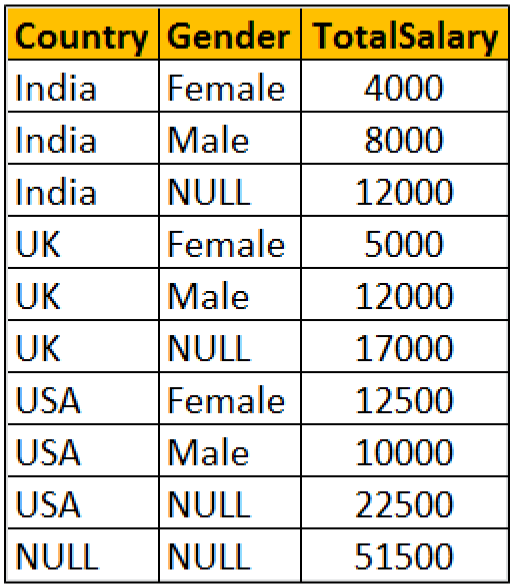
```
SELECT NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

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
```

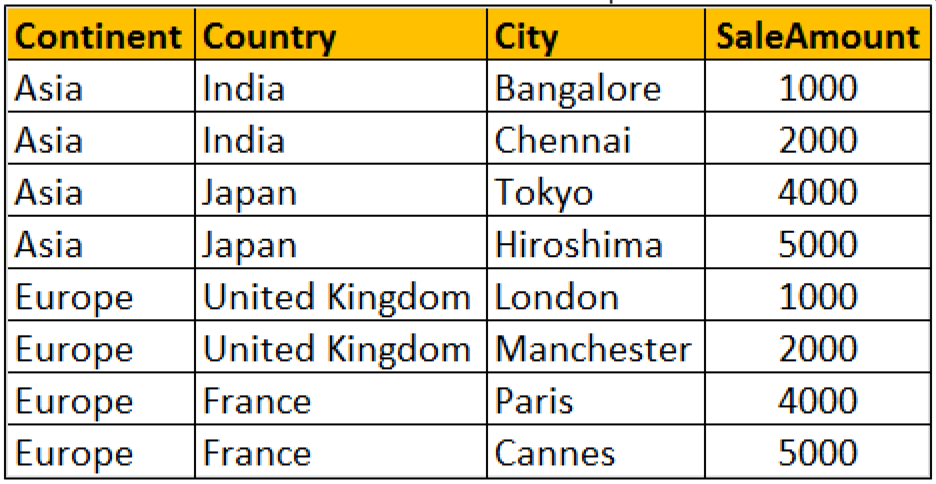
```
SELECT NULL, NULL, SUM(Salary) AS TotalSalary
FROM Employees
```

Part 104 - 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
```

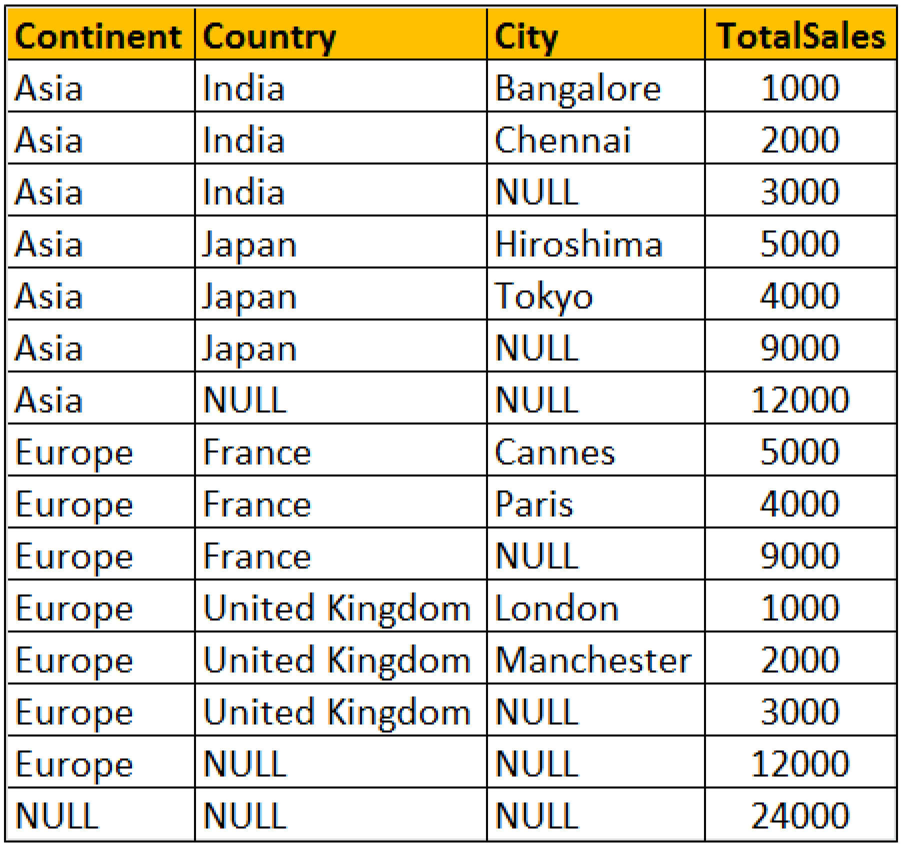
ROLLUP(Continent, Country, City) produces Sum of Salary for the following hierarchy

```
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)
```



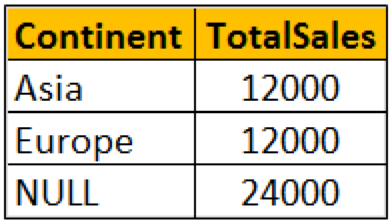
Continent	Country	City	TotalSales
Asia	India	Bangalore	1000
Asia	India	Chennai	2000
Asia	India	NULL	3000
Asia	Japan	Hiroshima	5000
Asia	Japan	Tokyo	4000
Asia	Japan	NULL	9000
Asia	NULL	NULL	12000
Europe	France	Cannes	5000
Europe	France	Paris	4000
Europe	France	NULL	9000
Europe	United Kingdom	London	1000
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

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)
```

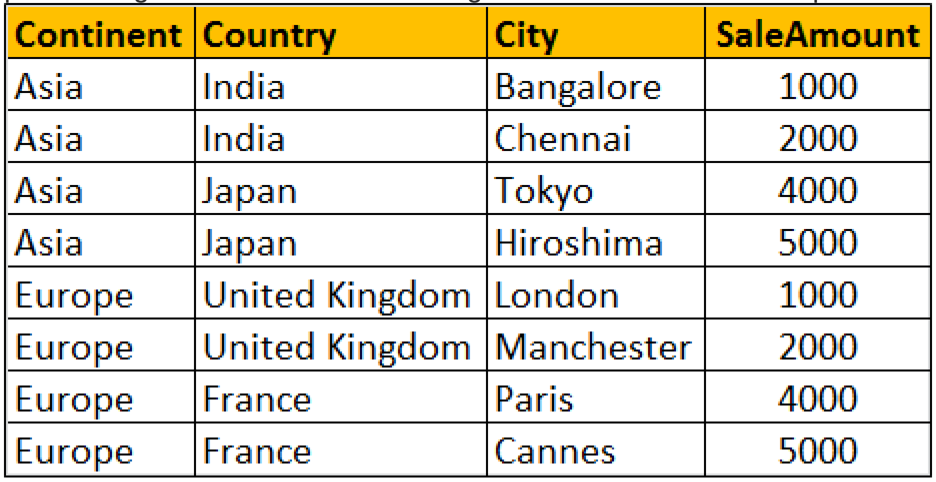


Continent	TotalSales
Asia	12000
Europe	12000
NULL	24000

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

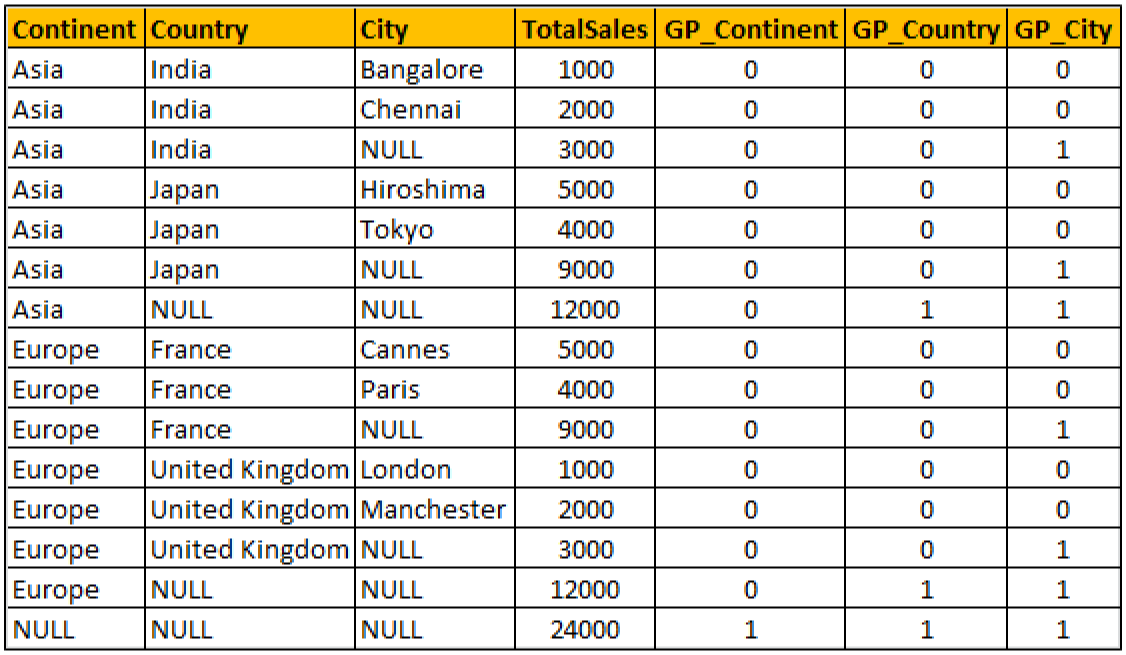
What is Grouping function

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 :

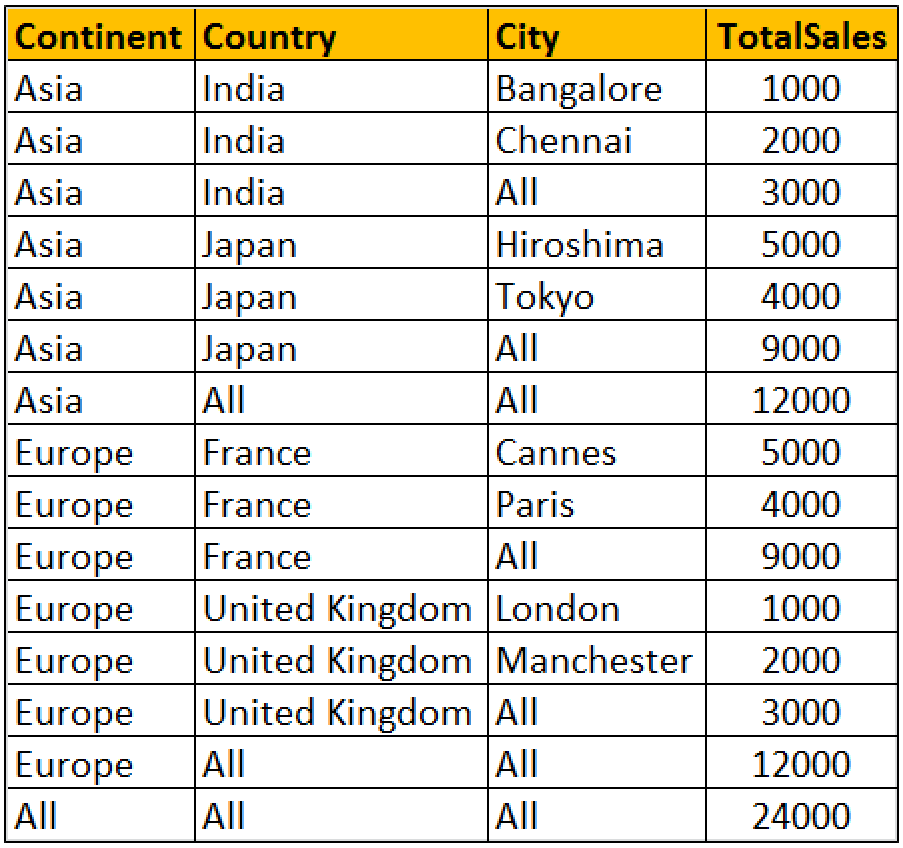


Continent	Country	City	TotalSales	GP_Continent	GP_Country	GP_City
Asia	India	Bangalore	1000	0	0	0
Asia	India	Chennai	2000	0	0	0
Asia	India	NULL	3000	0	0	1
Asia	Japan	Hiroshima	5000	0	0	0
Asia	Japan	Tokyo	4000	0	0	0
Asia	Japan	NULL	9000	0	0	1
Asia	NULL	NULL	12000	0	1	1
Europe	France	Cannes	5000	0	0	0
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.

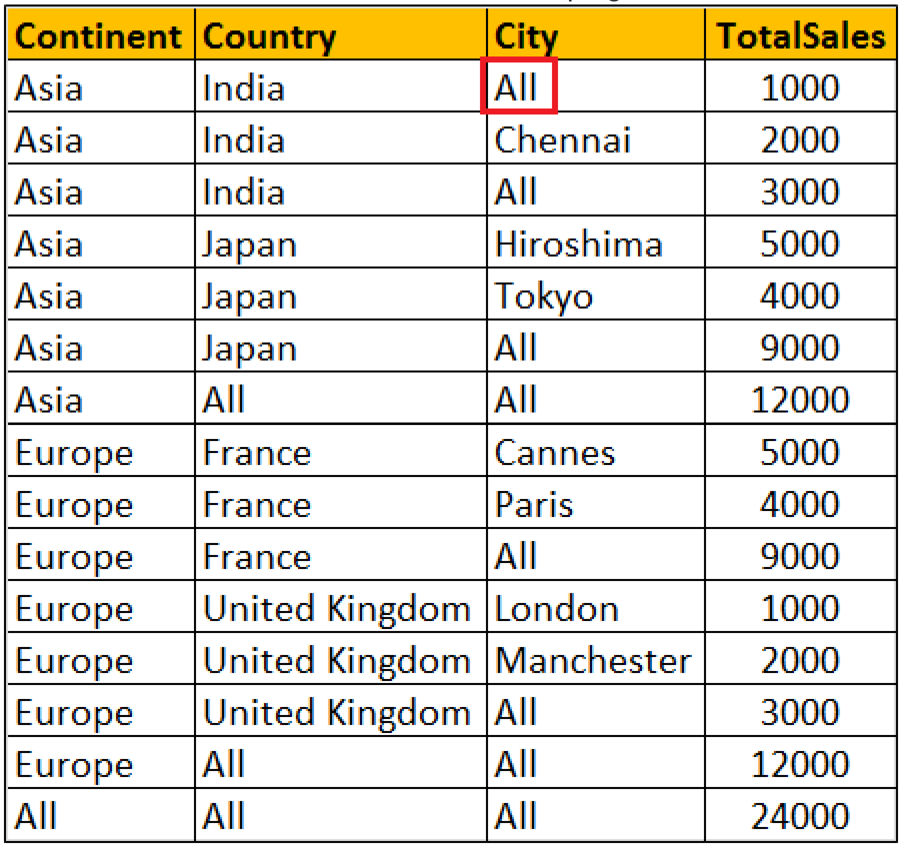
At the moment the raw data in our Sales has no NULL values. Let's introduce a NULL value in the City column of the row where Id = 1

```
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

Part 106 - 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,...)
```

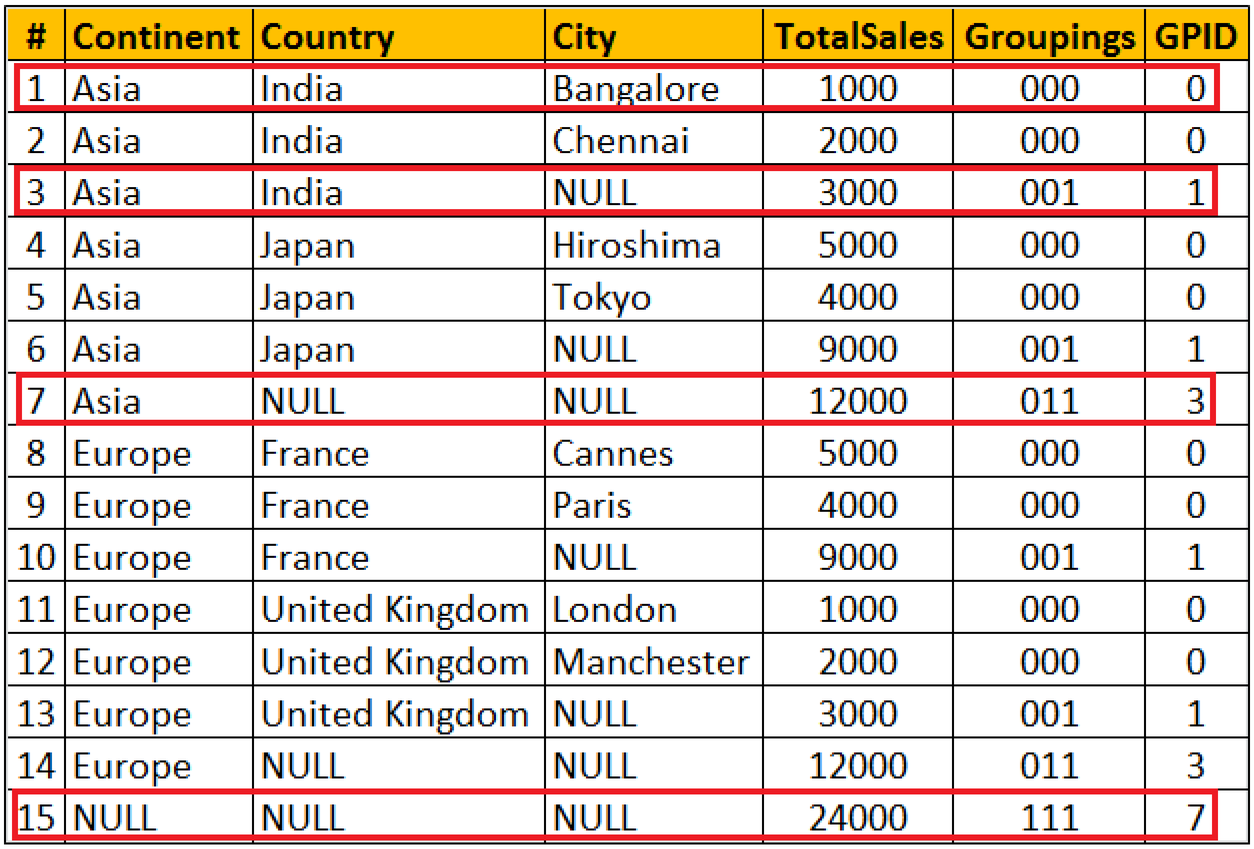
GROUPING indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set.

