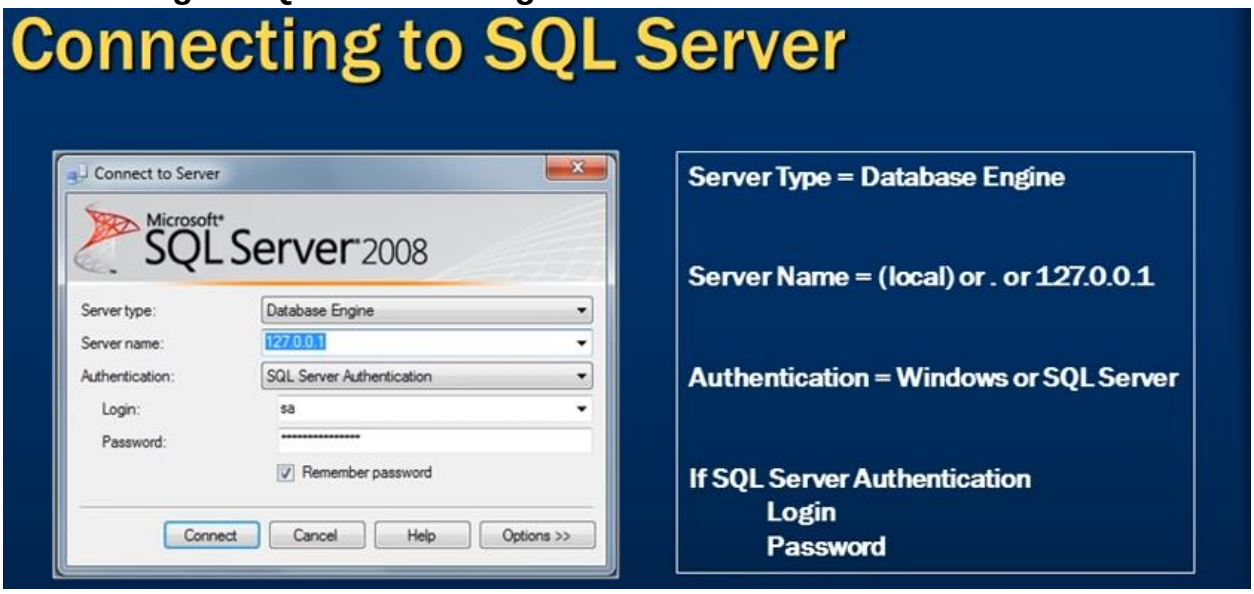
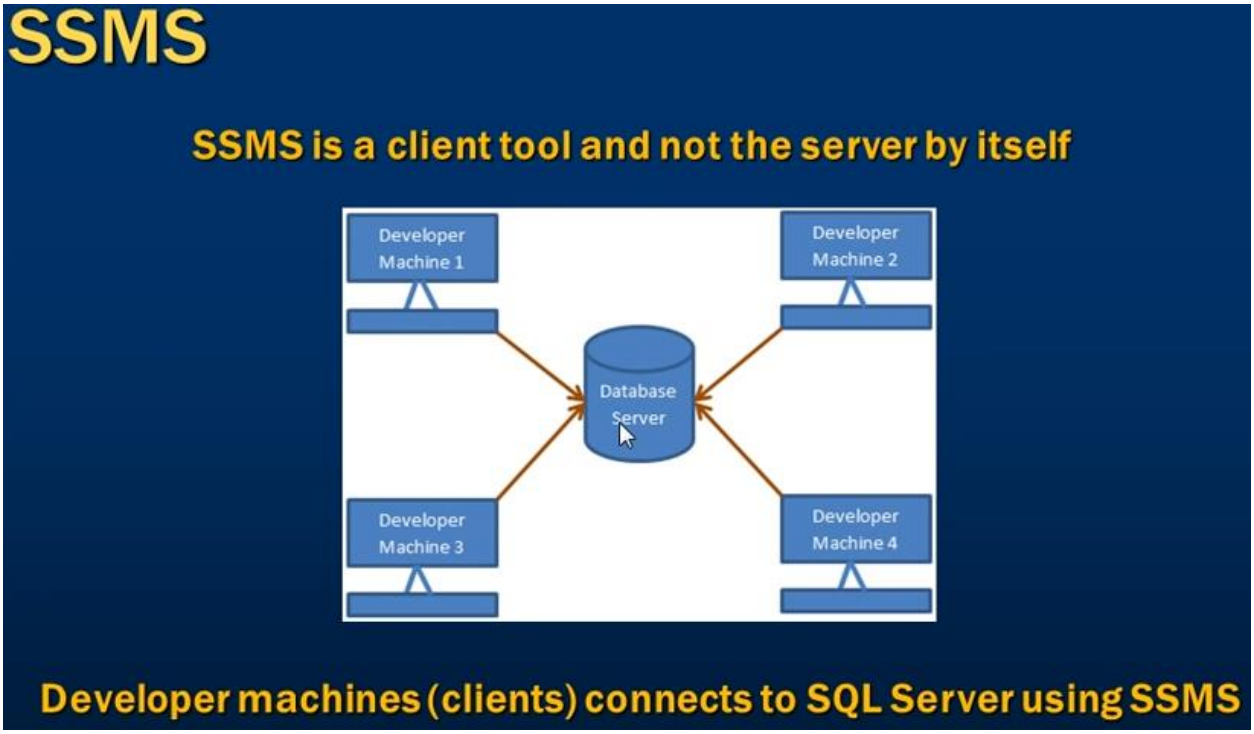
SQL and SQL Server