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
```

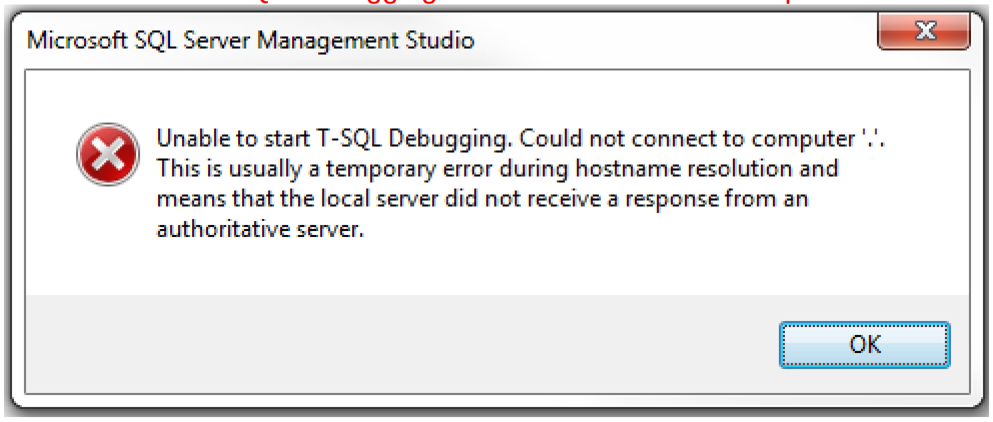
Result :

Continent	Country	City	TotalSales	GPID
Asia	NULL	NULL	12000	3
Europe	NULL	NULL	12000	3

Part 107 - Debugging sql server stored procedures

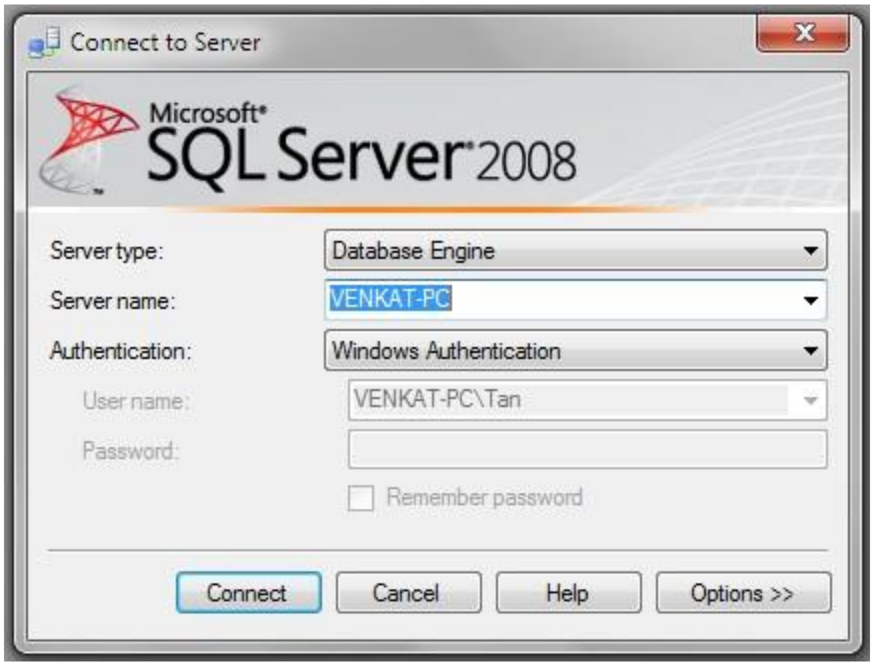
In this video we will discuss **how to debug stored procedures in SQL Server**.

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

Microsoft SQL Server 2008

Server type: Database Engine
Server name: VENKAT-PC
Authentication: Windows Authentication
User name: VENKAT-PC\Tan
Password:
Remember password

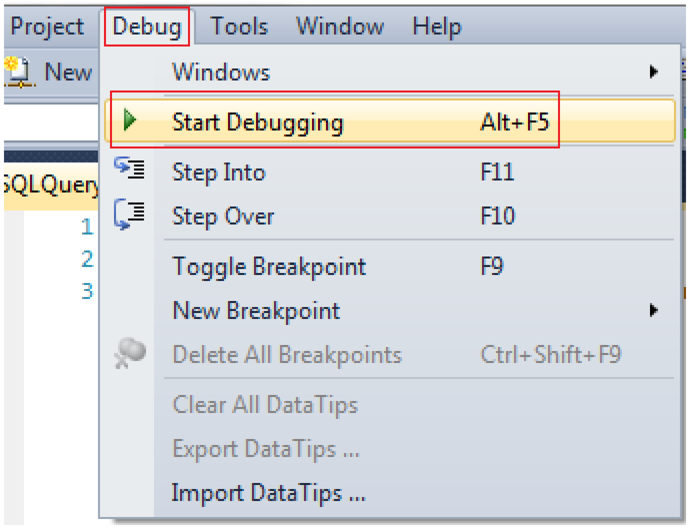
For the examples in this video we will be using the following stored procedure.

```
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.

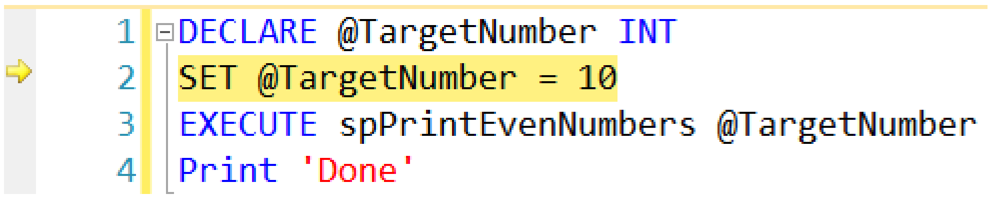
```
DECLARE @TargetNumber INT
SET @TargetNumber = 10
EXECUTE spPrintEvenNumbers @TargetNumber
Print 'Done'
```

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**



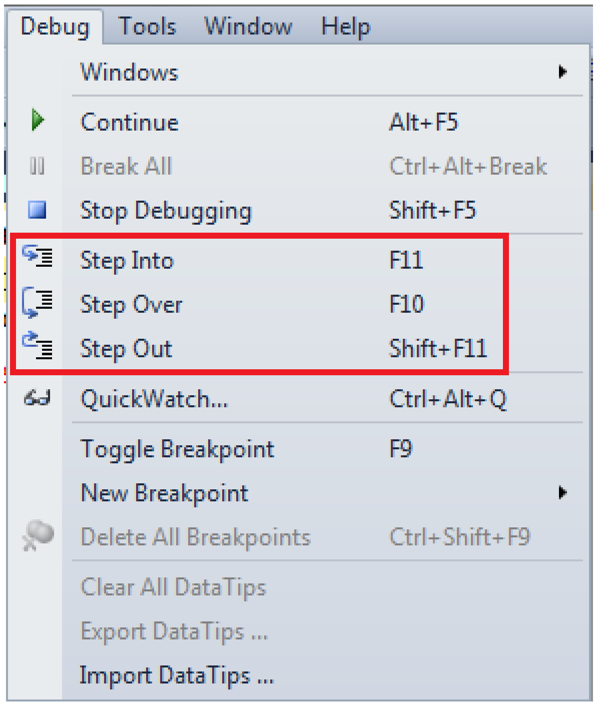
2. Use the keyboard shortcut **ALT + F5**

At this point you should have the debugger running. The line that is about to be executed is marked with an yellow arrow

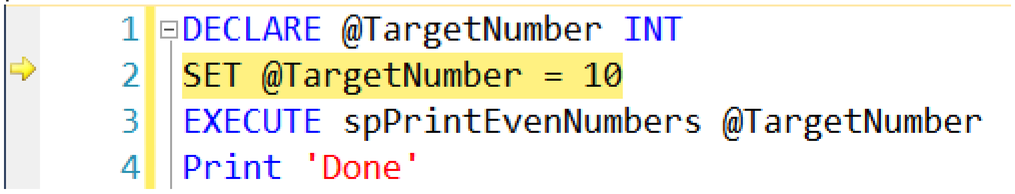


```
1 DECLARE @TargetNumber INT
2 SET @TargetNumber = 10
3 EXECUTE spPrintEvenNumbers @TargetNumber
4 Print 'Done'
```

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'
```

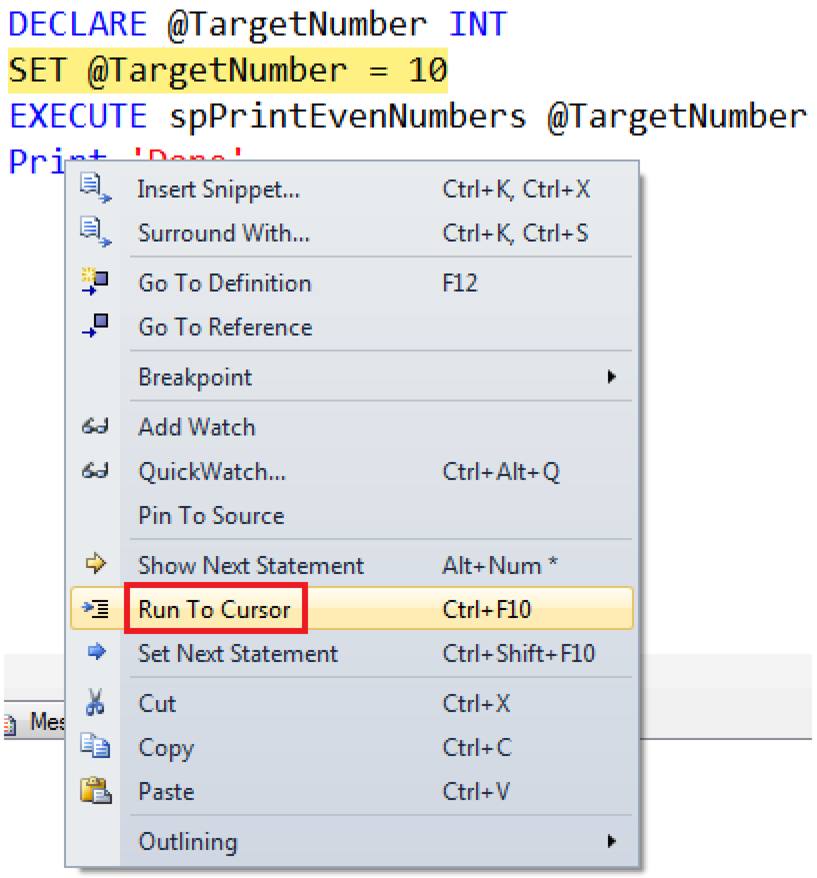
1. There is no difference when you STEP INTO (F11) or STEP OVER (F10) the code on LINE 2

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.

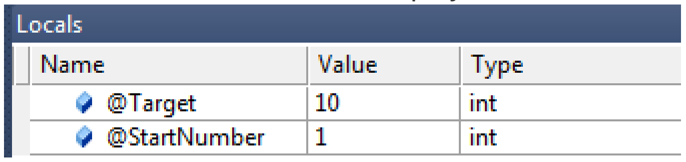
To stop debugging : There are 2 ways to stop debugging
1. In SSMS, click on the Debug Menu and select Stop Debugging
2. Use the keyboard shortcut SHIFT + F5

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



```sql
DECLARE @TargetNumber INT
SET @TargetNumber = 10
EXECUTE spPrintEvenNumbers @TargetNumber
Print 'Done'
```

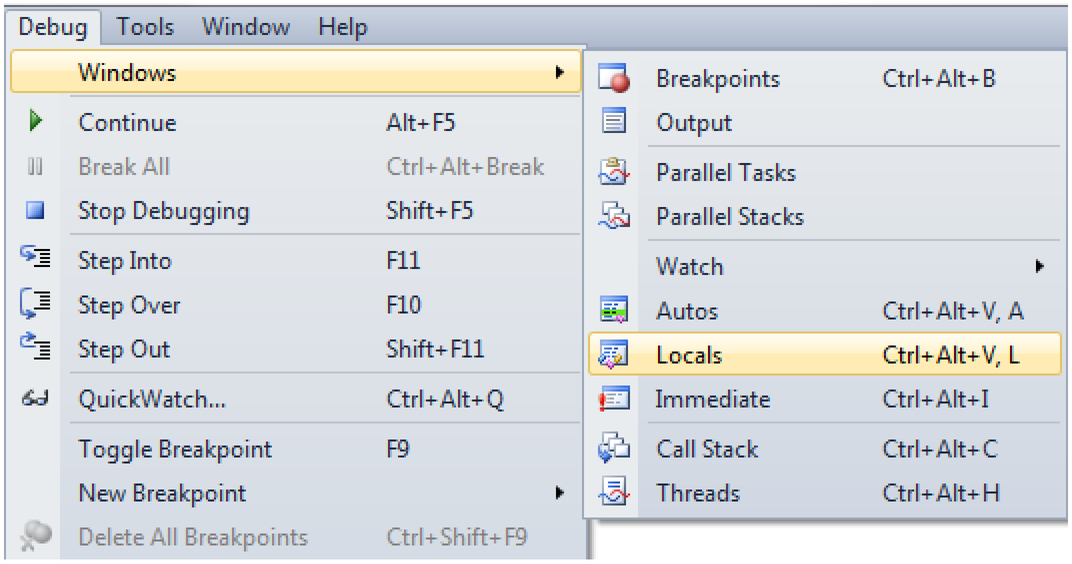
Locals Window in SSMS : Displays the current values of variables and parameters



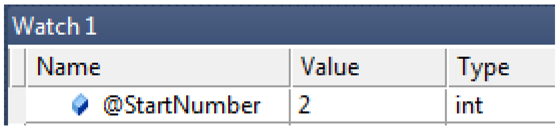
Locals

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.



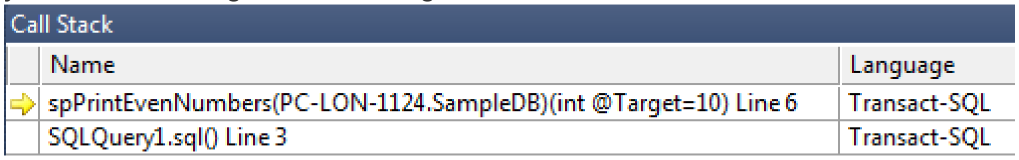
Watch Window in SSMS : Just like Locals window, Watch window is used to watch the values of variables. You can add and remove variables from the watch window. To add a variable to the Watch Window, right click on the variable and select "Add Watch" option from the context menu.



Watch 1

Name	Value	Type
@StartNumber	2	int

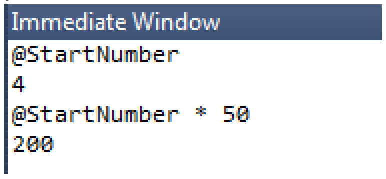
Call Stack Window in SSMS : Allows you to navigate up and down the call stack to see what values your application is storing at different levels. It's an invaluable tool for determining why your code is doing what it's doing.



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.



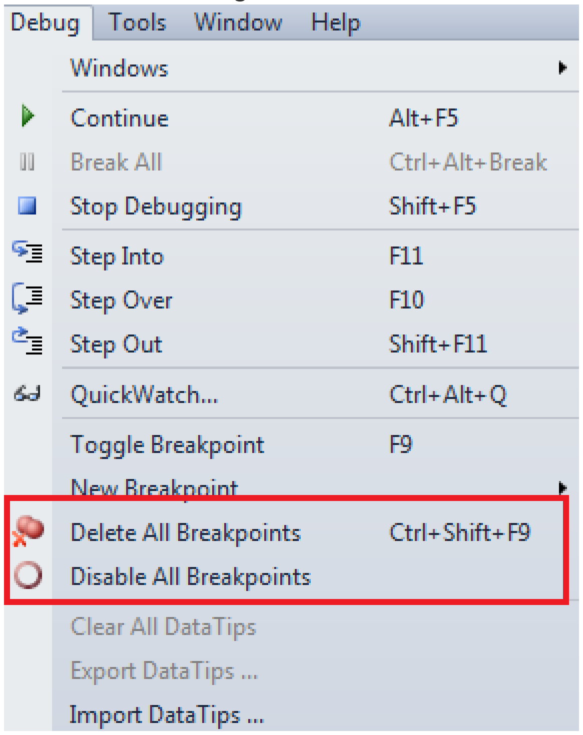
Immediate Window

```
@StartNumber
4
@StartNumber * 50
200
```

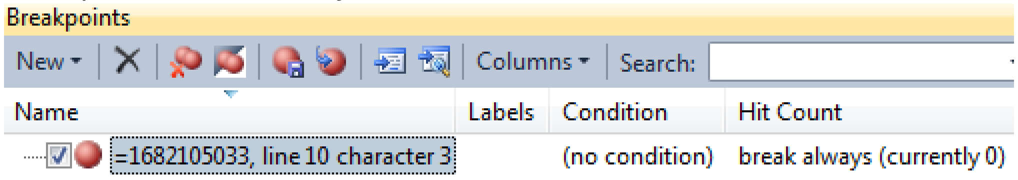
Breakpoints in SSMS : There are 2 ways to set a breakpoint in SSMS.
1. By clicking on the grey margin on the left hand side in SSMS (to remove click again)
2. By pressing F9 (to remove press F9 again)

Enable, Disable or Delete all breakpoints : There are 2 ways to Enable, Disable or Delete all breakpoints

1. From the Debug menu



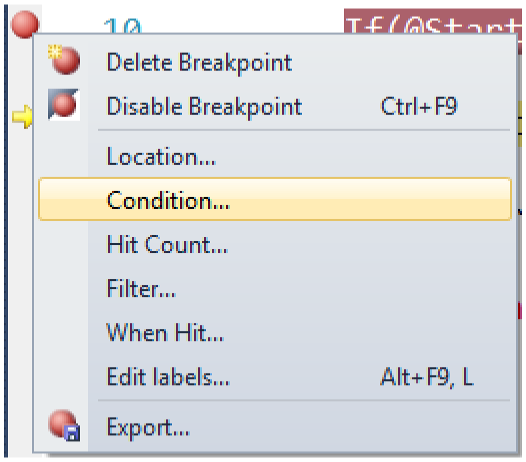
2. From the Breakpoints window. To view Breakpoints window select Debug => Windows => Breakpoints or use the keyboard shortcut ALT + CTRL + B



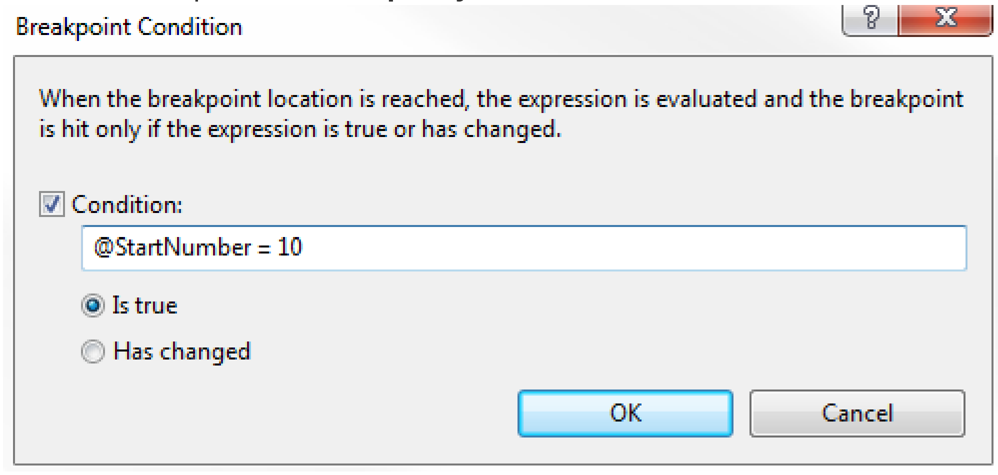
Name	Labels	Condition	Hit Count
=1682105033, line 10 character 3		(no condition)	break always (currently 0)

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**


Breakpoint 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

Part 108 - Over clause in SQL Server

In this video we will discuss the power and use of Over clause in SQL Server.

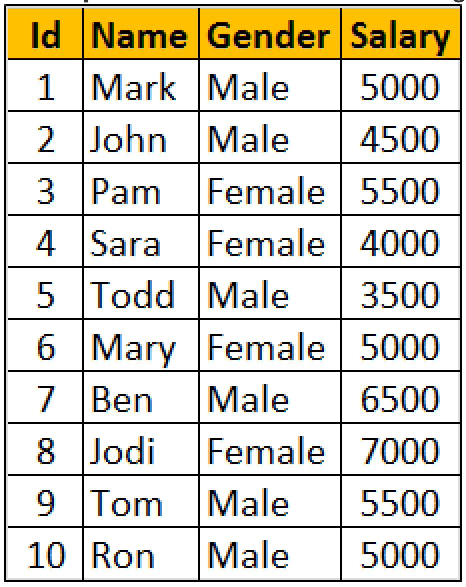
The **OVER** clause combined with **PARTITION BY** is used to break up data into partitions.
Syntax : function (...) OVER (PARTITION BY col1, Col2, ...)

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.

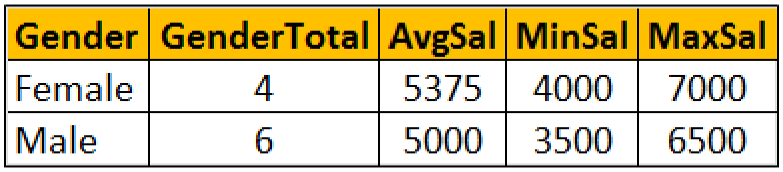


Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

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.

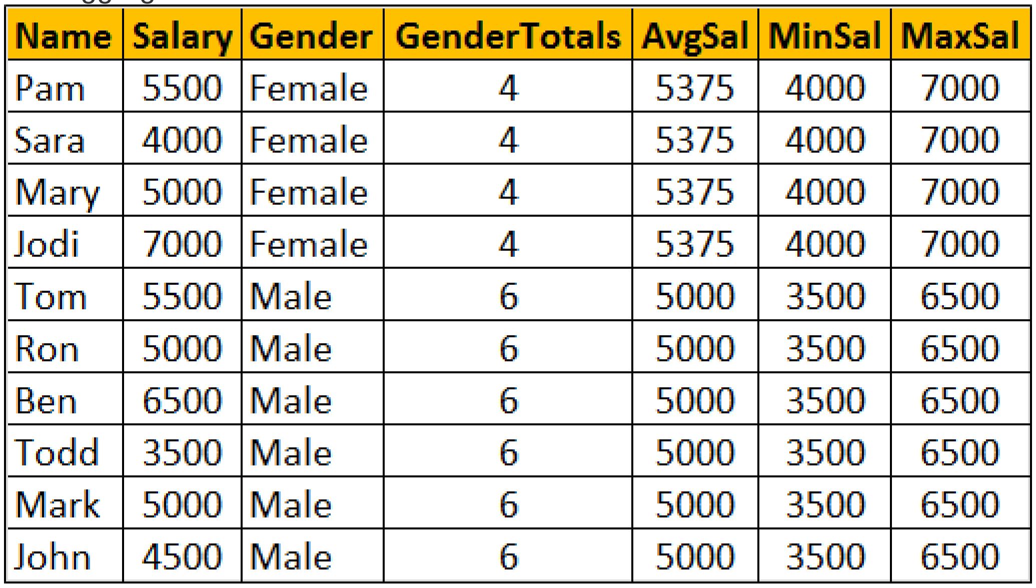


Gender	GenderTotal	AvgSal	MinSal	MaxSal
Female	4	5375	4000	7000
Male	6	5000	3500	6500

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
```

Part 109 - Row_Number function in SQL Server

In this video we will discuss Row_Number function in SQL Server. This is continuation toPart 108. Please watch Part 108 from SQL Server tutorial before proceeding.

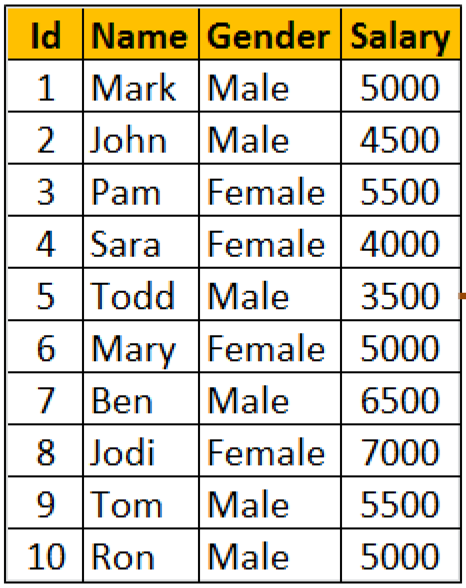
Row_Number function

- 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

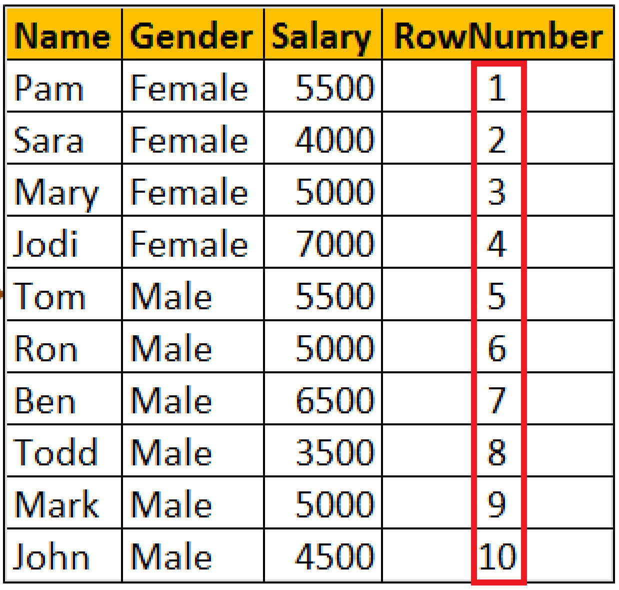
Syntax : ROW_NUMBER() OVER (ORDER BY Col1, Col2)

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

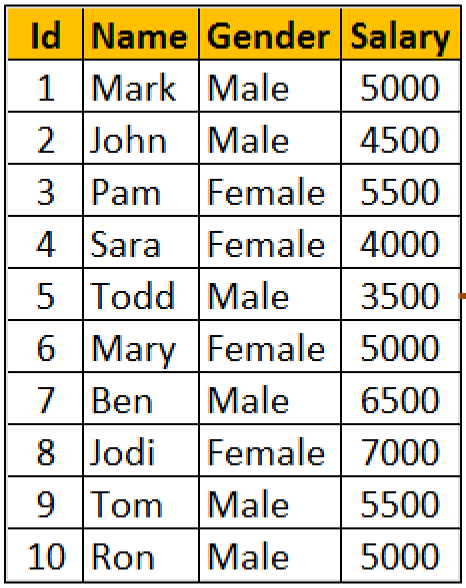


Name	Gender	Salary	RowNumber
Pam	Female	5500	1
Sara	Female	4000	2
Mary	Female	5000	3
Jodi	Female	7000	4
Tom	Male	5500	5
Ron	Male	5000	6
Ben	Male	6500	7
Todd	Male	3500	8
Mark	Male	5000	9
John	Male	4500	10

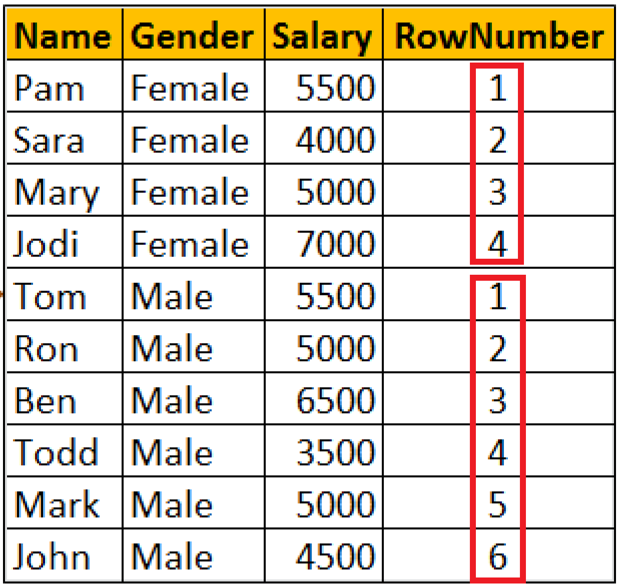
Please note : If ORDER BY clause is not specified you will get the following error
The function 'ROW_NUMBER' must have an OVER clause with ORDER BY

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) ASRowNumber
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.

Part 110 - 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

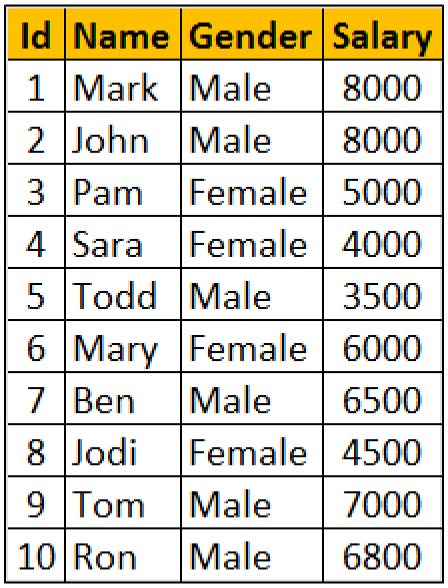
- Introduced in SQL Server 2005
- Returns a rank starting at 1 based on the ordering of rows imposed by the ORDER BY clause
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, rank is reset to 1 when the partition changes

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

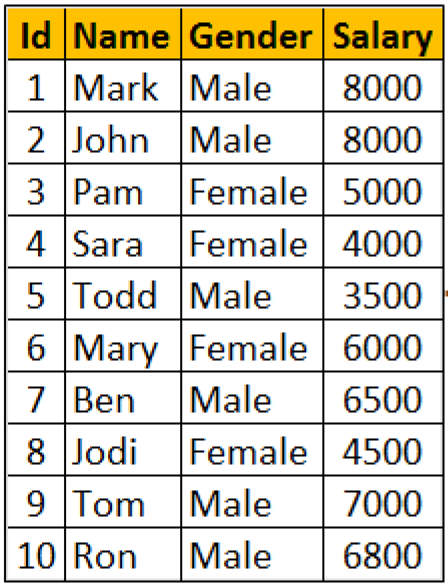
```
Create Table Employees
(
   Id int primary key,
   Name nvarchar(50),
   Gender nvarchar(10),
   Salary int
)
Go

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.

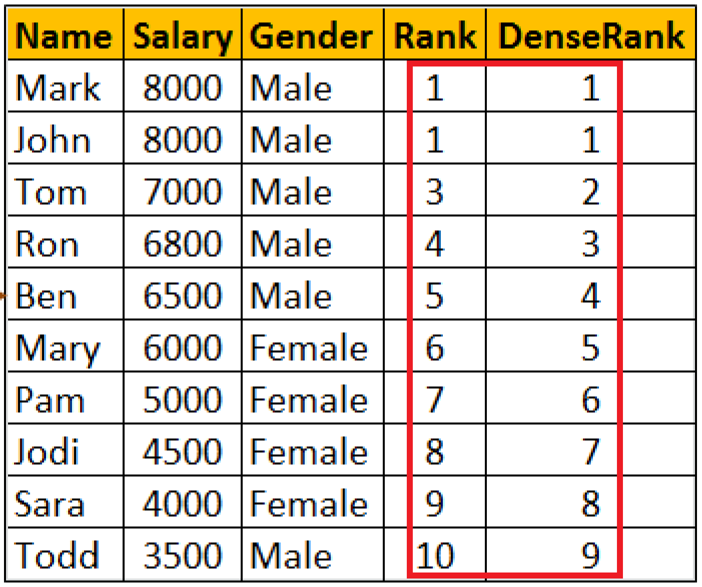
DENSE_RANK() on the other hand will not skip ranks if there is a tie. The first 2 rows get rank

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
```



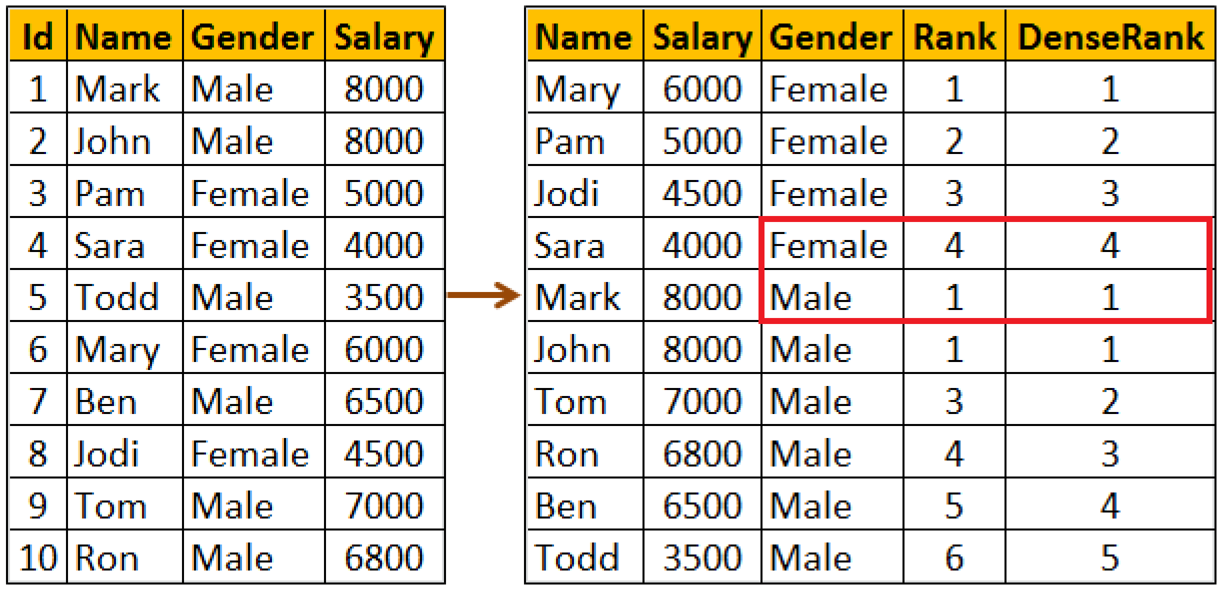
Id	Name	Gender	Salary
1	Mark	Male	8000
2	John	Male	8000
3	Pam	Female	5000
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	6000
7	Ben	Male	6500
8	Jodi	Female	4500
9	Tom	Male	7000
10	Ron	Male	6800



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) ASSalary_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'
```

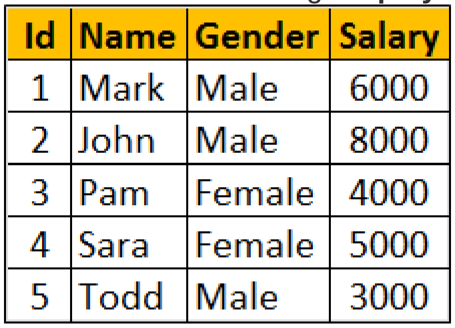
Part 111 - Difference between rank dense_rank and row_number in SQL

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



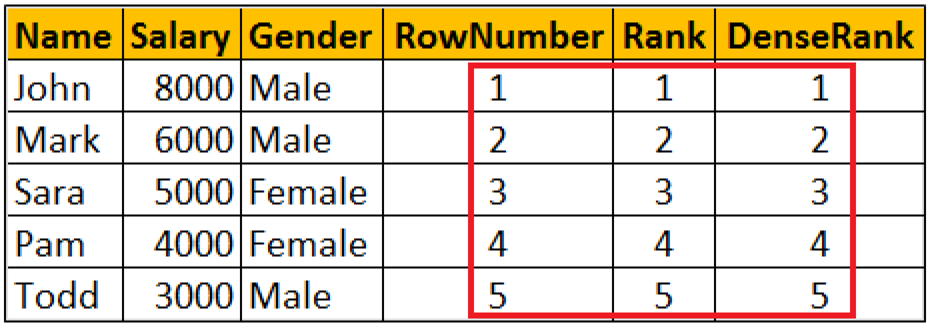
Id	Name	Gender	Salary
1	Mark	Male	6000
2	John	Male	8000
3	Pam	Female	4000
4	Sara	Female	5000
5	Todd	Male	3000

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.

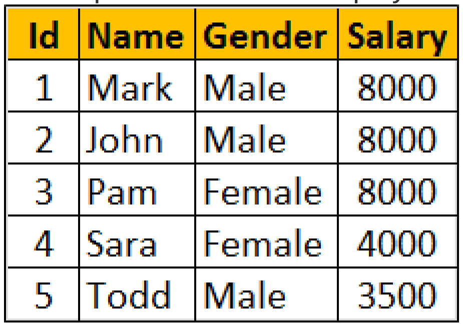
To do this
First delete existing data from the Employees table

```
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)
```

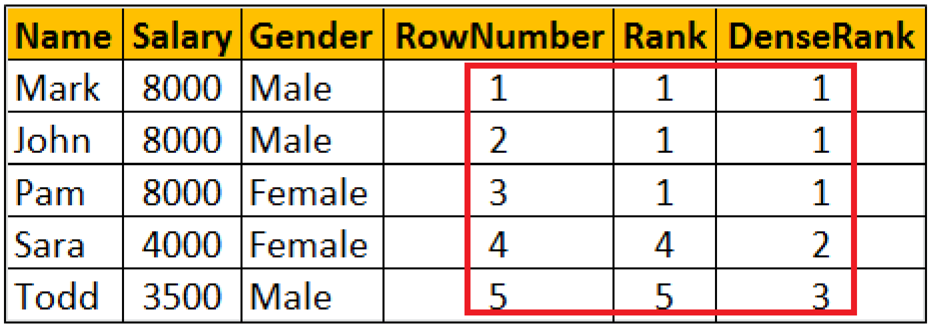
At this point data in the Employees table should be as shown below



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

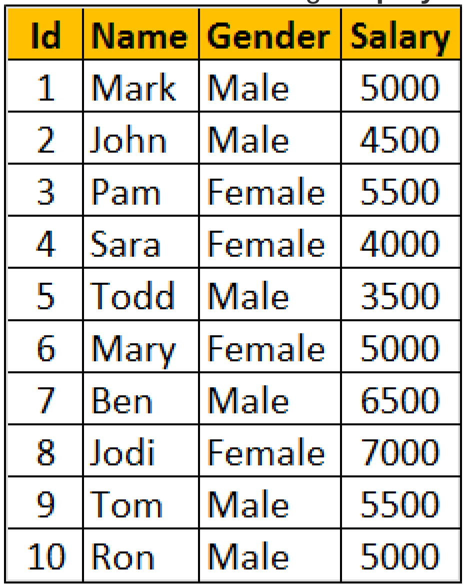
rows will have the rank it would have been assigned if there had been no duplicates. So RANK function skips rankings if there are duplicates.

- **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.

Part 112 - Calculate running total in SQL Server 2012

In this video we will discuss how to calculate running total in SQL Server 2012 and later versions.

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

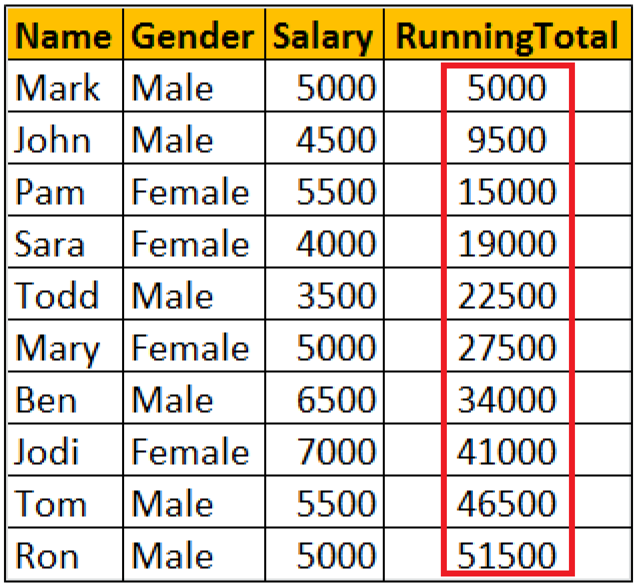
SQL Script to create Employees table

```
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
```

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.

Part 113 - NTILE function in SQL Server

In this video we will discuss **NTILE function in SQL Server**

NTILE function

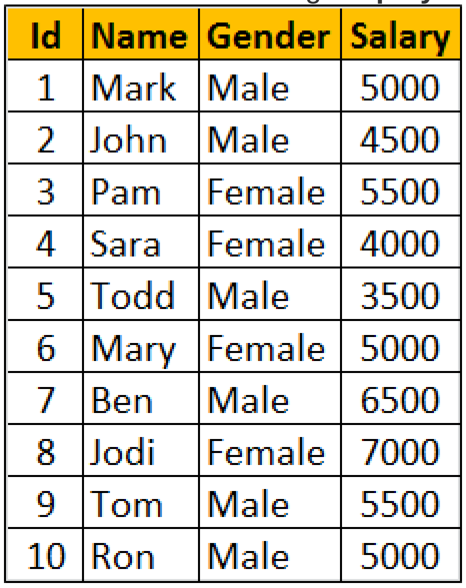
- Introduced in SQL Server 2005
- ORDER BY Clause is required
- PARTITION BY clause is optional
- Distributes the rows into a specified number of groups
- If the number of rows is not divisible by number of groups, you may have groups of two different sizes.
- Larger groups come before smaller groups

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.



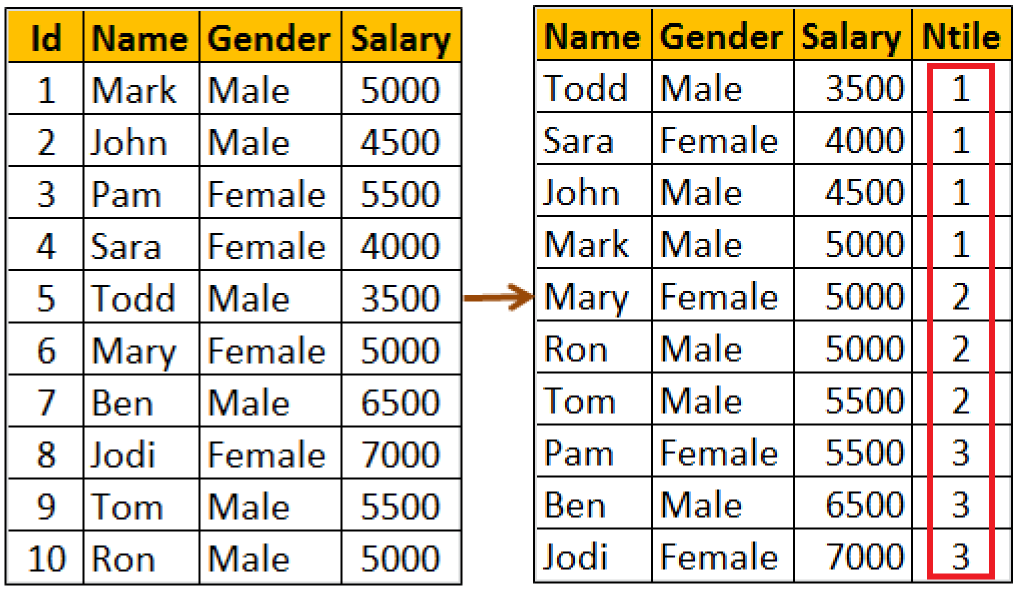
Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

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

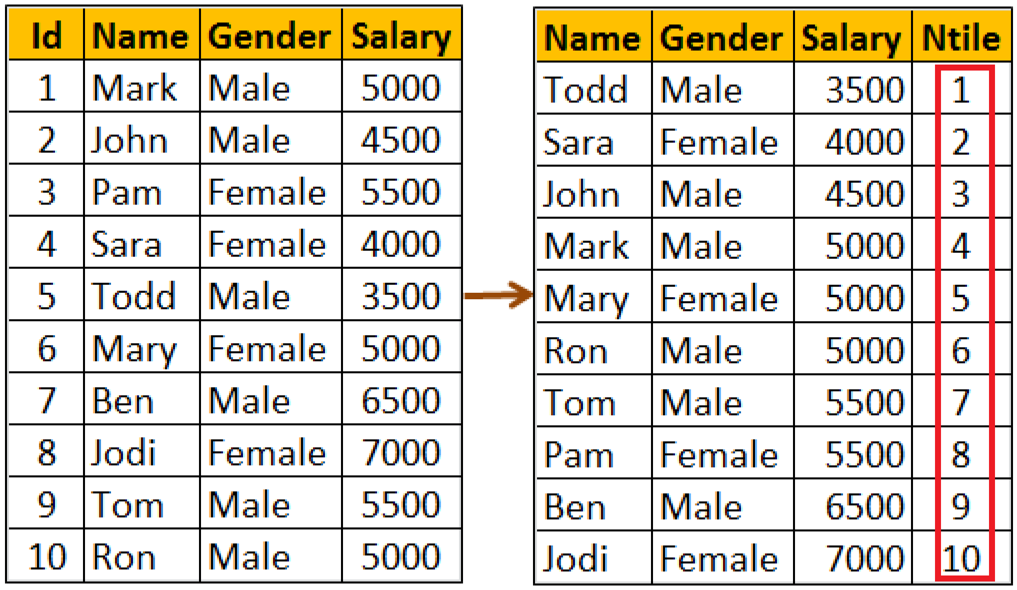
Name	Gender	Salary	Ntile
Todd	Male	3500	1
Sara	Female	4000	1
John	Male	4500	1
Mark	Male	5000	1
Mary	Female	5000	2
Ron	Male	5000	2
Tom	Male	5500	2
Pam	Female	5500	3
Ben	Male	6500	3
Jodi	Female	7000	3

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
```



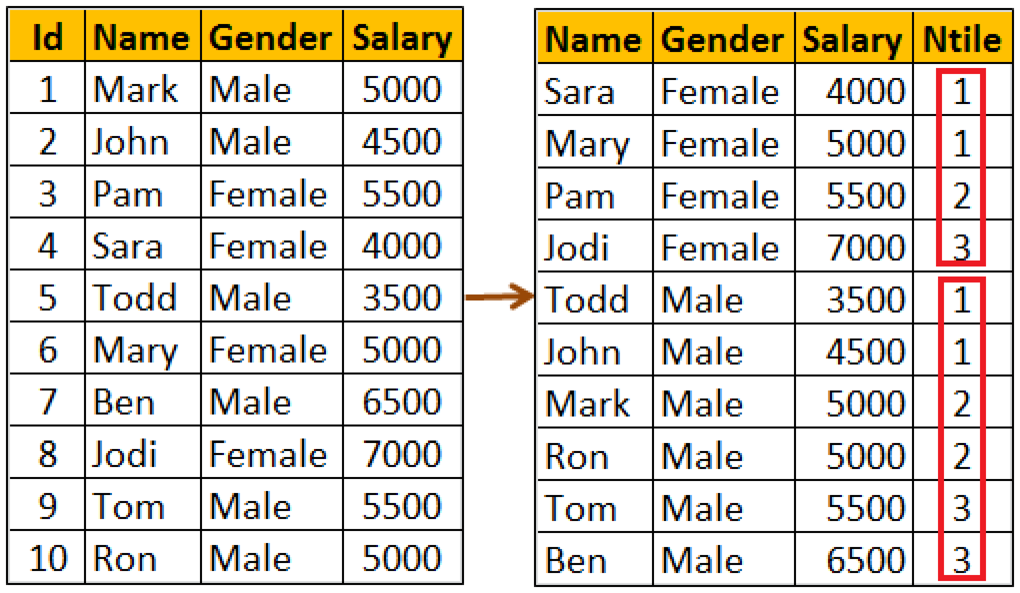
Id	Name	Gender	Salary
1	Mark	Male	5000
2	John	Male	4500
3	Pam	Female	5500
4	Sara	Female	4000
5	Todd	Male	3500
6	Mary	Female	5000
7	Ben	Male	6500
8	Jodi	Female	7000
9	Tom	Male	5500
10	Ron	Male	5000

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

Part 114 - Lead and Lag functions in SQL Server 2012

In this video we will discuss about Lead and Lag functions.