Connecting to SQL Server using SSMS

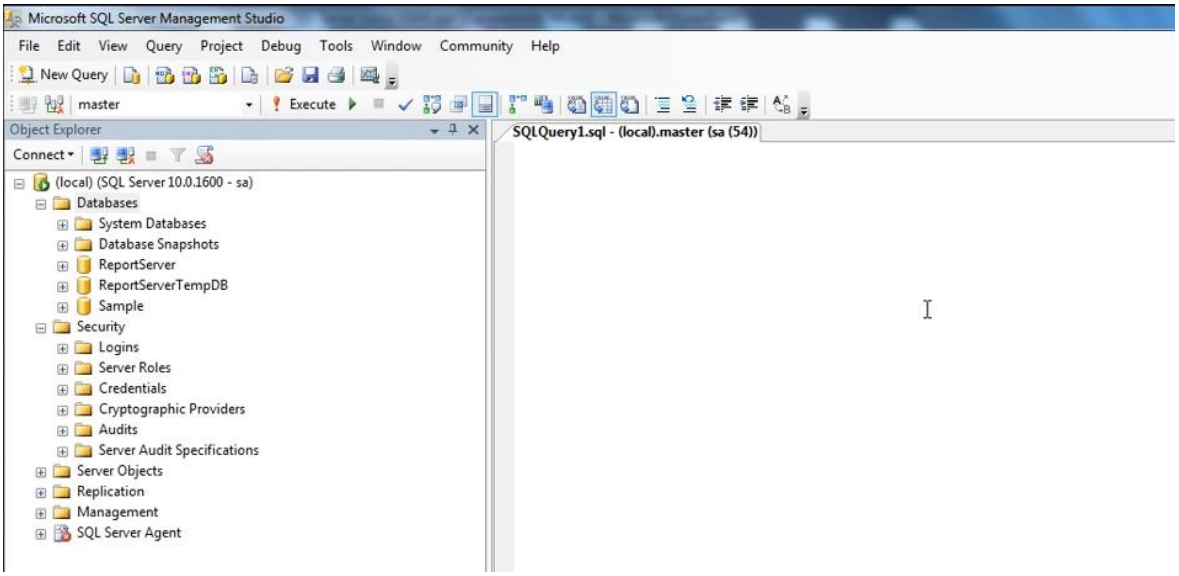


SSMS

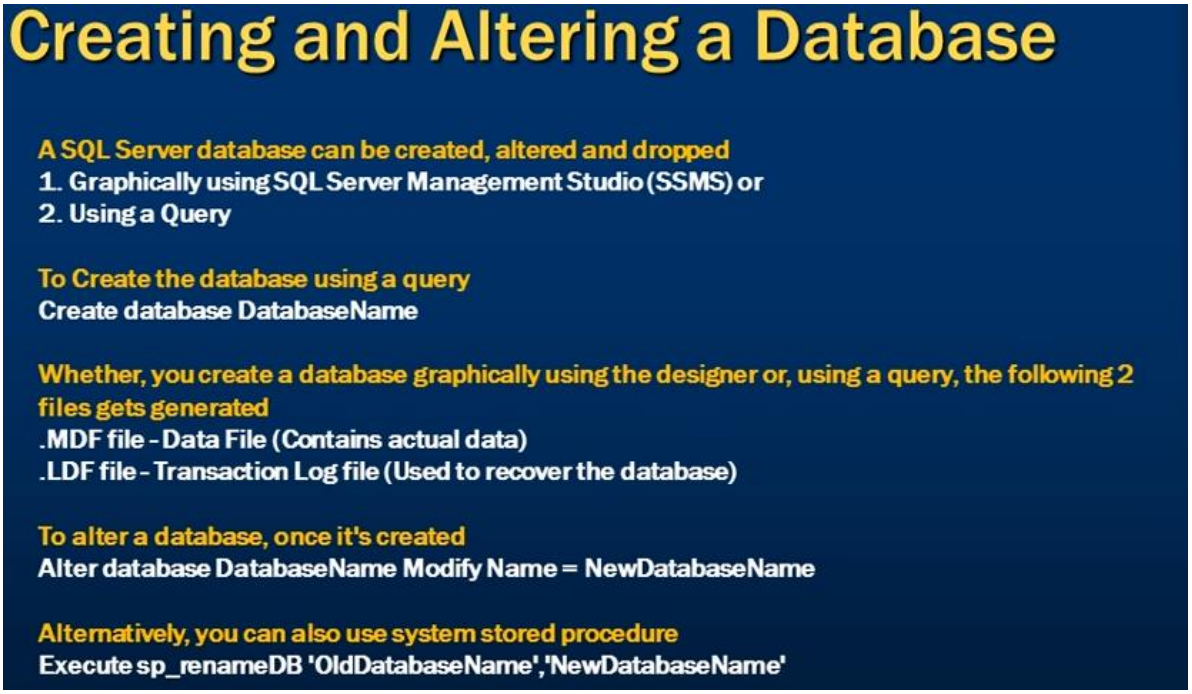


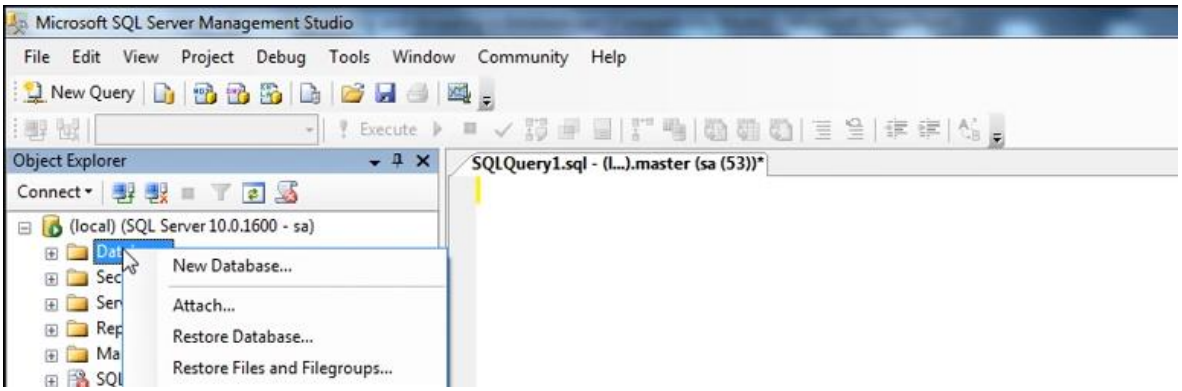