Lead and Lag functions

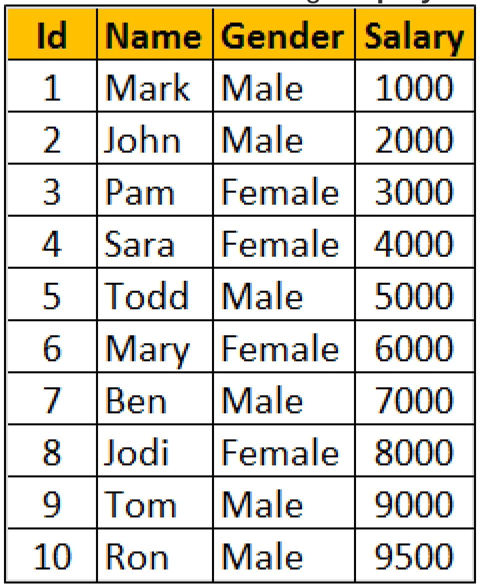
- Introduced in SQL Server 2012
- Lead function is used to access subsequent row data along with current row data
- Lag function is used to access previous row data along with current row data
- ORDER BY clause is required
- PARTITION BY clause is optional

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

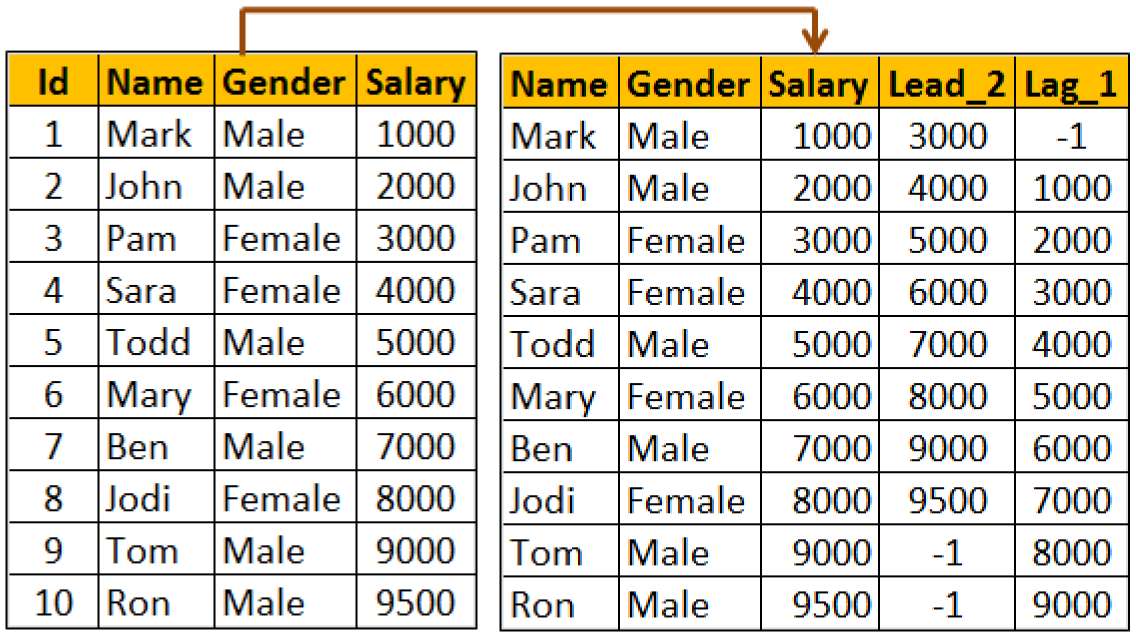
```
Create Table Employees
(
    Id int primary key,
    Name nvarchar(50),
    Gender nvarchar(10),
    Salary int
)
Go

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
```



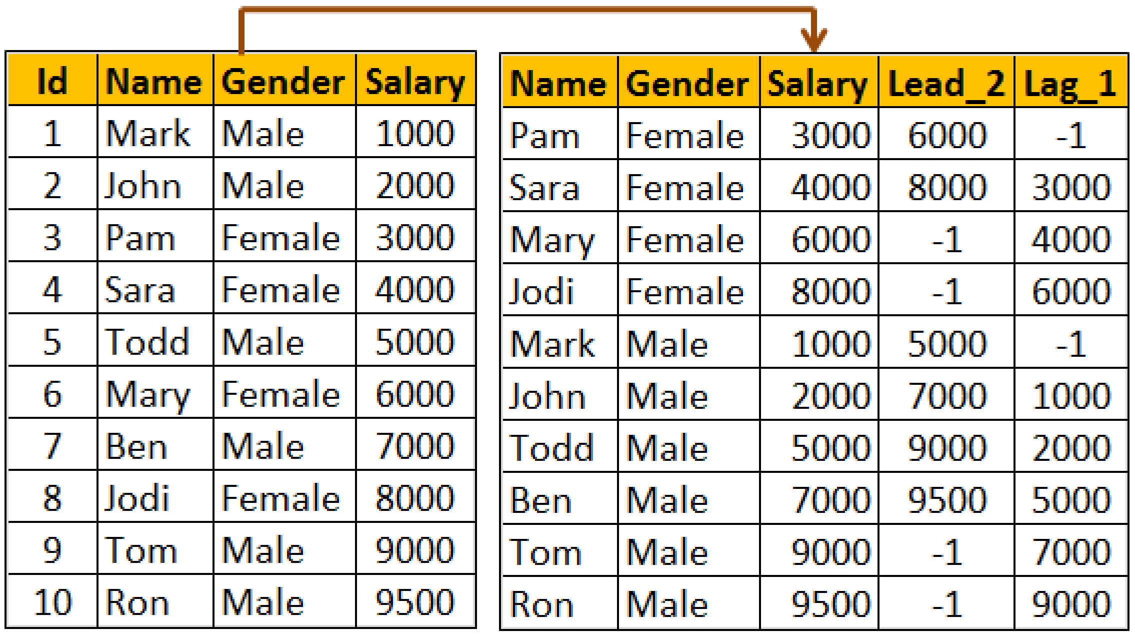
Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

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) ASLead_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

Part 115 - FIRST_VALUE function in SQL Server

In this video we will discuss FIRST_VALUE function in SQL Server

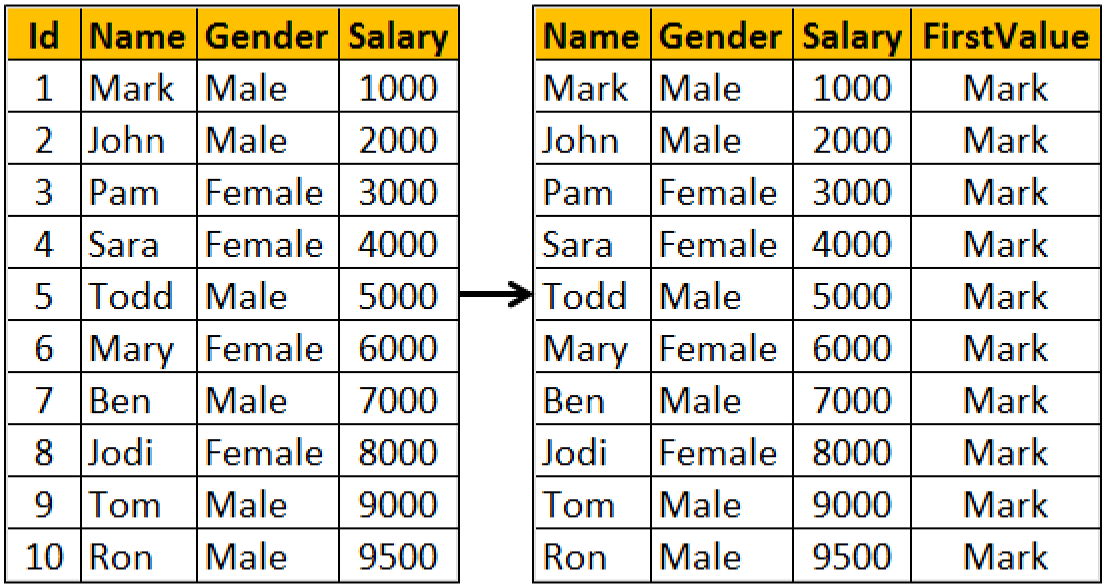
FIRST_VALUE function

- 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
```

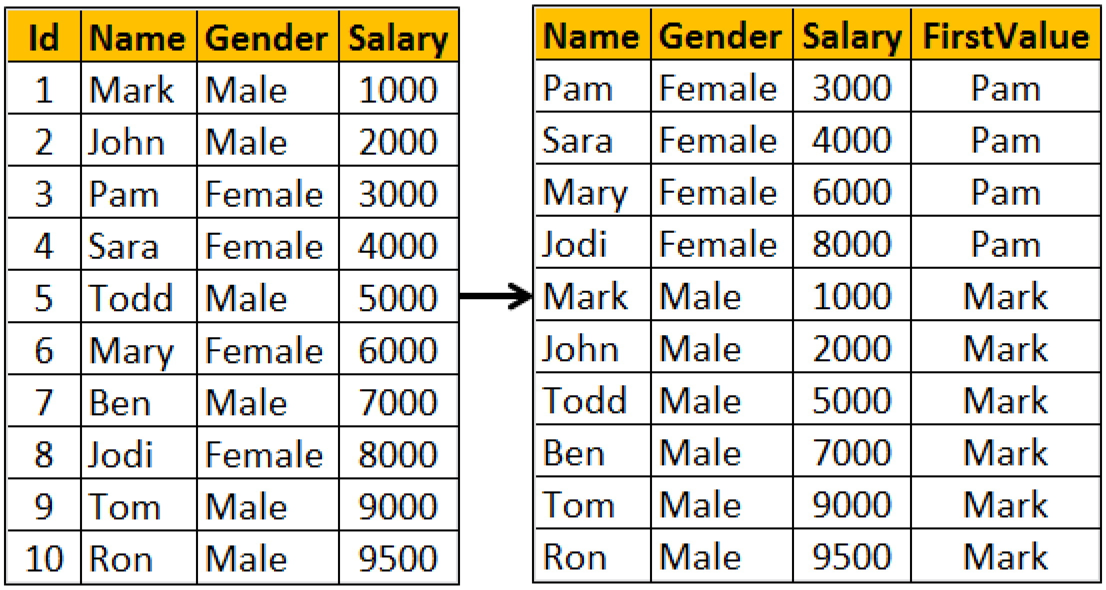


Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

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) ASFirstValue
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

Part 116 - 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...

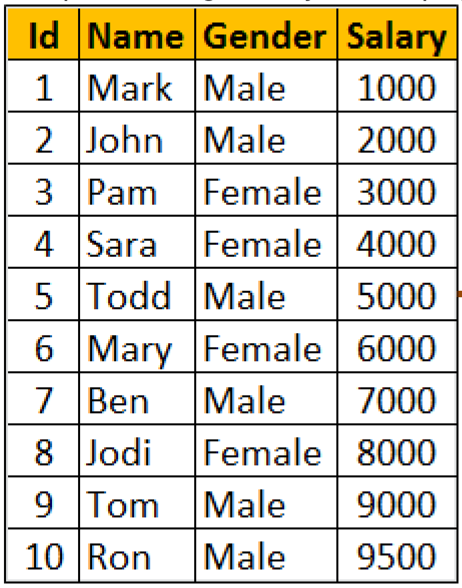
OVER Clause defines the partitioning and ordering of a rows (i.e a window) for the above functions to operate on. Hence these functions are called window functions. The OVER clause accepts the following three arguments to define a window for these functions to operate on.

- **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.

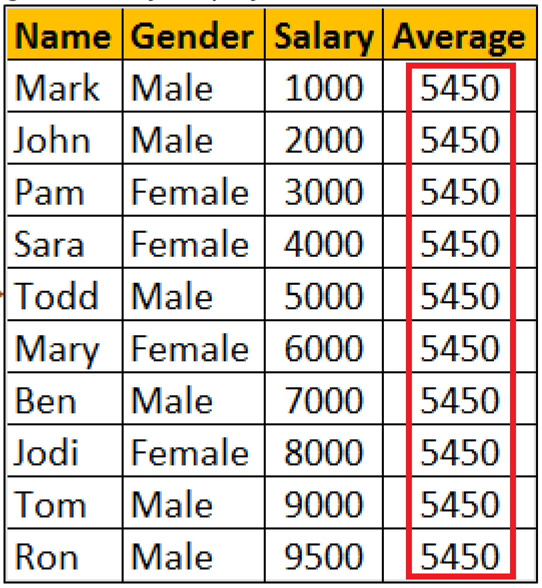
The default for **ROWS** or **RANGE** clause is
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

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

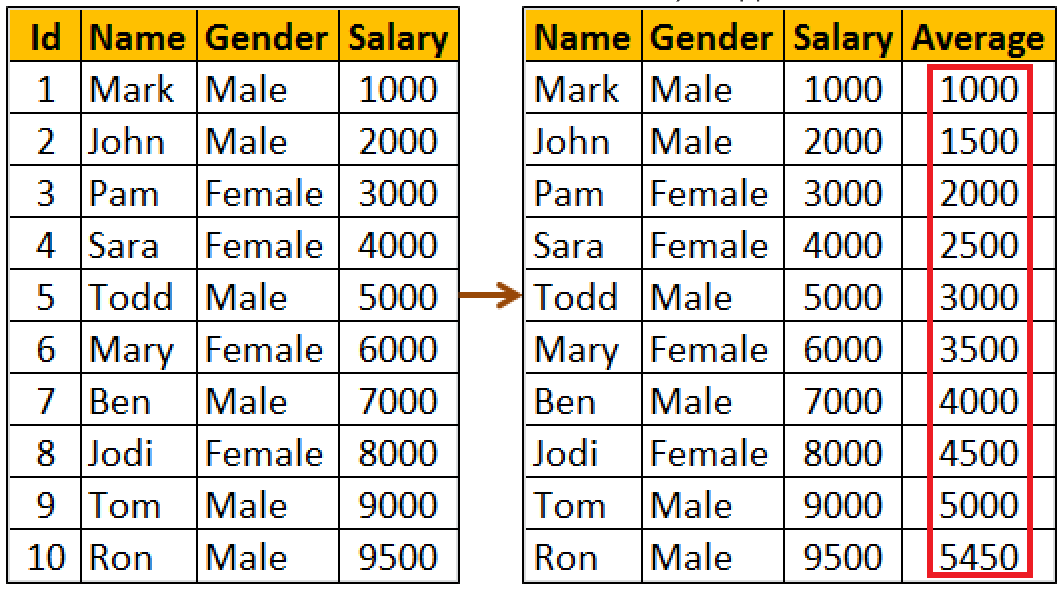


Name	Gender	Salary	Average
Mark	Male	1000	5450
John	Male	2000	5450
Pam	Female	3000	5450
Sara	Female	4000	5450
Todd	Male	5000	5450
Mary	Female	6000	5450
Ben	Male	7000	5450
Jodi	Female	8000	5450
Tom	Male	9000	5450
Ron	Male	9500	5450

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.

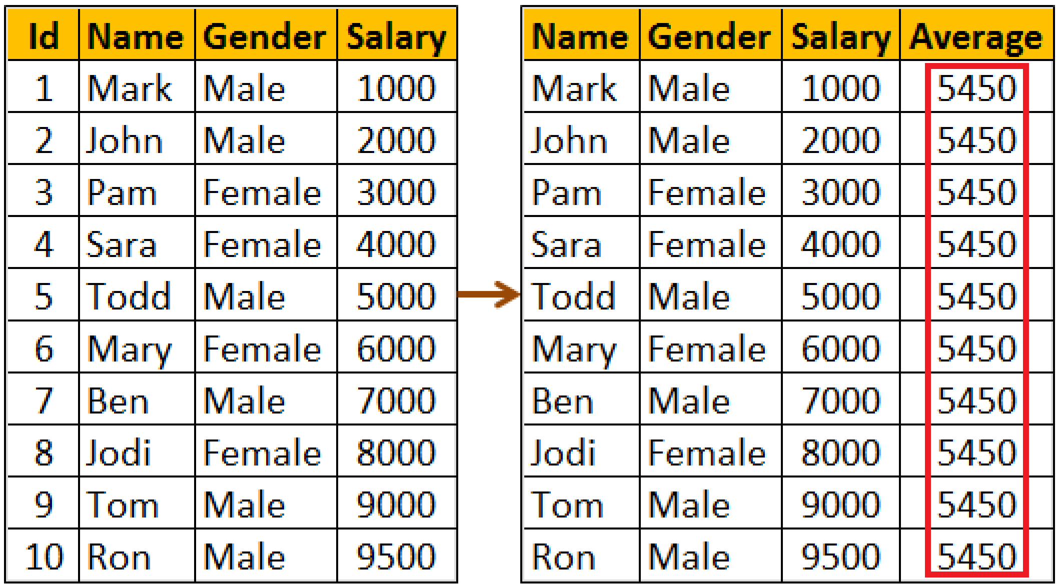


Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

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

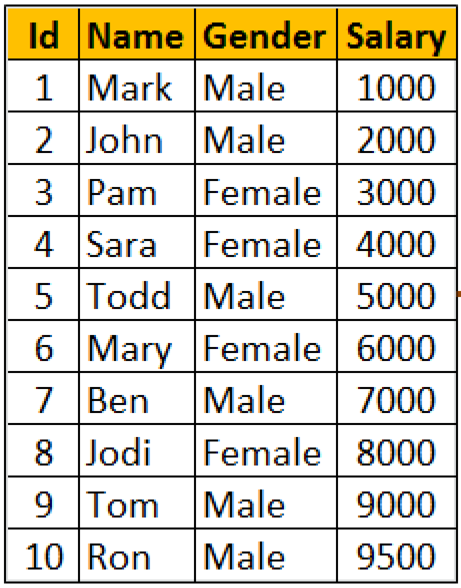
Name	Gender	Salary	Average
Mark	Male	1000	5450
John	Male	2000	5450
Pam	Female	3000	5450
Sara	Female	4000	5450
Todd	Male	5000	5450
Mary	Female	6000	5450
Ben	Male	7000	5450
Jodi	Female	8000	5450
Tom	Male	9000	5450
Ron	Male	9500	5450

The same result can also be achieved by using RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

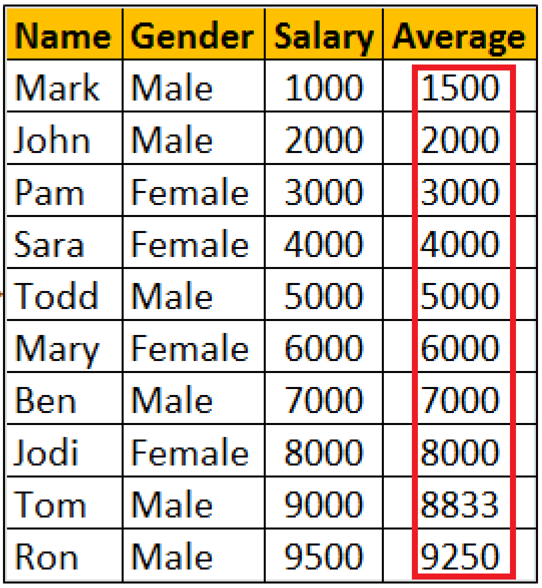
What is the difference between ROWS and RANGE
We will discuss this in a later video

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

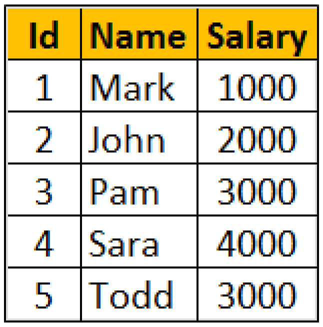


Name	Gender	Salary	Average
Mark	Male	1000	1500
John	Male	2000	2000
Pam	Female	3000	3000
Sara	Female	4000	4000
Todd	Male	5000	5000
Mary	Female	6000	6000
Ben	Male	7000	7000
Jodi	Female	8000	8000
Tom	Male	9000	8833
Ron	Male	9500	9250

Part 117 - 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.



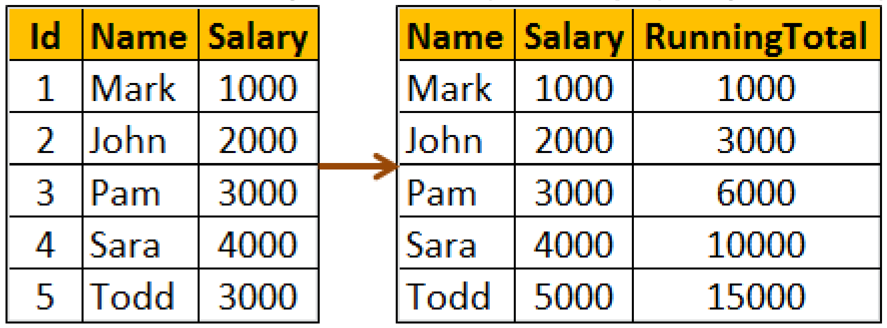
Id	Name	Salary
1	Mark	1000
2	John	2000
3	Pam	3000
4	Sara	4000
5	Todd	3000

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) ASRunningTotal
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) ASRunningTotal
FROM Employees
```

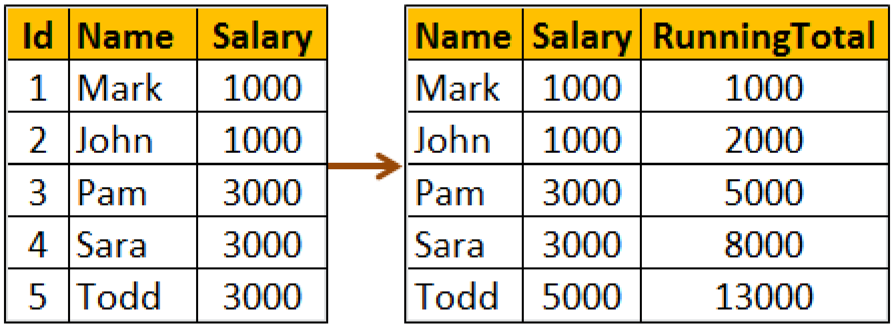
What is the difference between ROWS and RANGE
To understand the difference we need some duplicate values for the Salary column in the Employees table.

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) ASRunningTotal
FROM Employees
```



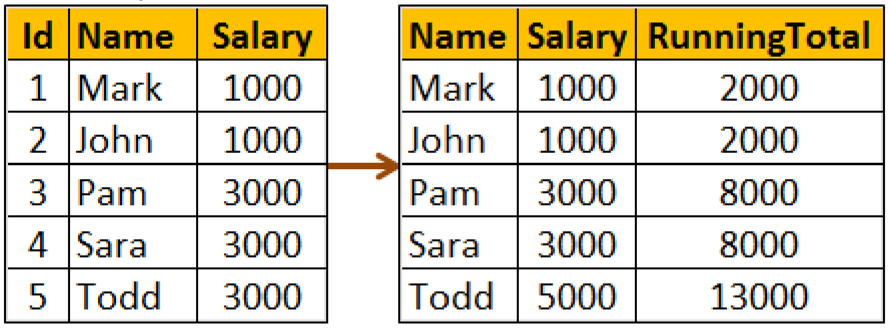
Id	Name	Salary
1	Mark	1000
2	John	1000
3	Pam	3000
4	Sara	3000
5	Todd	3000

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) ASRunningTotal
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

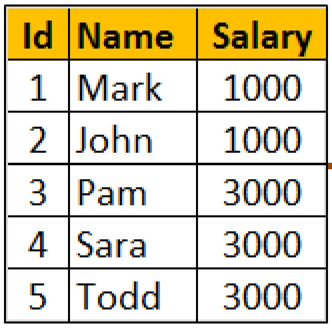
Name	Salary	RunningTotal
Mark	1000	2000
John	1000	2000
Pam	3000	8000
Sara	3000	8000
Todd	5000	13000

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.

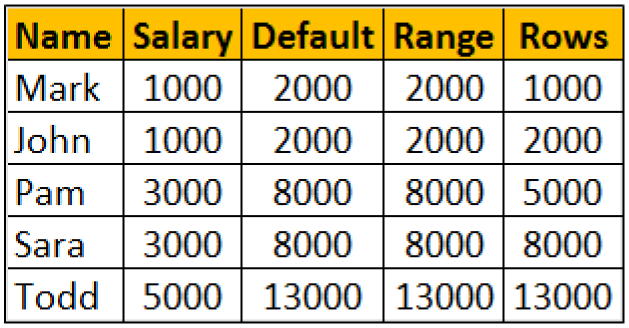
All together side by side. The following query shows how running total changes
1. When no value is specified for ROWS or RANGE clause
2. When RANGE clause is used explicitly with it's default value

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

Part 118 - LAST_VALUE function in SQL Server

In this video we will discuss LAST_VALUE function in SQL Server.

LAST_VALUE function

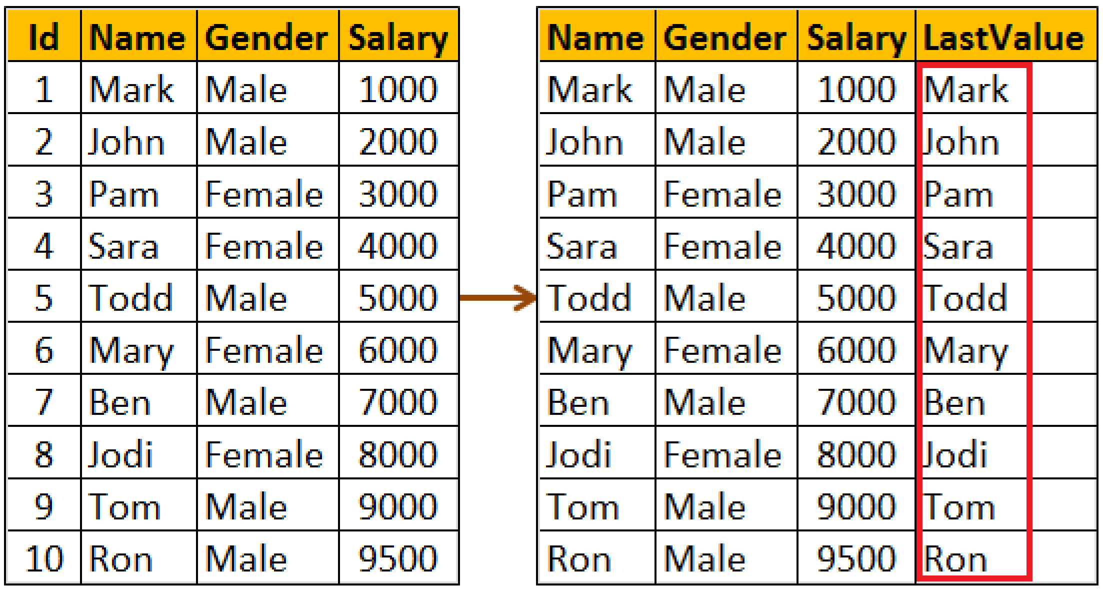
- Introduced in SQL Server 2012
- Retrieves the last value from the specified column
- ORDER BY clause is required
- PARTITION BY clause is optional
- ROWS or RANGE clause is optional, but for it to work correctly you may have to explicitly specify a value

Syntax : LAST_VALUE(Column_Name) OVER (ORDER BY Col1, Col2, ...)

LAST_VALUE function not working as expected : In the following example, LAST_VALUE function does not return the name of the highest paid employee. This is because we have not specified an explicit value for ROWS or RANGE clause. As a result it is using it's default value RANGE BETWEEN UNBOUNDED PRECEDING ANDCURRENT ROW.

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (ORDER BY Salary) AS LastValue
```

```
FROM Employees
```



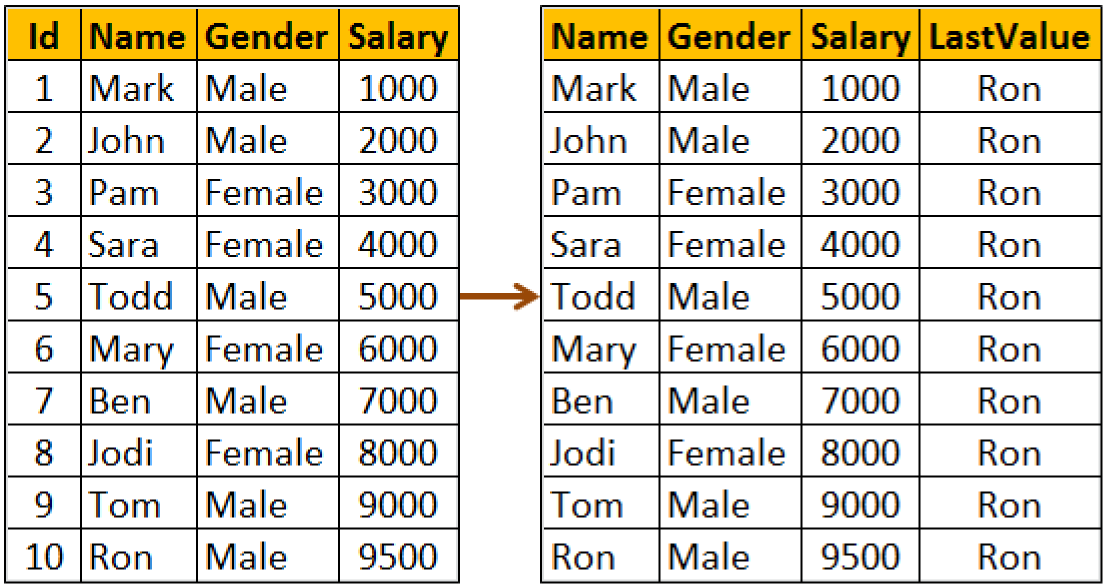
Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	LastValue
Mark	Male	1000	Mark
John	Male	2000	John
Pam	Female	3000	Pam
Sara	Female	4000	Sara
Todd	Male	5000	Todd
Mary	Female	6000	Mary
Ben	Male	7000	Ben
Jodi	Female	8000	Jodi
Tom	Male	9000	Tom
Ron	Male	9500	Ron

LAST_VALUE function working as expected : In the following example, LAST_VALUE function returns the name of the highest paid employee as expected. Notice we have set an explicit value for ROWS or RANGE clause
to ROWS BETWEEN UNBOUNDEDPRECEDING AND UNBOUNDED FOLLOWING

This tells the LAST_VALUE function that it's window starts at the first row and ends at the last row in the result set.

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (ORDER BY Salary ROWS BETWEEN UNBOUNDEDPRECED
ING AND UNBOUNDED FOLLOWING) AS LastValue
FROM Employees
```

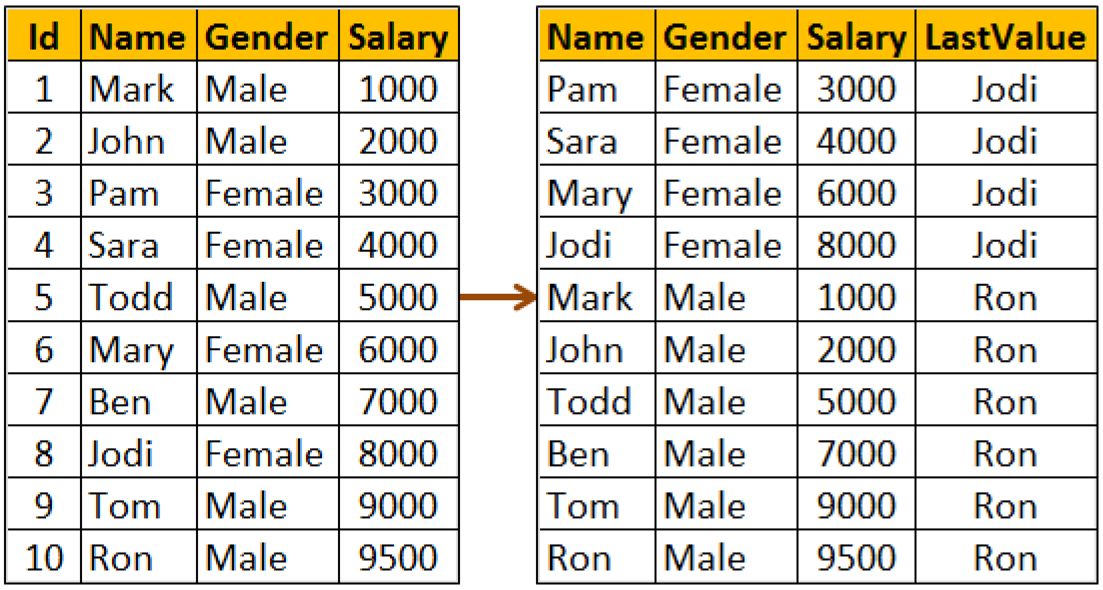


Id	Name	Gender	Salary
1	Mark	Male	1000
2	John	Male	2000
3	Pam	Female	3000
4	Sara	Female	4000
5	Todd	Male	5000
6	Mary	Female	6000
7	Ben	Male	7000
8	Jodi	Female	8000
9	Tom	Male	9000
10	Ron	Male	9500

Name	Gender	Salary	LastValue
Mark	Male	1000	Ron
John	Male	2000	Ron
Pam	Female	3000	Ron
Sara	Female	4000	Ron
Todd	Male	5000	Ron
Mary	Female	6000	Ron
Ben	Male	7000	Ron
Jodi	Female	8000	Ron
Tom	Male	9000	Ron
Ron	Male	9500	Ron

LAST_VALUE function example with partitions : In the following example, LAST_VALUE function returns the name of the highest paid employee from the respective partition.

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (PARTITION BY Gender ORDER BY Salary
   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)AS LastValue
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	LastValue
Pam	Female	3000	Jodi
Sara	Female	4000	Jodi
Mary	Female	6000	Jodi
Jodi	Female	8000	Jodi
Mark	Male	1000	Ron
John	Male	2000	Ron
Todd	Male	5000	Ron
Ben	Male	7000	Ron
Tom	Male	9000	Ron
Ron	Male	9500	Ron

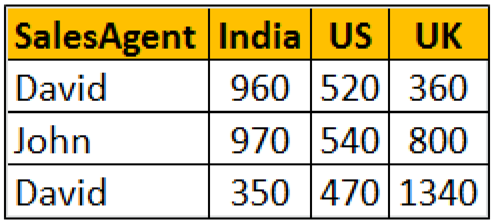
Part 119 - UNPIVOT in SQL Server

In this video we will discuss UNPIVOT operator in SQL Server.

PIVOT operator turns ROWS into COLUMNS, where as UNPIVOT turns COLUMNS into ROWS.

We discussed PIVOT operator in Part 54 of SQL Server tutorial. Please watch Part 54before proceeding.

Let us understand UNPIVOT with an example. We will use the following tblProductSales table in this demo.

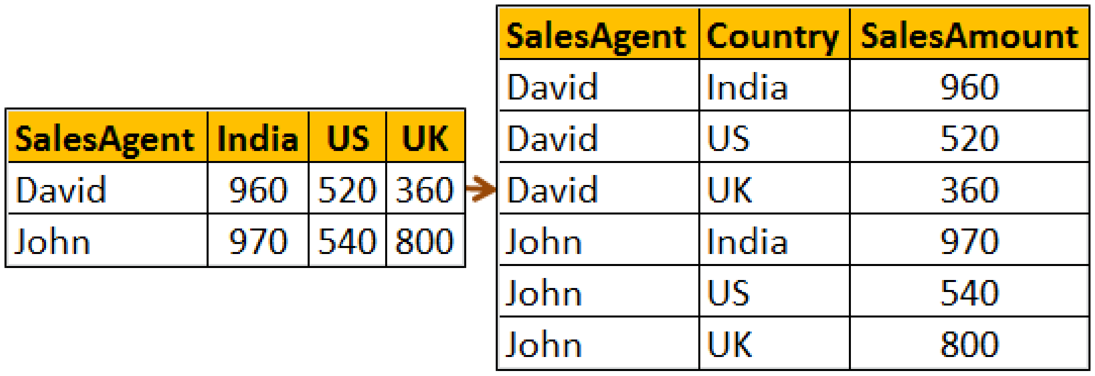


SalesAgent	India	US	UK
David	960	520	360
John	970	540	800
David	350	470	1340

SQL Script to create tblProductSales table

```
Create Table tblProductSales
(
 SalesAgent nvarchar(50),
 India int,
 US int,
 UK int
)
Go

Insert into tblProductSales values ('David', 960, 520, 360)
Insert into tblProductSales values ('John', 970, 540, 800)
Go
```

Write a query to turn COLUMNS into ROWS. The result of the query should be as shown below.



SalesAgent	India	US	UK
David	960	520	360
John	970	540	800

SalesAgent	Country	SalesAmount
David	India	960
David	US	520
David	UK	360
John	India	970
John	US	540
John	UK	800

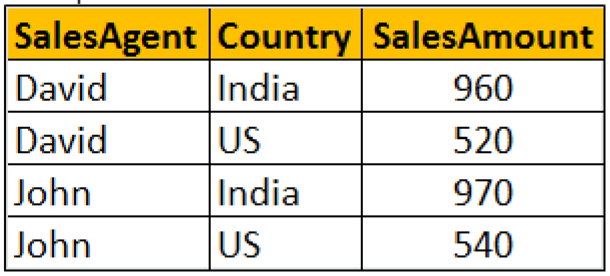
```
SELECT SalesAgent, Country, SalesAmount
FROM tblProductSales
UNPIVOT
(
      SalesAmount
      FOR Country IN (India, US ,UK)
) AS UnpivotExample
```

Part 120 - Reverse PIVOT table in SQL Server

In this video we will discuss if it's always possible to reverse what PIVOT operator has done using UNPIVOT operator.

Is it always possible to reverse what PIVOT operator has done using UNPIVOT operator.
No, not always. If the PIVOT operator has not aggregated the data, you can get your original data back using the UNPIVOT operator but not if the data is aggregated.

Let us understand this with an example. We will use the following table **tblProductSales**for the examples in this video.



SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540

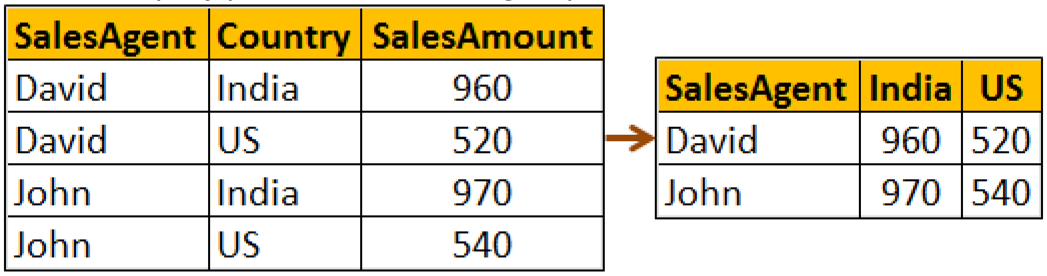
SQL Script to create tblProductSales table

```
Create Table tblProductSales
(
    SalesAgent nvarchar(10),
    Country nvarchar(10),
    SalesAmount int
)
Go

Insert into tblProductSales values('David','India',960)
Insert into tblProductSales values('David','US',520)
Insert into tblProductSales values('John','India',970)
Insert into tblProductSales values('John','US',540)
Go
```

Let's now use the PIVOT operator to turn ROWS into COLUMNS

```
SELECT SalesAgent, India, US
FROM tblProductSales
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable
```

The above query produces the following output



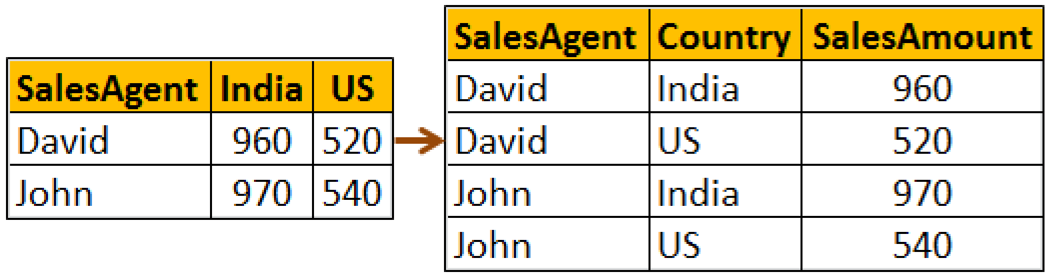
SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540

SalesAgent	India	US
David	960	520
John	970	540

Now let's use the UNPIVOT operator to reverse what PIVOT operator has done

```
SELECT SalesAgent, Country, SalesAmount
FROM
(SELECT SalesAgent, India, US
FROM tblProductSales
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable) P
UNPIVOT
(
    SalesAmount
    FOR Country IN (India, US)
) AS UnpivotTable
```

The above query reverses what PIVOT operator has done, and we get the original data back as shown below. We are able to get the original data back, because the SUM aggregate function that we used with the PIVOT operator did not perform any aggregation.



SalesAgent	India	US
David	960	520
John	970	540

SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540

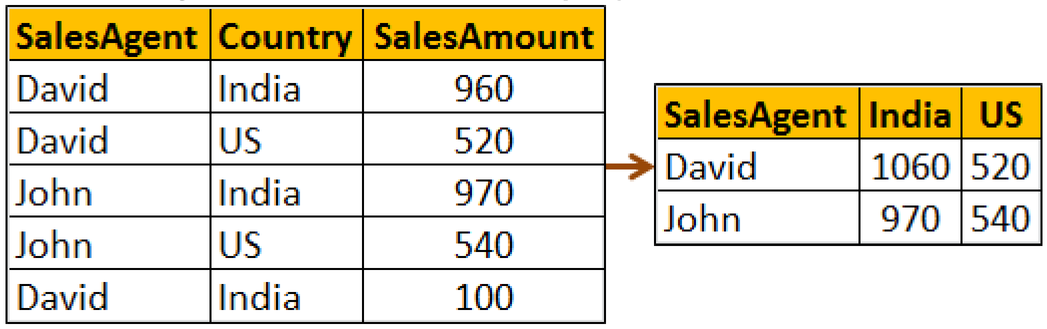
Now execute the following **INSERT** statement to insert a new row into **tblProductSales**table.