So, if other users are connected, you need to put the database in single user mode and then drop the database.

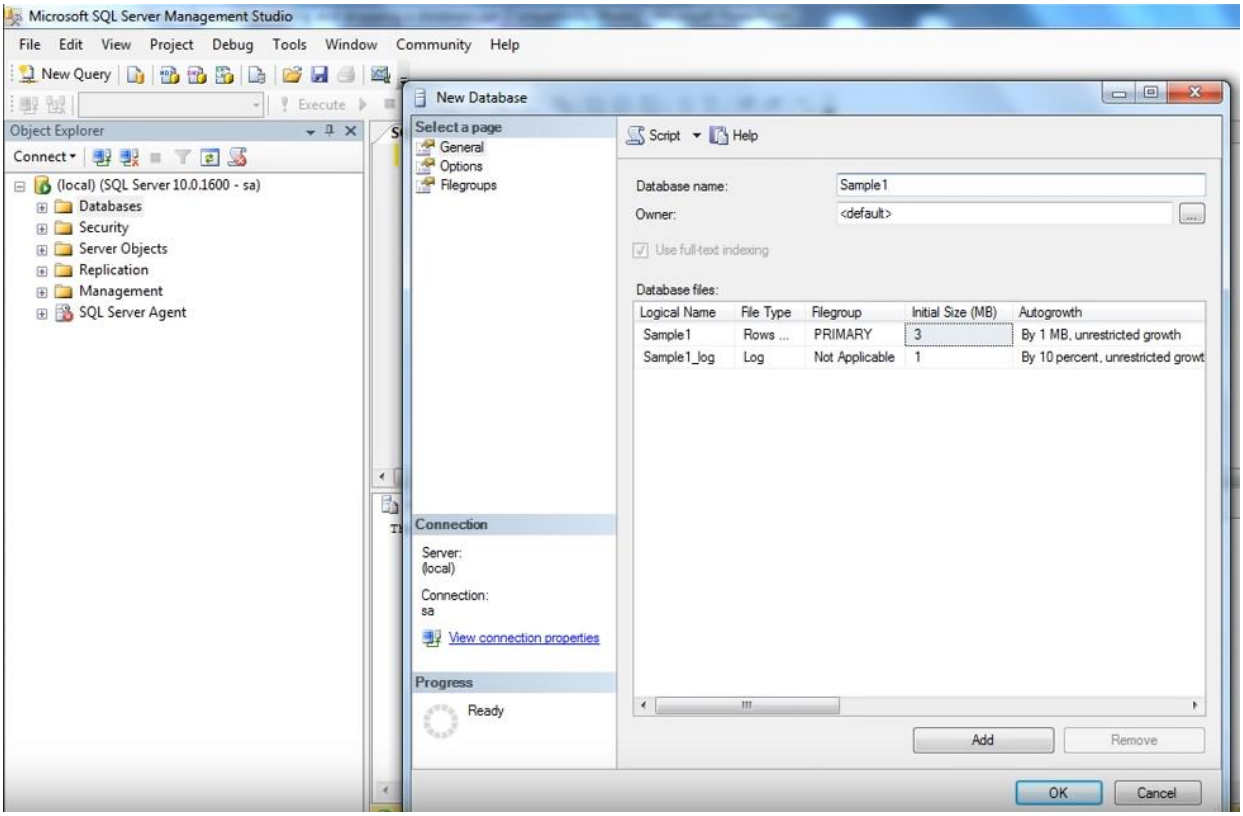
Alter Database DatabaseName Set SINGLE_USER With Rollback Immediate