```
Insert into tblProductSales values('David','India',100)
```

With this new row in the table, if you execute the following **PIVOT** query data will be aggregated

```
SELECT SalesAgent, India, US
FROM tblProductSales
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable
```

The following is the result of the above query



SalesAgent	Country	SalesAmount
David	India	960
David	US	520
John	India	970
John	US	540
David	India	100

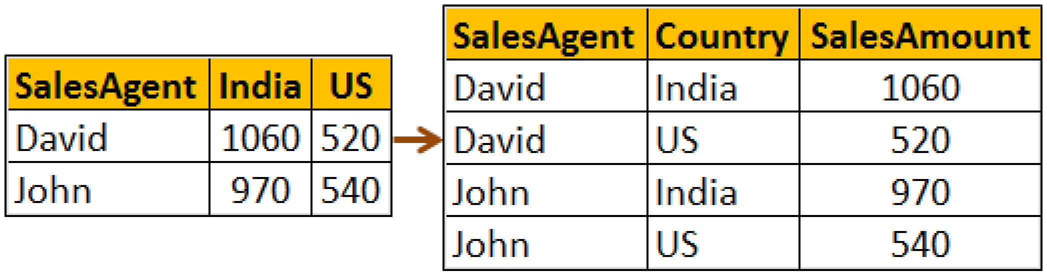
SalesAgent	India	US
David	1060	520
John	970	540

Now if we use UNPIVOT opertaor with the above query, we wouldn't get our orginial data back as the PIVOT operator has already aggrgated the data, and there is no way for SQL Server to know how to undo the aggregations.

```
SELECT SalesAgent, Country, SalesAmount
FROM
(SELECT SalesAgent, India, US
FROM tblProductSales
PIVOT
(
    SUM(SalesAmount)
    FOR Country IN (India, US)
) AS PivotTable) P
```

```
UNPIVOT
(
    SalesAmount
    FOR Country IN (India, US)

) AS UnpivotTable
```

Notice that for SalesAgent - David and Country - India we get only one row. In the original table we had 2 rows for the same combination.



SalesAgent	India	US
David	1060	520
John	970	540

SalesAgent	Country	SalesAmount
David	India	1060
David	US	520
John	India	970
John	US	540

Part 121 - Choose function in SQL Server

In this video we will discuss **Choose function in SQL Server**

Choose function

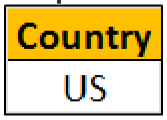
- Introduced in SQL Server 2012
- Returns the item at the specified index from the list of available values
- The index position starts at 1 and NOT 0 (ZERO)

Syntax : CHOOSE(index, val_1, val_2, ...)

Example : Returns the item at index position 2

```
SELECT CHOOSE(2, 'India','US', 'UK') AS Country
```

Output :



Country
US

Example : Using CHOOSE() function with table data. We will use the following**Employees** table for this example.

Id	Name	DateOfBirth
1	Mark	1980-01-11
2	John	1981-12-12
3	Amy	1979-11-21
4	Ben	1978-05-14
5	Sara	1970-03-17
6	David	1978-04-05

SQL Script to create Employees table

```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    DateOfBirth date
)
Go

Insert into Employees values ('Mark', '01/11/1980')
Insert into Employees values ('John', '12/12/1981')
Insert into Employees values ('Amy', '11/21/1979')
Insert into Employees values ('Ben', '05/14/1978')
Insert into Employees values ('Sara', '03/17/1970')
Insert into Employees values ('David', '04/05/1978')
Go
```

We want to display Month name along with employee Name and Date of Birth.

Name	DateOfBirth	MONTH
Mark	1980-01-11	JAN
John	1981-12-12	DEC
Amy	1979-11-21	NOV
Ben	1978-05-14	MAY
Sara	1970-03-17	MAR
David	1978-04-05	APR

Using CASE statement in SQL Server

```
SELECT Name, DateOfBirth,
      CASE DATEPART(MM, DateOfBirth)
```

```
            WHEN 1 THEN 'JAN'
            WHEN 2 THEN 'FEB'
            WHEN 3 THEN 'MAR'
            WHEN 4 THEN 'APR'
            WHEN 5 THEN 'MAY'
            WHEN 6 THEN 'JUN'
            WHEN 7 THEN 'JUL'
            WHEN 8 THEN 'AUG'
            WHEN 9 THEN 'SEP'
            WHEN 10 THEN 'OCT'
            WHEN 11 THEN 'NOV'
            WHEN 12 THEN 'DEC'
        END
       AS [MONTH]
FROM Employees
```

Using CHOOSE function in SQL Server : The amount of code we have to write is lot less than using CASE statement.

```
SELECT Name, DateOfBirth,CHOOSE(DATEPART(MM, DateOfBirth),
       'JAN', 'FEB', 'MAR', 'APR', 'MAY', 'JUN', 'JUL', 'AUG',
       'SEP', 'OCT', 'NOV', 'DEC') AS [MONTH]
FROM Employees
```

Part 122 - IIF function in SQL Server

In this video we will discuss **IIF function in SQL Server**.

IIF function

- Introduced in SQL Server 2012
- Returns one of two the values, depending on whether the Boolean expression evaluates to true or false
- IIF is a shorthand way for writing a CASE expression

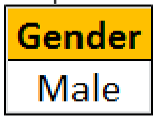
Syntax : IIF (boolean_expression, true_value, false_value)

Example : Returns Male as the boolean expression evaluates to TRUE

```
DECLARE @Gender INT
```

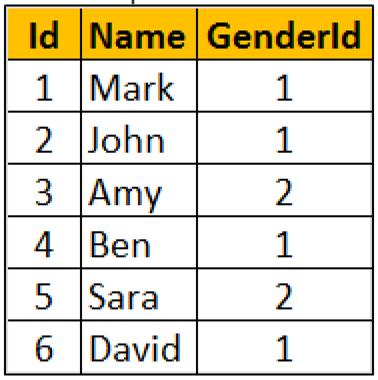
```
SET @Gender = 1
SELECT IIF( @Gender = 1, 'Male', 'Femlae') AS Gender
```

Output :



Gender
Male

Example : Using IIF() function with table data. We will use the following **Employees** table for this example.

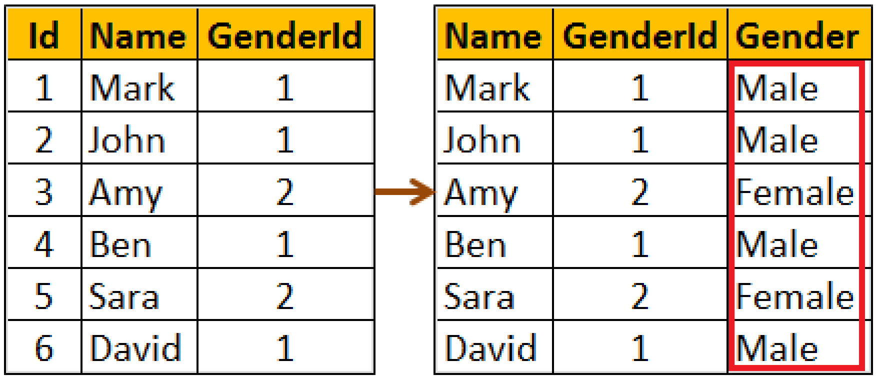


Id	Name	GenderId
1	Mark	1
2	John	1
3	Amy	2
4	Ben	1
5	Sara	2
6	David	1

SQL Script to create Employees table

```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    GenderId int
)
Go

Insert into Employees values ('Mark', 1)
Insert into Employees values ('John', 1)
Insert into Employees values ('Amy', 2)
Insert into Employees values ('Ben', 1)
Insert into Employees values ('Sara', 2)
Insert into Employees values ('David', 1)
Go
```

Write a query to display **Gender** along with **employee Name** and **GenderId**. We can achieve this either by using **CASE** or **IIF**.



Id	Name	GenderId
1	Mark	1
2	John	1
3	Amy	2
4	Ben	1
5	Sara	2
6	David	1

Name	GenderId	Gender
Mark	1	Male
John	1	Male
Amy	2	Female
Ben	1	Male
Sara	2	Female
David	1	Male

Using CASE statement

```
SELECT Name, GenderId,
       CASE WHEN GenderId = 1
                 THEN 'Male'
                 ELSE 'Female'
               END AS Gender
FROM Employees
```

Using IIF function

```
SELECT Name, GenderId, IIF(GenderId = 1, 'Male', 'Female') AS Gender
FROM Employees
```

Part 123 - TRY_PARSE function in SQL Server 2012

In this video we will discuss

- TRY_PARSE function
- Difference between PARSE and TRY_PARSE functions

TRY_PARSE function

- Introduced in SQL Server 2012
- Converts a string to Date/Time or Numeric type
- Returns NULL if the provided string cannot be converted to the specified data type
- Requires .NET Framework Common Language Runtime (CLR)

Syntax : TRY_PARSE (string_value AS data_type)

Example : Convert string to INT. As the string can be converted to INT, the result will be 99 as expected.

```
SELECT TRY_PARSE('99' AS INT) AS Result
```

Output :

Result
99

Example : Convert string to INT. The string cannot be converted to INT, so TRY_PARSE returns NULL

```
SELECT TRY_PARSE('ABC' AS INT) AS Result
```

Output :

Result
NULL

Use **CASE** statement or **IIF** function to provide a meaningful error message instead of NULL when the conversion fails.

Example : Using CASE statement to provide a meaningful error message when the conversion fails.

```
SELECT
CASE WHEN TRY_PARSE('ABC' AS INT) IS NULL
         THEN 'Conversion Failed'
         ELSE 'Conversion Successful'
END AS Result
```

Output : As the conversion fails, you will now get a message 'Conversion Failed' instead of NULL

Result
Conversion Failed

Example : Using IIF function to provide a meaningful error message when the conversion fails.

```
SELECT IIF(TRY_PARSE('ABC' AS INT) IS NULL, 'Conversion Failed',
              'Conversion Successful') AS Result
```

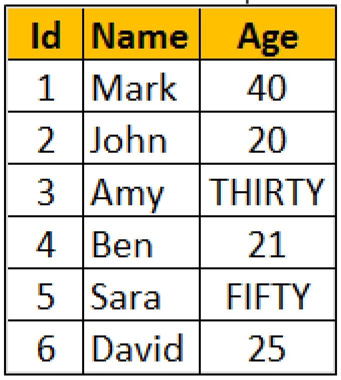
What is the difference between PARSE and TRY_PARSE
PARSE will result in an error if the conversion fails, where as TRY_PARSE will return NULL instead of an error.

Since ABC cannot be converted to INT, PARSE will return an error
```
SELECT PARSE('ABC' AS INT) AS Result
```

Since ABC cannot be converted to INT, TRY_PARSE will return NULL instead of an error

```
SELECT TRY_PARSE('ABC' AS INT) AS Result
```

Example : Using TRY_PARSE() function with table data. We will use the following Employees table for this example.



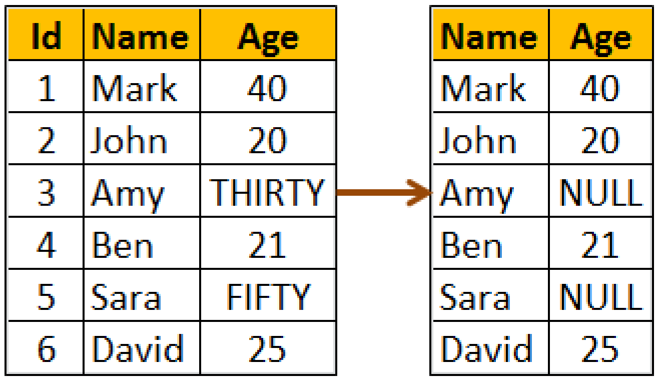
Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

SQL Script to create Employees table

```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    Age nvarchar(10)
)
Go

Insert into Employees values ('Mark', '40')
Insert into Employees values ('John', '20')
Insert into Employees values ('Amy', 'THIRTY')
Insert into Employees values ('Ben', '21')
Insert into Employees values ('Sara', 'FIFTY')
Insert into Employees values ('David', '25')
Go
```

The data type of Age column is nvarchar. So string values like (THIRTY, FIFTY) are also stored. Now, we want to write a query to convert the values in Age column to int and return along with the Employee name. Notice TRY_PARSE function returns NULL for the rows where age cannot be converted to INT.

```
SELECT Name, TRY_PARSE(Age AS INT) AS Age
FROM Employees
```



Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

Name	Age
Mark	40
John	20
Amy	NULL
Ben	21
Sara	NULL
David	25

If you use PARSE instead of TRY_PARSE, the query fails with an error.

```
SELECT Name, PARSE(Age AS INT) AS Age
FROM Employees
```

The above query returns the following error
Error converting string value 'THIRTY' into data type int using culture

Part 124 - TRY_CONVERT function in SQL Server 2012

In this video we will discuss

- TRY_CONVERT function
- Difference between CONVERT and TRY_CONVERT functions
- Difference between TRY_PARSE and TRY_CONVERT functions

TRY_CONVERT function

- Introduced in SQL Server 2012
- Converts a value to the specified data type
- Returns NULL if the provided value cannot be converted to the specified data type
- If you request a conversion that is explicitly not permitted, then TRY_CONVERT fails with an error

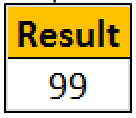
Syntax : TRY_CONVERT (data_type, value, [style])

Style parameter is optional. The range of acceptable values is determined by the target data_type. For the list of all possible values for style parameter, please visit the following MSDN article
https://msdn.microsoft.com/en-us/library/ms187928.aspx

Example : Convert string to INT. As the string can be converted to INT, the result will be 99 as expected.

```
SELECT TRY_CONVERT(INT, '99') AS Result
```

Output :

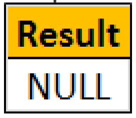


Result
99

Example : Convert string to INT. The string cannot be converted to INT, so TRY_CONVERT returns NULL

```
SELECT TRY_CONVERT(INT, 'ABC') AS Result
```

Output :



Result
NULL

Example : Converting an integer to XML is not explicitly permitted. so in this case TRY_CONVERT fails with an error

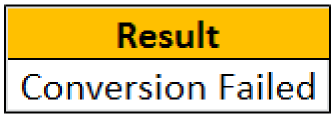
```
SELECT TRY_CONVERT(XML, 10) AS Result
```

If you want to provide a meaningful error message instead of NULL when the conversion fails, you can do so using CASE statement or IIF function.

Example : Using CASE statement to provide a meaningful error message when the conversion fails.

```
SELECT
CASE WHEN TRY_CONVERT(INT, 'ABC') IS NULL
         THEN 'Conversion Failed'
         ELSE 'Conversion Successful'
END AS Result
```

Output : As the conversion fails, you will now get a message 'Conversion Failed' instead of NULL



Result
Conversion Failed

Example : Using IIF function to provide a meaningful error message when the conversion fails.

```
SELECT IIF(TRY_CONVERT(INT, 'ABC') IS NULL, 'Conversion Failed',
            'Conversion Successful') AS Result
```

What is the difference between CONVERT and TRY_CONVERT
CONVERT will result in an error if the conversion fails, where as TRY_CONVERT will return NULL instead of an error.

Since ABC cannot be converted to INT, CONVERT will return an error
```
SELECT CONVERT(INT, 'ABC') AS Result
```

Since ABC cannot be converted to INT, TRY_CONVERT will return NULL instead of an error
```
SELECT TRY_CONVERT(INT, 'ABC') AS Result
```

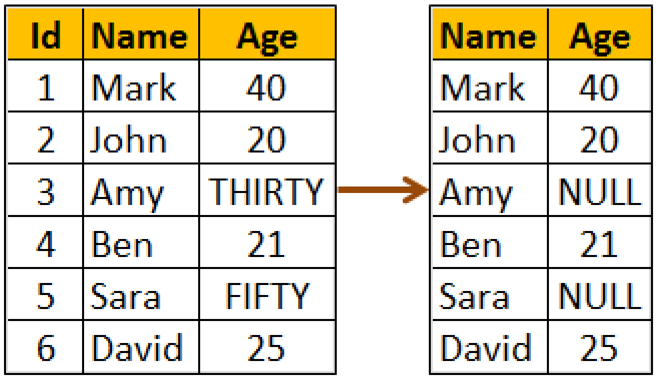
Example : Using TRY_CONVERT() function with table data. We will use the following Employees table for this example.

Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

SQL Script to create Employees table
```
Create table Employees
(
    Id int primary key identity,
    Name nvarchar(10),
    Age nvarchar(10)
)
Go

Insert into Employees values ('Mark', '40')
Insert into Employees values ('John', '20')
Insert into Employees values ('Amy', 'THIRTY')
Insert into Employees values ('Ben', '21')
Insert into Employees values ('Sara', 'FIFTY')
Insert into Employees values ('David', '25')
Go
```

The data type of Age column is nvarchar. So string values like (THIRTY, FIFTY) are also stored. Now, we want to write a query to convert the values in Age column to int and return along with the Employee name. Notice TRY_CONVERT function returns NULL for the rows where age cannot be converted to INT.

```
SELECT Name, TRY_CONVERT(INT, Age) AS Age
FROM Employees
```



Id	Name	Age
1	Mark	40
2	John	20
3	Amy	THIRTY
4	Ben	21
5	Sara	FIFTY
6	David	25

Name	Age
Mark	40
John	20
Amy	NULL
Ben	21
Sara	NULL
David	25

If you use CONVERT instead of TRY_CONVERT, the query fails with an error.