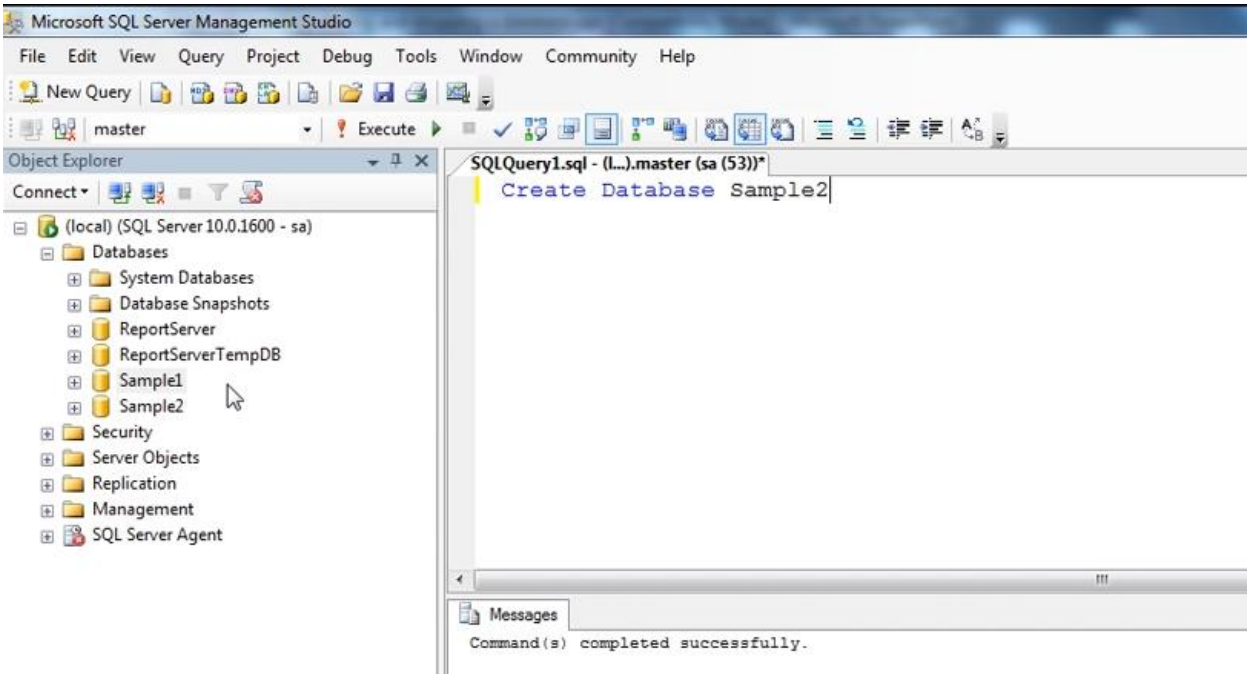


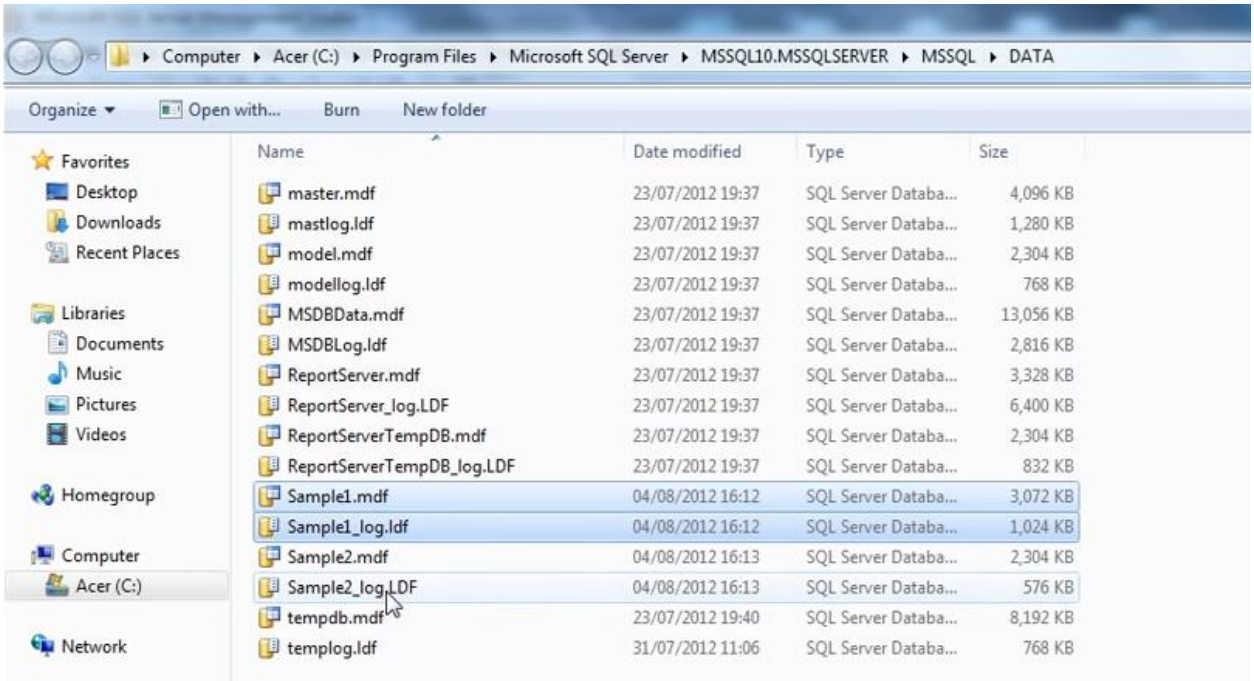
Creating, Altering, Dropping a Database

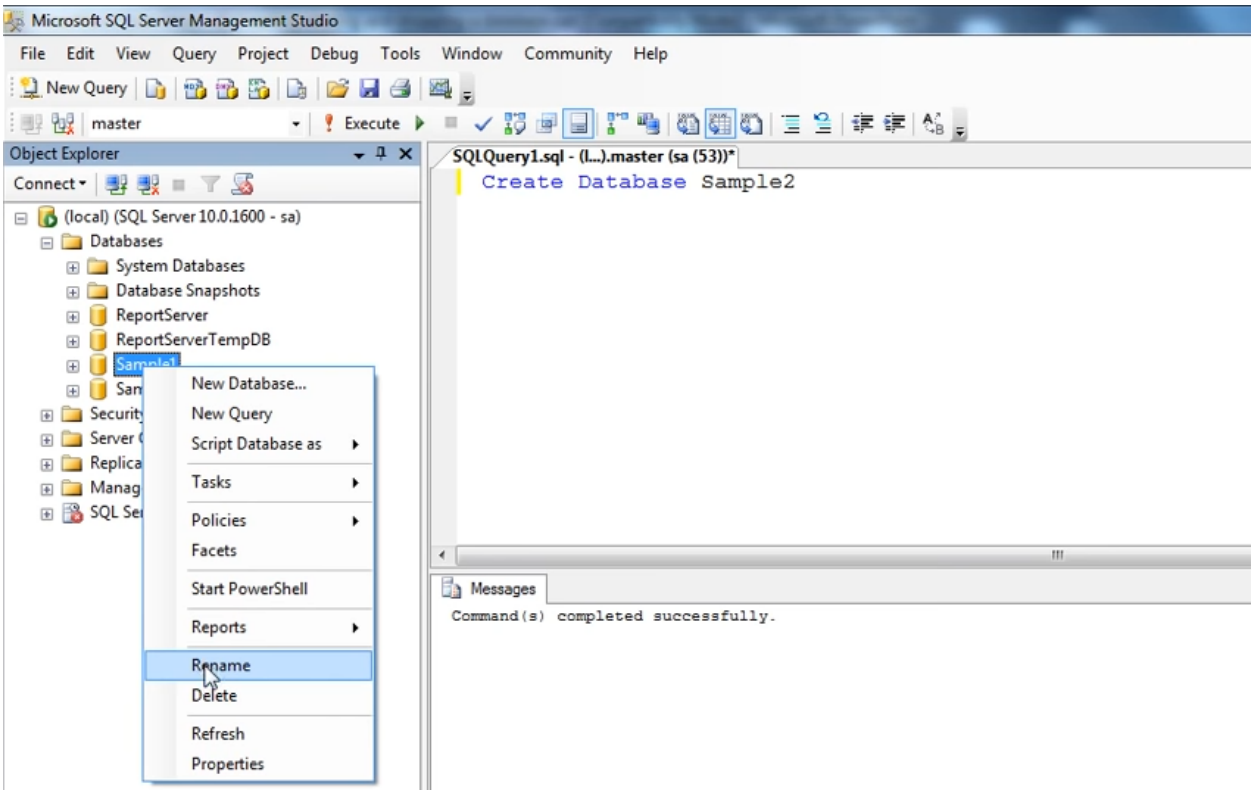


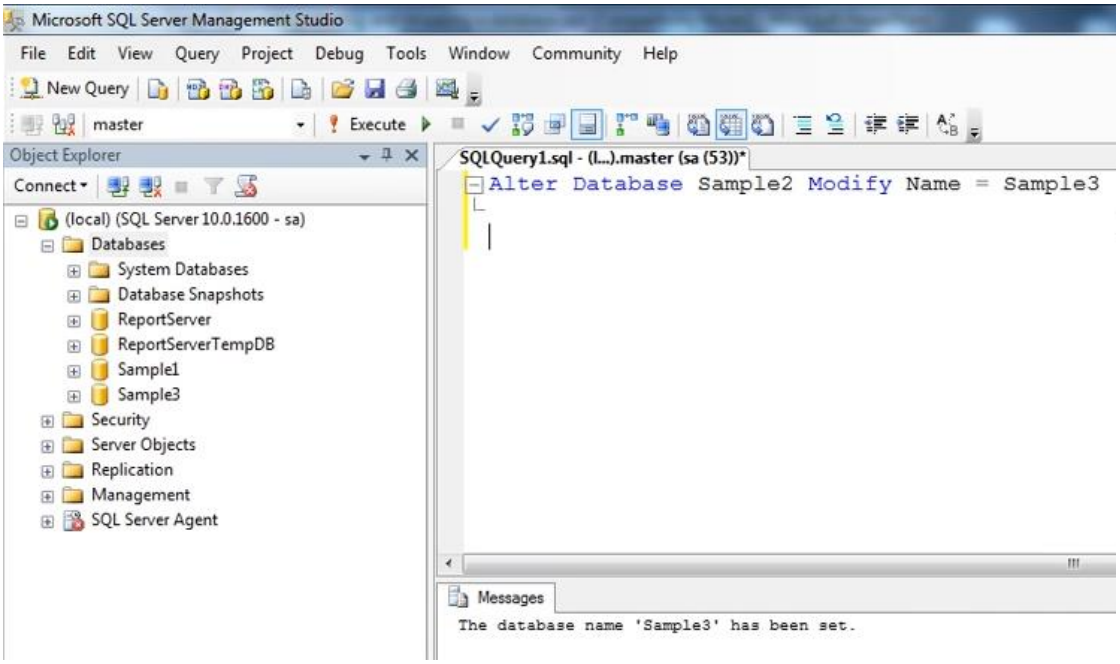



Microsoft SQL Server Management Studio
File Edit View Project Debug Tools Window Community Help
New Query
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
Security
Server Objects
Replication
Management
SQL Server Agent
New Database
Select a page
General
Options
Filegroups
Script
Help
Database name:
Sample1
Owner:
<default>
Use full-text indexing
Database files:
Logical Name
File Type
Filegroup
Initial Size (MB)
Autogrowth
Sample1
Rows ...
PRIMARY
3
By 1 MB, unrestricted growth
Sample1_log
Log
Not Applicable
1
By 10 percent, unrestricted growt
Connection
Server:
(local)
Connection:
sa
View connection properties
Progress
Ready
Add
Remove
OK
Cancel


Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
master
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample1
Sample2
Security
Server Objects
Replication
Management
SQL Server Agent
SQLQuery1.sql - (l...).master (sa (53))*
Create Database Sample2
Messages
Command(s) completed successfully.


Computer ▸ Acer (C:) ▸ Program Files ▸ Microsoft SQL Server ▸ MSSQL10.MSSQLSERVER ▸ MSSQL ▸ DATA
Organize
Open with...
Burn
New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Homegroup
Computer
Acer (C:)
Network
Name
Date modified
Type
Size
master.mdf 23/07/2012 19:37 SQL Server Databa... 4,096 KB
mastlog.ldf 23/07/2012 19:37 SQL Server Databa... 1,280 KB
model.mdf 23/07/2012 19:37 SQL Server Databa... 2,304 KB
modellog.ldf 23/07/2012 19:37 SQL Server Databa... 768 KB
MSDBData.mdf 23/07/2012 19:37 SQL Server Databa... 13,056 KB
MSDBLog.ldf 23/07/2012 19:37 SQL Server Databa... 2,816 KB
ReportServer.mdf 23/07/2012 19:37 SQL Server Databa... 3,328 KB
ReportServer_log.LDF 23/07/2012 19:37 SQL Server Databa... 6,400 KB
ReportServerTempDB.mdf 23/07/2012 19:37 SQL Server Databa... 2,304 KB
ReportServerTempDB_log.LDF 23/07/2012 19:37 SQL Server Databa... 832 KB
Sample1.mdf 04/08/2012 16:12 SQL Server Databa... 3,072 KB
Sample1_log.ldf 04/08/2012 16:12 SQL Server Databa... 1,024 KB
Sample2.mdf 04/08/2012 16:13 SQL Server Databa... 2,304 KB
Sample2_log.LDF 04/08/2012 16:13 SQL Server Databa... 576 KB
tempdb.mdf 23/07/2012 19:40 SQL Server Databa... 8,192 KB
templog.ldf 31/07/2012 11:06 SQL Server Databa... 768 KB


Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
master
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample1
New Database...
New Query
Script Database as
Tasks
Policies
Facets
Start PowerShell
Reports
Rename
Delete
Refresh
Properties
SQLQuery1.sql - (l...).master (sa (53))*
Create Database Sample2
Messages
Command(s) completed successfully.