```
SELECT NAME, CONVERT(INT, Age) AS Age
FROM Employees
```

The above query returns the following error
Conversion failed when converting the nvarchar value 'THIRTY' to data type int.

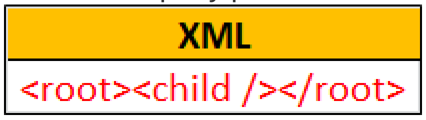
Difference between TRY_PARSE and TRY_CONVERT functions
TRY_PARSE can only be used for converting from string to date/time or number data types where as TRY_CONVERT can be used for any general type conversions.

For example, you can use TRY_CONVERT to convert a string to XML data type, where as you can do the same using TRY_PARSE

Converting a string to XML data type using TRY_CONVERT

```
SELECT TRY_CONVERT(XML, '<root><child/></root>') AS [XML]
```

The above query produces the following



XML
<root><child /></root>

Converting a string to XML data type using TRY_PARSE

```
SELECT TRY_PARSE('<root><child/></root>' AS XML) AS [XML]
```

The above query will result in the following error
Invalid data type xml in function TRY_PARSE

Another difference is TRY_PARSE relies on the presence of .the .NET Framework Common Language Runtime (CLR) where as TRY_CONVERT does not.

Part 125 - EOMONTH function in SQL Server 2012

In this video we will discuss **EOMONTH function in SQL Server 2012**

EOMONTH function

- Introduced in SQL Server 2012
- Returns the last day of the month of the specified date

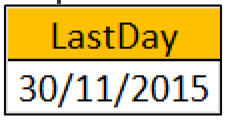
Syntax : EOMONTH (start_date [, month_to_add])

start_date : The date for which to return the last day of the month
month_to_add : Optional. Number of months to add to the start_date. EOMONTH adds the specified number of months to start_date, and then returns the last day of the month for the resulting date.

Example : Returns last day of the month November

```
SELECT EOMONTH('11/20/2015') AS LastDay
```

Output :

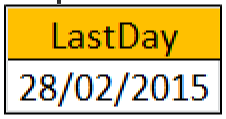


LastDay
30/11/2015

Example : Returns last day of the month of February from a NON-LEAP year

```
SELECT EOMONTH('2/20/2015') AS LastDay
```

Output :



LastDay
28/02/2015

Example : Returns last day of the month of February from a LEAP year

```
SELECT EOMONTH('2/20/2016') AS LastDay
```

Output :

LastDay
29/02/2016

month_to_add optional parameter can be used to add or subtract a specified number of months from the start_date, and then return the last day of the month from the resulting date.

The following example adds 2 months to the start_date and returns the last day of the month from the resulting date

```
SELECT EOMONTH('3/20/2016', 2) AS LastDay
```

Output :

LastDay
31/05/2016

The following example subtracts 1 month from the start_date and returns the last day of the month from the resulting date

```
SELECT EOMONTH('3/20/2016', -1) AS LastDay
```

Output :

LastDay
29/02/2016

Using **EOMONTH** function with table data. We will use the following **Employees** table for this example.

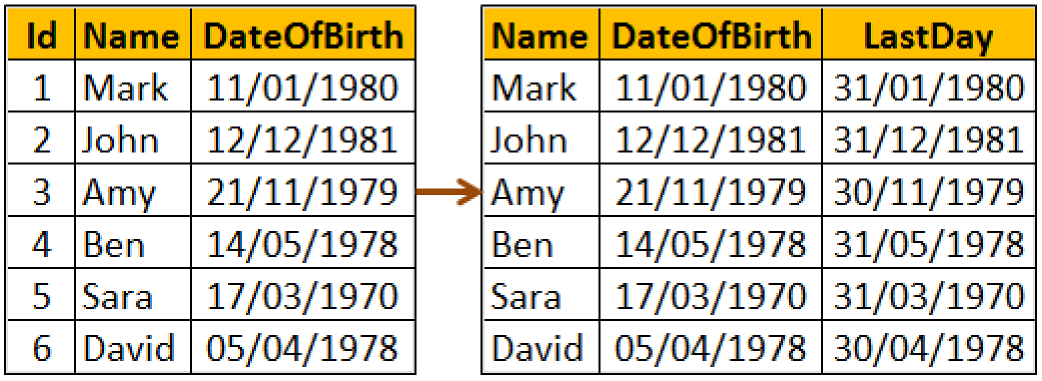
Id	Name	DateOfBirth
1	Mark	11/01/1980
2	John	12/12/1981
3	Amy	21/11/1979
4	Ben	14/05/1978
5	Sara	17/03/1970
6	David	05/04/1978

SQL Script to create Employees table

```
Create table Employees

(

   Id int primary key identity,
    Name nvarchar(10),
    DateOfBirth date
)
Go

Insert into Employees values ('Mark', '01/11/1980')
Insert into Employees values ('John', '12/12/1981')
Insert into Employees values ('Amy', '11/21/1979')
Insert into Employees values ('Ben', '05/14/1978')
Insert into Employees values ('Sara', '03/17/1970')
Insert into Employees values ('David', '04/05/1978')
Go
```

The following example returns the last day of the month from the DateOfBirth of every employee.

```
SELECT Name, DateOfBirth, EOMONTH(DateOfBirth) AS LastDay
FROM Employees
```

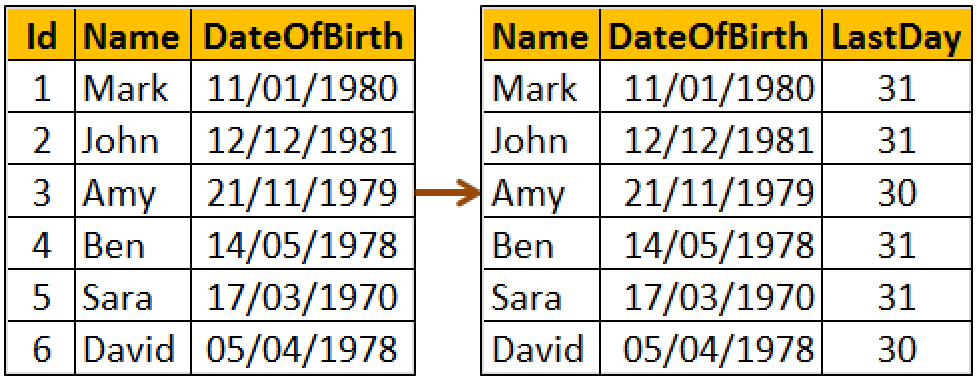


Id	Name	DateOfBirth
1	Mark	11/01/1980
2	John	12/12/1981
3	Amy	21/11/1979
4	Ben	14/05/1978
5	Sara	17/03/1970
6	David	05/04/1978

Name	DateOfBirth	LastDay
Mark	11/01/1980	31/01/1980
John	12/12/1981	31/12/1981
Amy	21/11/1979	30/11/1979
Ben	14/05/1978	31/05/1978
Sara	17/03/1970	31/03/1970
David	05/04/1978	30/04/1978

If you want just the last day instead of the full date, you can use **DATEPART** function

```
SELECT Name, DateOfBirth, DATEPART(DD,EOMONTH(DateOfBirth)) AS LastDay
FROM Employees
```



Id	Name	DateOfBirth
1	Mark	11/01/1980
2	John	12/12/1981
3	Amy	21/11/1979
4	Ben	14/05/1978
5	Sara	17/03/1970
6	David	05/04/1978

Name	DateOfBirth	LastDay
Mark	11/01/1980	31
John	12/12/1981	31
Amy	21/11/1979	30
Ben	14/05/1978	31
Sara	17/03/1970	31
David	05/04/1978	30

Part 126 - DATEFROMPARTS function in SQL Server

In this video we will discuss **DATEFROMPARTS function in SQL Server**

DATEFROMPARTS function

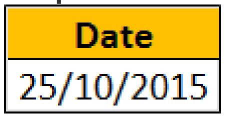
- Introduced in SQL Server 2012
- Returns a date value for the specified year, month, and day
- The data type of all the 3 parameters (year, month, and day) is integer
- If invalid argument values are specified, the function returns an error
- If any of the arguments are NULL, the function returns null

Syntax : DATEFROMPARTS (year, month, day)

Example : All the function arguments have valid values, so DATEFROMPARTS returns the expected date

```
SELECT DATEFROMPARTS ( 2015, 10, 25) AS [Date]
```

Output :



Date
25/10/2015

Example : Invalid value specified for month parameter, so the function returns an error

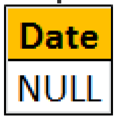
```
SELECT DATEFROMPARTS ( 2015, 15, 25) AS [Date]
```

Output : Cannot construct data type date, some of the arguments have values which are not valid.

Example : NULL specified for month parameter, so the function returns NULL.

```
SELECT DATEFROMPARTS ( 2015, NULL, 25) AS [Date]
```

Output :



```
Date
NULL
```

Other new date and time functions introduced in SQL Server 2012

- EOMONTH (Discussed in Part 125 of SQL Server tutorial)
- **DATETIMEFROMPARTS :** Returns DateTime
- **Syntax**

: DATETIMEFROMPARTS (year, month, day, hour, minute, seconds,milliseconds)

- **SMALLDATETIMEFROMPARTS :** Returns SmallDateTime
- **Syntax :** SMALLDATETIMEFROMPARTS (year, month, day, hour, minute)
- **We will discuss the following functions in a later video**
- TIMEFROMPARTS
- DATETIME2FROMPARTS
- DATETIMEOFFSETFROMPARTS

In our next video we will discuss the **difference between DateTime and SmallDateTime**.

Part 127 - Difference between DateTime and SmallDateTime in SQL Server

In this video we will discuss the **difference between DateTime and SmallDateTime in SQL Server**

The following **table summarizes the differences**

Attribute	SmallDateTime	DateTime
Date Range	January 1, 1900, through June 6, 2079	January 1, 1753, through December 31, 9999
Time Range	00:00:00 through 23:59:59	00:00:00 through 23:59:59.997
Accuracy	1 Minute	3.33 Milli-seconds
Size	4 Bytes	8 Bytes
Default value	1900-01-01 00:00:00	1900-01-01 00:00:00

The range for SmallDateTime is **January 1, 1900**, through **June 6, 2079**. A value outside of this range, is not allowed.

The following 2 queries have values outside of the range of SmallDateTime data type.

```
Insert into Employees ([SmallDateTime]) values ('01/01/1899')
Insert into Employees ([SmallDateTime]) values ('07/06/2079')
```

When executed, the above queries fail with the following error
The conversion of a varchar data type to a smalldatetime data type resulted in an out-of-range value

The range for DateTime is **January 1, 1753**, through **December 31, 9999**. A value outside of this range, is not allowed.

The following query has a value outside of the range of DateTime data type.

```
Insert into Employees ([DateTime]) values ('01/01/1752')
```

When executed, the above query fails with the following error
The conversion of a varchar data type to a datetime data type resulted in an out-of-range value.

Part 128 - DateTime2FromParts function in SQL Server 2012

In this video we will discuss **DateTime2FromParts function in SQL Server 2012**.

DateTime2FromParts function

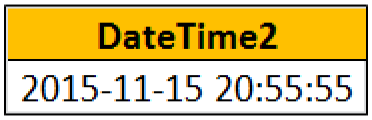
- Introduced in SQL Server 2012
- Returns DateTime2
- The data type of all the parameters is integer
- If invalid argument values are specified, the function returns an error
- If any of the required arguments are NULL, the function returns null
- If the precision argument is null, the function returns an error

Syntax : DATETIME2FROMPARTS (year, month, day, hour, minute, seconds, fractions, precision)

Example : All the function arguments have valid values, so DATETIME2FROMPARTS returns DATETIME2 value as expected.

```
SELECT DATETIME2FROMPARTS ( 2015, 11, 15, 20, 55, 55, 0, 0 ) AS [DateTime2]
```

Output :



DateTime2
2015-11-15 20:55:55

Example : Invalid value specified for month parameter, so the function returns an error

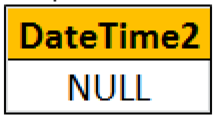
```
SELECT DATETIME2FROMPARTS ( 2015, 15, 15, 20, 55, 55, 0, 0 ) AS [DateTime2]
```

Output : Cannot construct data type datetime2, some of the arguments have values which are not valid.

Example : If any of the required arguments are NULL, the function returns null. NULL specified for month parameter, so the function returns NULL.

```
SELECT DATETIME2FROMPARTS ( 2015, NULL, 15, 20, 55, 55, 0, 0 ) AS [DateTime2]
```

Output :



DateTime2
NULL

Example : If the precision argument is null, the function returns an error

```
SELECT DATETIME2FROMPARTS ( 2015, 15, 15, 20, 55, 55, 0, NULL ) AS[DateTime2]
```

Output : Scale argument is not valid. Valid expressions for data type datetime2 scale argument are integer constants and integer constant expressions.

TIMEFROMPARTS : Returns time value

Syntax : TIMEFROMPARTS (hour, minute, seconds, fractions, precision)

Next video : We will discuss the **difference between DateTime and DateTime2 in SQL Server**

Part 129 - Difference between DateTime and DateTime2 in SQL Server

In this video we will discuss the **difference between DateTime and DateTime2 in SQL Server**

Differences between DateTime and DateTime2

Attribute	DateTime	DateTime2

Date Range	January 1, 1753, through December 31, 9999	January 1, 0001, through December 31, 9999
Time Range	00:00:00 through 23:59:59.997	00:00:00 through 23:59:59.9999999
Accuracy	3.33 Milli-seconds	100 nanoseconds
Size	8 Bytes	6 to 8 Bytes (Depends on the precision)
Default Value	1900-01-01 00:00:00	1900-01-01 00:00:00

DATETIME2 has a bigger date range than DATETIME. Also, DATETIME2 is more accurate than DATETIME. So I would recommend using DATETIME2 over DATETIME when possible. I think the only reason for using DATETIME over DATETIME2 is for backward compatibility.

DateTime2 Syntax : DATETIME2 [(fractional seconds precision)]

With DateTime2

- Optional fractional seconds precision can be specified
- The precision scale is from 0 to 7 digits
- The default precision is 7 digits
- For precision 1 and 2, storage size is 6 bytes
- For precision 3 and 4, storage size is 7 bytes
- For precision 5, 6 and 7, storage size is 8 bytes

The following script creates a table variable with 7 DATETIME2 columns with different precision start from 1 through 7

```
DECLARE @TempTable TABLE
(
   DateTime2Precision1 DATETIME2(1),
   DateTime2Precision2 DATETIME2(2),
   DateTime2Precision3 DATETIME2(3),
   DateTime2Precision4 DATETIME2(4),
   DateTime2Precision5 DATETIME2(5),
   DateTime2Precision6 DATETIME2(6),
   DateTime2Precision7 DATETIME2(7)
)
```

Insert DateTime value into each column

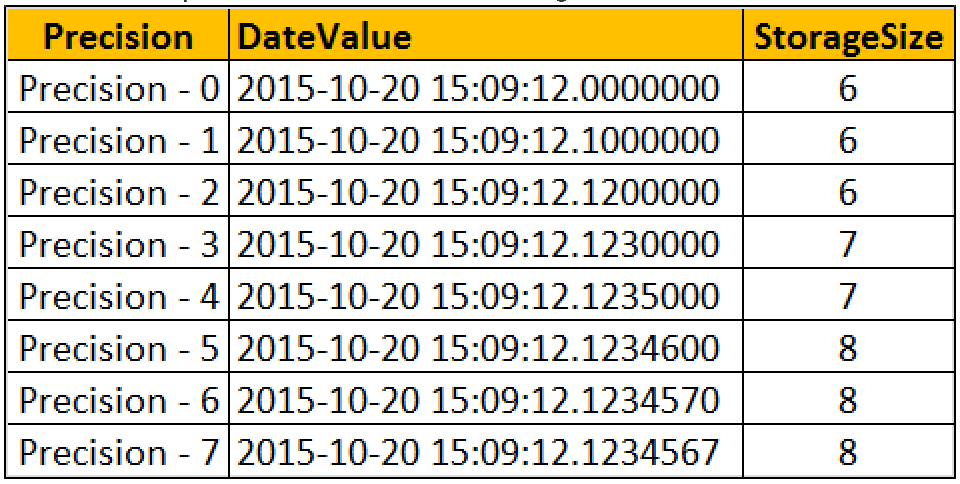
```
INSERT INTO @TempTable VALUES
(
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567',
   '2015-10-20 15:09:12.1234567'
)
```

The following query retrieves the prcision, the datetime value, and the storage size.

```
SELECT 'Precision - 1' AS [Precision],
        DateTime2Precision1 AS DateValue,
        DATALENGTH(DateTime2Precision1) AS StorageSize
FROM @TempTable

UNION ALL

SELECT 'Precision - 2',
        DateTime2Precision2,
        DATALENGTH(DateTime2Precision2) AS StorageSize
FROM @TempTable

UNION ALL

SELECT 'Precision - 3',
        DateTime2Precision3,
        DATALENGTH(DateTime2Precision3)
FROM @TempTable

UNION ALL

SELECT 'Precision - 4',
        DateTime2Precision4,
        DATALENGTH(DateTime2Precision4)
FROM @TempTable

UNION ALL

SELECT 'Precision - 5',
        DateTime2Precision5,
        DATALENGTH(DateTime2Precision5)
FROM @TempTable

UNION ALL
```

```
SELECT 'Precision - 6',
       DateTime2Precision6,
       DATALENGTH(DateTime2Precision6)
FROM @TempTable

UNION ALL
SELECT 'Precision - 7',
       DateTime2Precision7,
       DATALENGTH(DateTime2Precision7) AS StorageSize
FROM @TempTable
```

Notice as the precision increases the storage size also increases



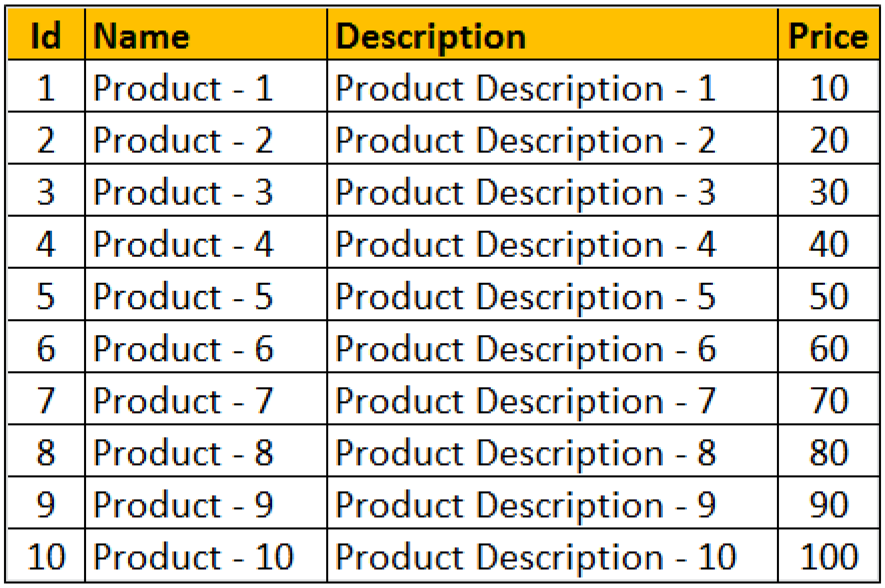
Precision	DateValue	StorageSize
Precision - 0	2015-10-20 15:09:12.0000000	6
Precision - 1	2015-10-20 15:09:12.1000000	6
Precision - 2	2015-10-20 15:09:12.1200000	6
Precision - 3	2015-10-20 15:09:12.1230000	7
Precision - 4	2015-10-20 15:09:12.1235000	7
Precision - 5	2015-10-20 15:09:12.1234600	8
Precision - 6	2015-10-20 15:09:12.1234570	8
Precision - 7	2015-10-20 15:09:12.1234567	8

Part 130 - Offset fetch next in SQL Server 2012

In this video we will discuss **OFFSET FETCH Clause in SQL Server 2012**

One of the common tasks for a SQL developer is to come up with a stored procedure that can return a page of results from the result set. With SQL Server 2012 OFFSET FETCH Clause it is very easy to implement paging.

Let's understand this with an example. We will use the following **tblProducts** table for the examples in this video. The table has got 100 rows. In the image I have shown just 10 rows.



Id	Name	Description	Price
1	Product - 1	Product Description - 1	10
2	Product - 2	Product Description - 2	20
3	Product - 3	Product Description - 3	30
4	Product - 4	Product Description - 4	40
5	Product - 5	Product Description - 5	50
6	Product - 6	Product Description - 6	60
7	Product - 7	Product Description - 7	70
8	Product - 8	Product Description - 8	80
9	Product - 9	Product Description - 9	90
10	Product - 10	Product Description - 10	100

SQL Script to create tblProducts table

```
Create table tblProducts
(
    Id int primary key identity,
    Name nvarchar(25),
    [Description] nvarchar(50),
    Price int
)
Go
```

SQL Script to populate tblProducts table with 100 rows

```
Declare @Start int
Set @Start = 1

Declare @Name varchar(25)
Declare @Description varchar(50)

While(@Start <= 100)
Begin
    Set @Name = 'Product - ' + LTRIM(@Start)
    Set @Description = 'Product Description - ' + LTRIM(@Start)
    Insert into tblProducts values (@Name, @Description, @Start * 10)
    Set @Start = @Start + 1
End
```

OFFSET FETCH Clause

- Introduced in SQL Server 2012
- Returns a page of results from the result set
- ORDER BY clause is required

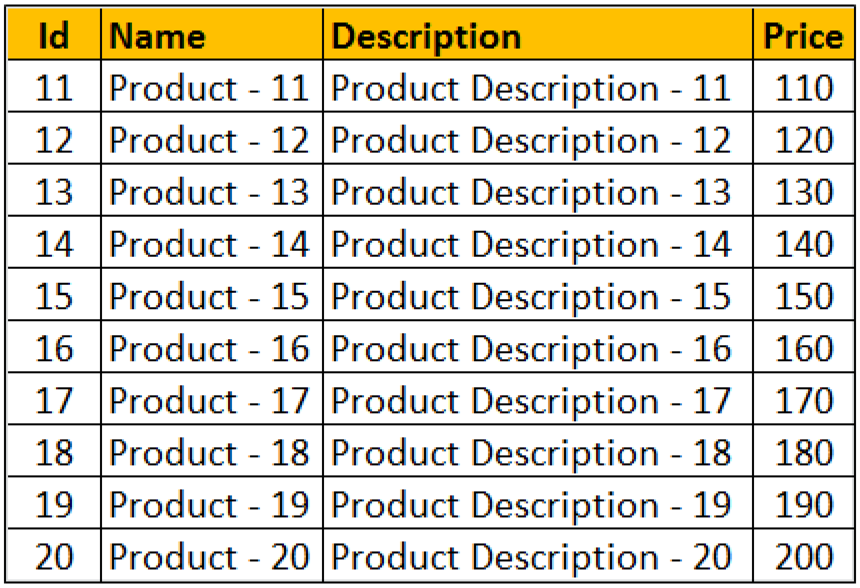
OFFSET FETCH Syntax :

```
SELECT * FROM Table_Name
ORDER BY Column_List
OFFSET Rows_To_Skip ROWS
FETCH NEXT Rows_To_Fetch ROWS ONLY
```

The following SQL query
1. Sorts the table data by Id column
2. Skips the first 10 rows and
3. Fetches the next 10 rows

```
SELECT * FROM tblProducts
ORDER BY Id
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY
```

Result :



Id	Name	Description	Price
11	Product - 11	Product Description - 11	110
12	Product - 12	Product Description - 12	120
13	Product - 13	Product Description - 13	130
14	Product - 14	Product Description - 14	140
15	Product - 15	Product Description - 15	150
16	Product - 16	Product Description - 16	160
17	Product - 17	Product Description - 17	170
18	Product - 18	Product Description - 18	180
19	Product - 19	Product Description - 19	190
20	Product - 20	Product Description - 20	200

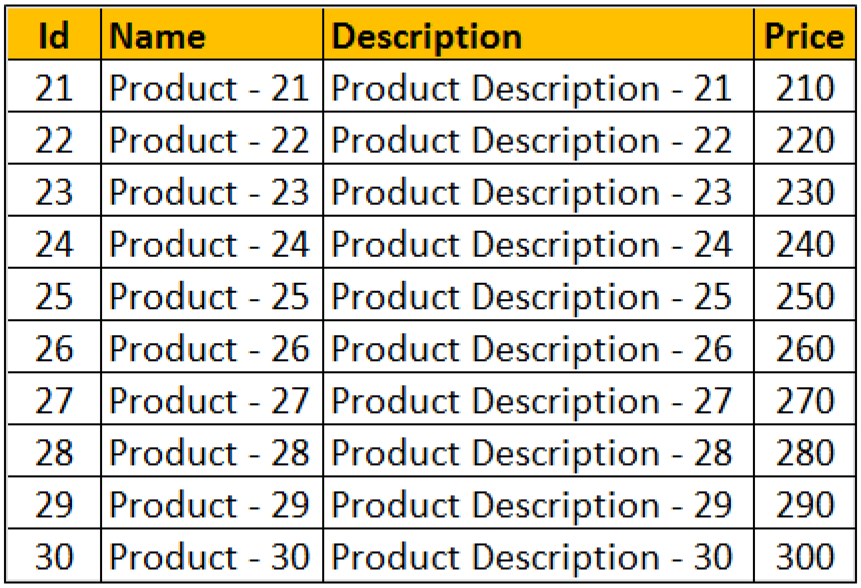
From the front-end application, we would typically send the **PAGE NUMBER** and the**PAGE SIZE** to get a page of rows. The following stored procedure accepts PAGE NUMBER and the PAGE SIZE as parameters and returns the correct set of rows.