Deleting or Dropping a database

To Delete or Drop a database
Drop Database DatabaseThatYouWantToDrop

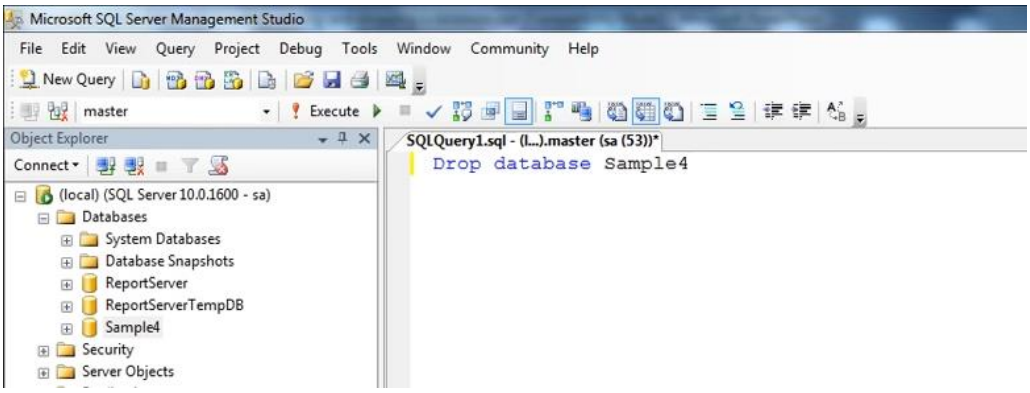
Dropping a database, deletes the LDF and MDF files.

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

Creating Tables

The aim of this session is to create tblPerson and tblGender tables and establish primary key and foreign key constraints.