```
CREATE PROCEDURE spGetRowsByPageNumberAndSize
@PageNumber INT,
@PageSize INT
AS
```

```
BEGIN
   SELECT * FROM tblProducts
   ORDER BY Id
   OFFSET (@PageNumber - 1) * @PageSize ROWS
   FETCH NEXT @PageSize ROWS ONLY
END
```

With PageNumber = 3 and PageSize = 10, the stored procedure returns the correct set of rows

```
EXECUTE spGetRowsByPageNumberAndSize 3, 10
```



Id	Name	Description	Price
21	Product - 21	Product Description - 21	210
22	Product - 22	Product Description - 22	220
23	Product - 23	Product Description - 23	230
24	Product - 24	Product Description - 24	240
25	Product - 25	Product Description - 25	250
26	Product - 26	Product Description - 26	260
27	Product - 27	Product Description - 27	270
28	Product - 28	Product Description - 28	280
29	Product - 29	Product Description - 29	290
30	Product - 30	Product Description - 30	300

Part 131 - Identifying object dependencies in SQL Server

In this video we will discuss **how to identify object dependencies in SQL Server using SQL Server Management Studio**.

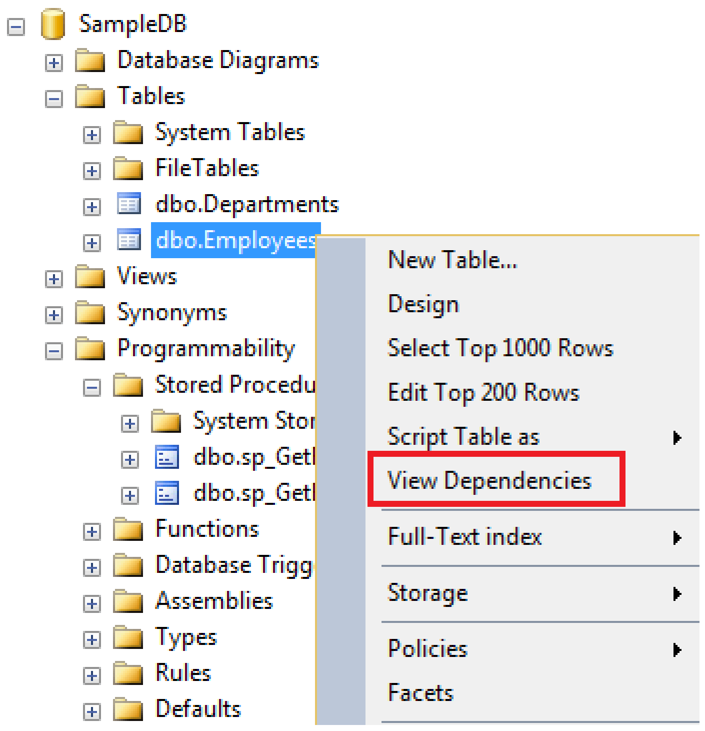
The following SQL Script creates 2 tables, 2 stored procedures and a view

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

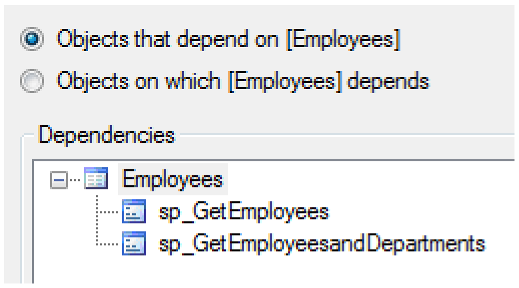
```
Create table Employees
(
   Id int primary key identity,
   Name nvarchar(50),
   Gender nvarchar(10),
   DeptId int foreign key references Departments(Id)
)
Go

Create procedure sp_GetEmployees
as
Begin
   Select * from Employees
End
Go

Create procedure sp_GetEmployeesandDepartments
as
Begin
   Select Employees.Name as EmployeeName,
              Departments.Name as DepartmentName
   from Employees
   join Departments
   on Employees.DeptId = Departments.Id
End
Go

Create view VwDepartments
as
Select * from Departments
Go
```

How to find dependencies using SQL Server Management Studio
Use View Dependencies option in SQL Server Management studio to find the object dependencies

For example : To find the dependencies on the Employees table, right click on it and select View Dependencies from the context menu



In the **Object Dependencies** window, depending on the radio button you select, you can find the objects that depend on **Employees** table and the objects on which **Employees**table depends on.


(● Objects that depend on [Employees])
Objects on which [Employees] depends
Dependencies
Employees
sp_GetEmployees
sp_GetEmployeesandDepartments

Identifying object dependencies is important especially when you intend to modify or delete an object upon which other objects depend. Otherwise you may risk breaking the functionality.

For example, there are 2 stored procedures (sp_GetEmployees and sp_GetEmployeesandDepartments) that depend on the Employees table. If we are not aware of these dependencies and if we delete the Employees table, both stored procedures will fail with the following error.

```
Msg 208, Level 16, State 1, Procedure sp_GetEmployees, Line 4
Invalid object name 'Employees'.
```

There are other ways for finding object dependencies in SQL Server which we will discuss in our upcoming videos.

Part 132 - sys.dm_sql_referencing_entities in SQL Server

In this video we will discuss

- How to find object dependencies using the following dynamic management functions
- sys.dm_sql_referencing_entities
- sys.dm_sql_referenced_entities
- Difference between
- Referencing entity and Referenced entity
- Schema-bound dependency and Non-schema-bound dependency

This is continuation to Part 131, in which we discussed how to find object dependencies using SQL Server Management Studio. Please watch Part 131 from SQL Server tutorialbefore proceeding.

The following example returns all the objects that depend on Employees table.

```
Select * from sys.dm_sql_referencing_entities('dbo.Employees','Object')
```

Difference between referencing entity and referenced entity
A dependency is created between two objects when one object appears by name inside a SQL statement stored in another object. The object which is appearing inside the SQL expression is known as referenced entity and the object which has the SQL expression is known as a referencing entity.

To get the REFERENCING ENTITIES use SYS.DM_SQL_REFERENCING_ENTITIES dynamic management function

To get the REFERENCED ENTITIES use SYS.DM_SQL_REFERENCED_ENTITIES dynamic management function

Now, let us say we have a stored procedure and we want to find the all objects that this stored procedure depends on. This can be very achieved using another dynamic management function, sys.dm_sql_referenced_entities.

The following query returns all the referenced entities of the stored procedure sp_GetEmployeesandDepartments

```
Select * from
sys.dm_sql_referenced_entities('dbo.sp_GetEmployeesandDepartments','Object')
```

Please note : For both these dynamic management functions to work we need to specify the schema name as well. Without the schema name you may not get any results.

Difference between Schema-bound dependency and Non-schema-bound dependency

Schema-bound dependency : Schema-bound dependency prevents referenced objects from being dropped or modified as long as the referencing object exists

Example : A view created with SCHEMABINDING, or a table created with foreign key constraint.

Non-schema-bound dependency : A non-schema-bound dependency doesn't prevent the referenced object from being dropped or modified.

Part 133 - sp_depends in SQL Server

In this video we will discuss sp_depends system stored procedure.

There are several ways to find object dependencies in SQL Server

1. View Dependencies feature in SQL Server Management Studio - Part 131

2. SQL Server dynamic management functions - Part 132

sys.dm_sql_referencing_entities

sys.dm_sql_referenced_entities

3. sp_depends system stored procedure - This video

sp_depends

A system stored procedure that returns object dependencies
For example,

- If you specify a table name as the argument, then the views and procedures that depend on the specified table are displayed
- If you specify a view or a procedure name as the argument, then the tables and views on which the specified view or procedure depends are displayed.

Syntax : `Execute sp_depends 'ObjectName'`

The following SQL Script creates a table and a stored procedure

```
Create table Employees
(
   Id int primary key identity,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go

Create procedure sp_GetEmployees
```

```
as
Begin
   Select * from Employees
End
Go
```

Returns the stored procedure that depends on table Employees

```
sp_depends 'Employees'
```

Ouptut :

name	type
dbo.sp_GetEmployees	stored procedure

Returns the name of the table and the respective column names on which the stored procedure sp_GetEmployees depends

```
sp_depends 'sp_GetEmployees'
```

Output :

name	type	updated	selected	column
dbo.Employees	user table	no	yes	Id
dbo.Employees	user table	no	yes	Name
dbo.Employees	user table	no	yes	Gender

Sometime sp_depends does not report dependencies correctly. For example, at the moment we have Employees table and a stored procedure sp_GetEmployees.

Now drop the table Employees

```
Drop table Employees
```

and then recreate the table again

```
Create table Employees
(
   Id int primary key identity,
   Name nvarchar(50),
   Gender nvarchar(10)
)
Go
```

Now execute the following, to find the objects that depend on Employees table

```
sp_depends 'Employees'
```

We know that stored procedure **sp_GetEmployees** still depends on **Employees** table. But sp_depends does not report this dependency, as the Employees table is dropped and

recreated.
Object does not reference any object, and no objects reference it.

sp_depends is on the deprecation path. This might be removed from the future versions of SQL server.

Part 134 - Sequence object in SQL Server 2012

In this video we will discuss **sequence object in SQL Server**.

Sequence object

- Introduced in SQL Server 2012
- Generates sequence of numeric values in an ascending or descending order

Syntax :

```
CREATE SEQUENCE [schema_name . ] sequence_name
   [ AS [ built_in_integer_type | user-defined_integer_type ] ]
   [ START WITH <constant> ]
   [ INCREMENT BY <constant> ]
   [ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]
   [ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]
   [ CYCLE | { NO CYCLE } ]
   [ { CACHE [ <constant> ] } | { NO CACHE } ]
   [ ; ]
```

Property	Description
DataType	Built-in integer type (tinyint , smallint, int, bigint, decimal etc...) or user-defined integer type. Default bigint.
START WITH	The first value returned by the sequence object
INCREMENT BY	The value to increment or decrement by. The value will be decremented if a negative value is specified.
MINVALUE	Minimum value for the sequence object
MAXVALUE	Maximum value for the sequence object
CYCLE	Specifies whether the sequence object should restart when the max value (for incrementing sequence object) or min value (for decrementing sequence object) is reached. Default is NO CYCLE, which throws an error when minimum or maximum value is exceeded.
CACHE	Cache sequence values for performance. Default value is CACHE.

Creating an Incrementing Sequence : The following code create a Sequence object that starts with 1 and increments by 1

```
CREATE SEQUENCE [dbo].[SequenceObject]
AS INT
START WITH 1
INCREMENT BY 1
```

Generating the Next Sequence Value : Now we have a sequence object created. To generate the sequence value use NEXT VALUE FOR clause

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

Output : 1

Every time you execute the above query the sequence value will be incremented by 1. I executed the above query 5 times, so the current sequence value is 5.

Retrieving the current sequence value : If you want to see what the current Sequence value before generating the next, use **sys.sequences**

```
SELECT * FROM sys.sequences WHERE name = 'SequenceObject'
```

Alter the Sequence object to reset the sequence value :

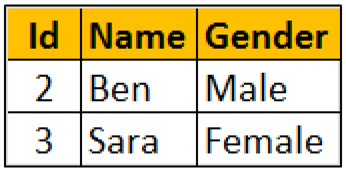
```
ALTER SEQUENCE [SequenceObject] RESTART WITH 1
```

Select the next sequence value to make sure the value starts from 1

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

Using sequence value in an INSERT query :

```
CREATE TABLE Employees
(
   Id INT PRIMARY KEY,
   Name NVARCHAR(50),
   Gender NVARCHAR(10)
)

-- Generate and insert Sequence values
INSERT INTO Employees VALUES
(NEXT VALUE for [dbo].[SequenceObject], 'Ben', 'Male')
INSERT INTO Employees VALUES
(NEXT VALUE for [dbo].[SequenceObject], 'Sara', 'Female')

-- Select the data from the table
SELECT * FROM Employees
```



Id	Name	Gender
2	Ben	Male
3	Sara	Female

Creating the decrementing Sequence : The following code create a Sequence object that starts with 100 and decrements by 1

```
CREATE SEQUENCE [dbo].[SequenceObject]
AS INT
START WITH 100
INCREMENT BY -1
```

Specifying MIN and MAX values for the sequence : Use the MINVALUE and MAXVALUE arguments to specify the MIN and MAX values respectively.

Step 1 : Create the Sequence object

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

Step 2 : Retrieve the next sequence value. The sequence value starts at 100. Every time we call NEXT VALUE, the value will be incremented by 10.

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

If you call NEXT VALUE, when the value reaches 150 (MAXVALUE), you will get the following error
The sequence object 'SequenceObject' has reached its minimum or maximum value. Restart the sequence object to allow new values to be generated.

Recycling Sequence values : When the sequence object has reached it's maximum value, and if you want to restart from the minimum value, set CYCLE option

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

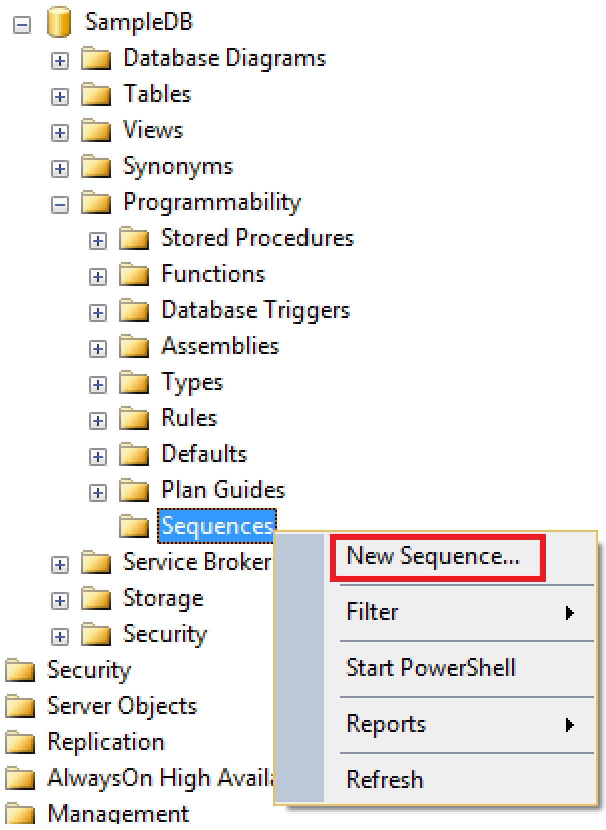
At this point, whe the sequence object has reached it's maximum value, and if you ask for the NEXT VALUE, sequence object starts from the minimum value again which in this case is 100.

To improve performance, the Sequence object values can be cached using the CACHE option. When the values are cached they are read from the memory instead of from the disk, which improves the performance. When the cache option is specified you can also specify the size of th cache , that is the number of values to cache.

The following example, creates the sequence object with 10 values cached. When the 11th value is requested, the next 10 values will be cached again.

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

Using SQL Server Graphical User Interface (GUI) to create the sequence object :
1. Expand the database folder
2. Expand Programmability folder
3. Right click on Sequences folder
4. Select New Sequence



Next video : Difference between SEQUENCE and IDENTITY in SQL Server

Part 135 - Difference between sequence and identity in SQL Server

In this video we will discuss the **difference between SEQUENCE and IDENTITY in SQL Server**

This is continuation to Part 134. Please watch Part 134 from SQL Server tutorial before proceeding.

Sequence object is similar to the Identity property, in the sense that it generates sequence of numeric values in an ascending order just like the identity property. However there are several differences between the 2 which we will discuss in this video.

Identity property is a table column property meaning it is tied to the table, where as the sequence is a user-defined database object and is not tied to any specific table meaning it's value can be shared by multiple tables.

Example : Identity property tied to the Id column of the Employees table.

```
CREATE TABLE Employees
(
   Id INT PRIMARY KEY IDENTITY(1,1),
   Name NVARCHAR(50),
   Gender NVARCHAR(10)
)
```

Example : Sequence object not tied to any specific table

```
CREATE SEQUENCE [dbo].[SequenceObject]
AS INT
START WITH 1
INCREMENT BY 1
```

This means the above sequence object can be used with any table.

Example : Sharing sequence object value with multiple tables.

Step 1 : Create Customers and Users tables

```
CREATE TABLE Customers
(
   Id INT PRIMARY KEY,
   Name NVARCHAR(50),
   Gender NVARCHAR(10)
)
GO
```

```
CREATE TABLE Users
(
   Id INT PRIMARY KEY,
   Name NVARCHAR(50),
   Gender NVARCHAR(10)
)
GO
```

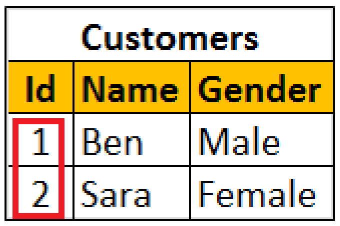
Step 2 : Insert 2 rows into Customers table and 3 rows into Users table. Notice the same sequence object is generating the ID values for both the tables.

```
INSERT INTO Customers VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Ben', 'Male')
INSERT INTO Customers VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Sara', 'Female')

INSERT INTO Users VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Tom', 'Male')
INSERT INTO Users VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'Pam', 'Female')
INSERT INTO Users VALUES
  (NEXT VALUE for [dbo].[SequenceObject], 'David', 'Male')
GO
```

Step 3 : Query the tables

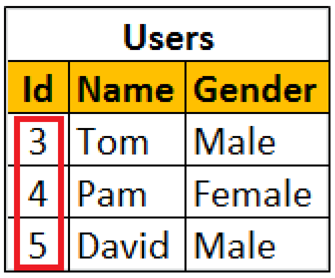
```
SELECT * FROM Customers
SELECT * FROM Users
GO
```

Output : Notice the same sequence object has generated the values for ID columns in both the tables



Customers

Id	Name	Gender
1	Ben	Male
2	Sara	Female



Users

Id	Name	Gender
3	Tom	Male
4	Pam	Female
5	David	Male

To generate the next identity value, a row has to be inserted into the table, where as with sequence object there is no need to insert a row into the table to generate the next sequence value. You can use NEXT VALUE FOR clause to generate the next sequence value.

Example : Generating Identity value by inserting a row into the table

```
INSERT INTO Employees VALUES ('Todd', 'Male')
```

Example : Generating the next sequence value using NEXT VALUE FOR clause.

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

Maximum value for the identity property cannot be specified. The maximum value will be the maximum value of the correspoding column data type. With the sequence object you can use the MAXVALUE option to specify the maximum value. If the MAXVALUE option is not specified for the sequence object, then the maximum value will be the maximum value of it's data type.

Example : Specifying maximum value for the sequence object using the MAXVALUE option

```
CREATE SEQUENCE [dbo].[SequenceObject]
START WITH 1
INCREMENT BY 1
MAXVALUE 5
```

CYCLE option of the Sequence object can be used to specify whether the sequence should restart automatically when the max value (for incrementing sequence object) or min value (for decrementing sequence object) is reached, where as with the Identity property we don't have any such option to automatically restart the identity values.

Example : Specifying the CYCLE option of the Sequence object, so the sequence will restart automatically when the max value is exceeded

```
CREATE SEQUENCE [dbo].[SequenceObject]
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 5
CYCLE
```