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

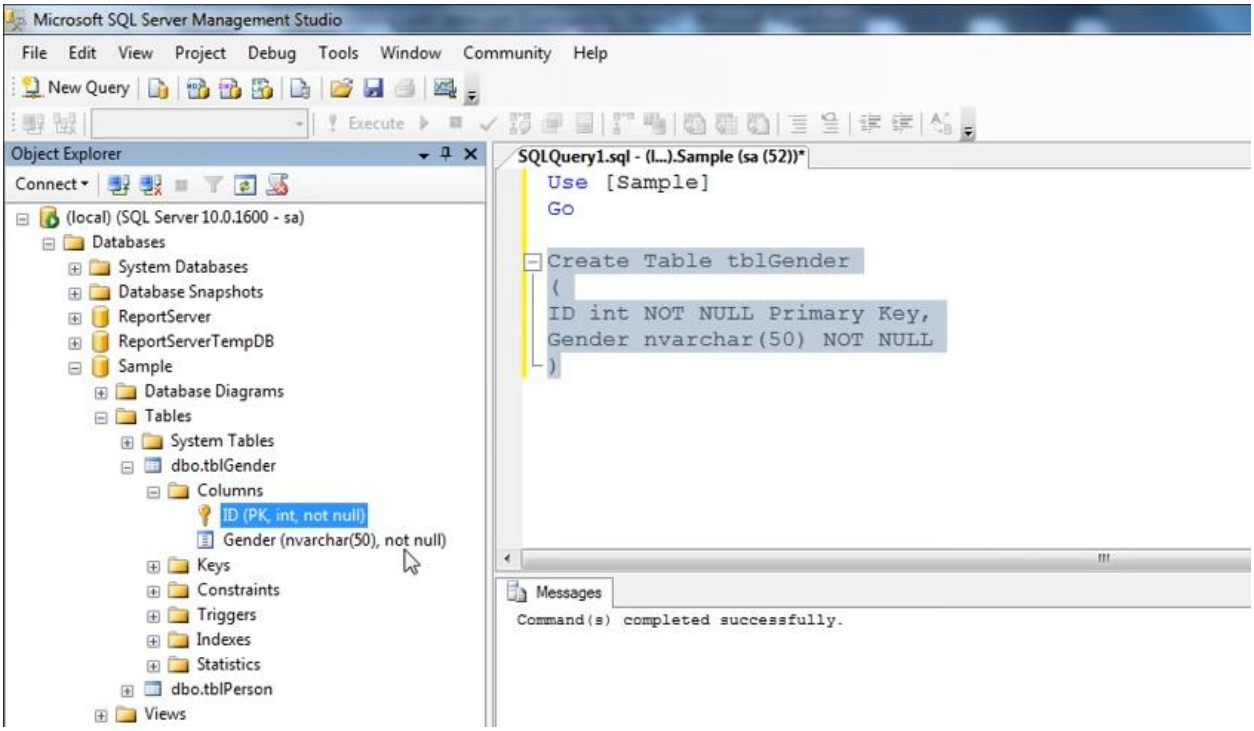
tblGender

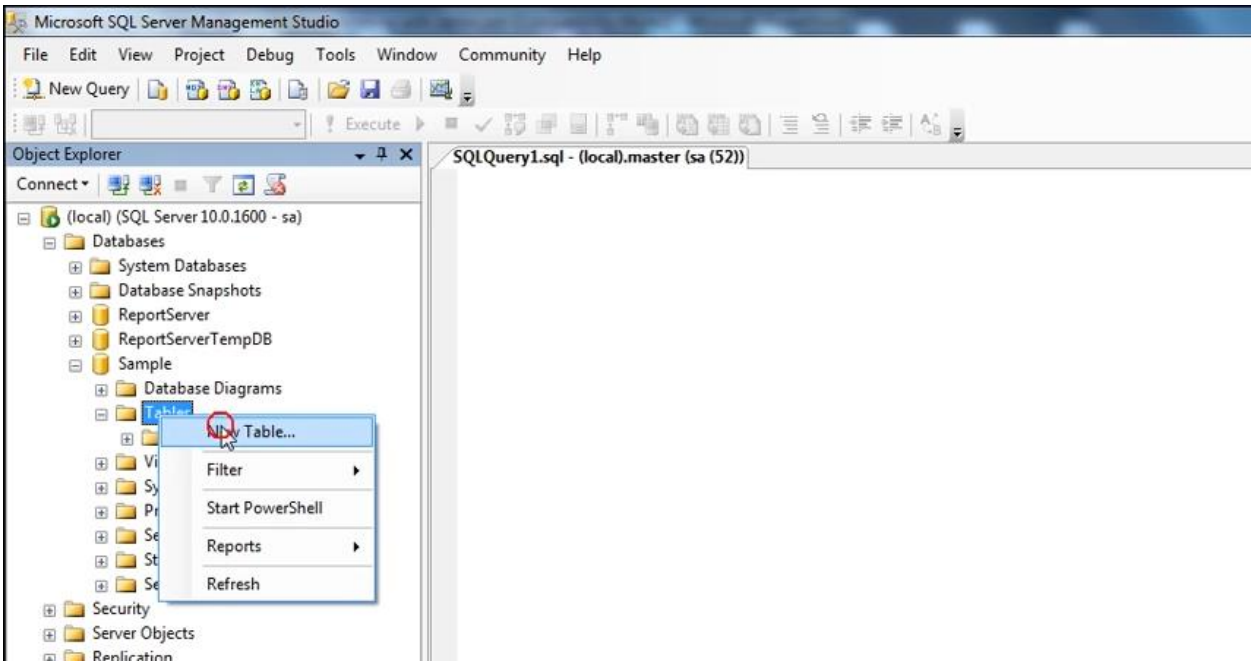
ID	Gender
1	Male
2	Female
3	Unknown

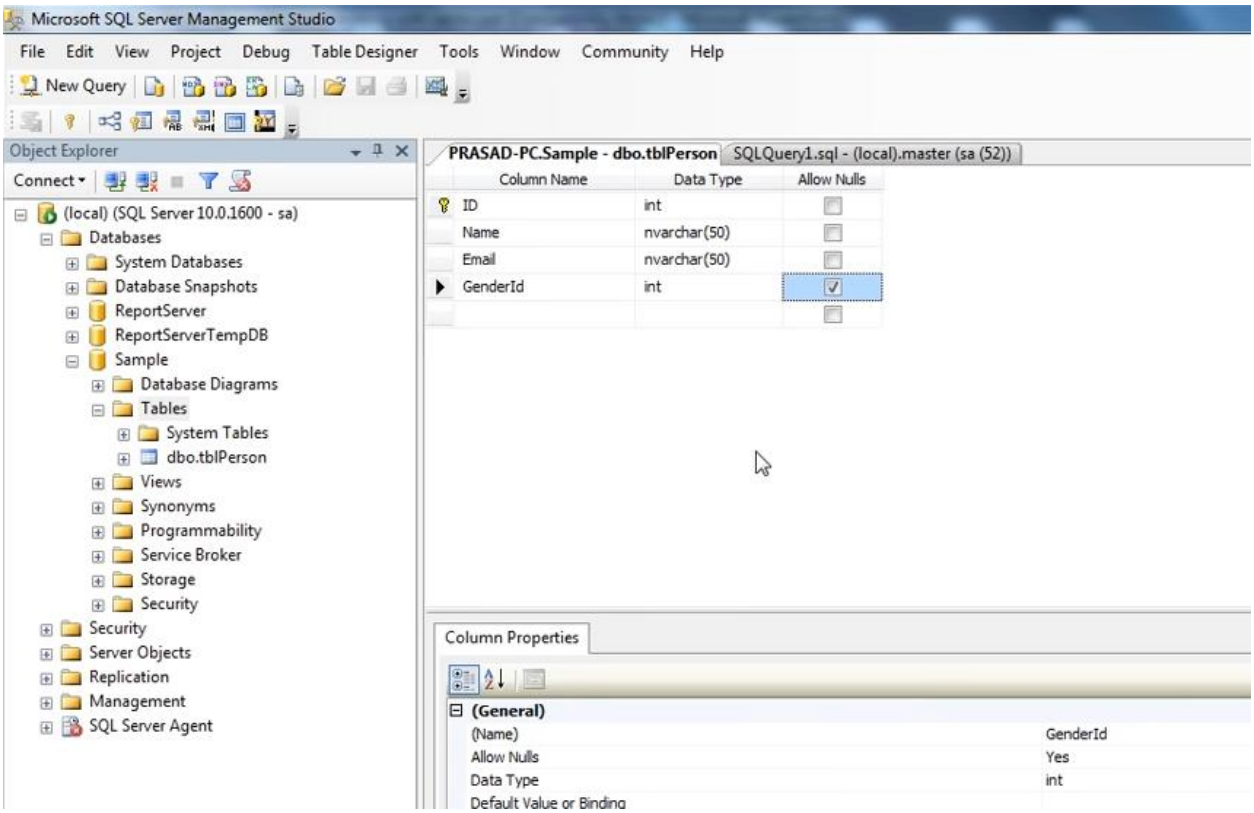
In SQL Server, tables can be created graphically using SQL Server Management Studio (SSMS) or using a query. Foreign key references can be added graphically using SSMS or using a query.

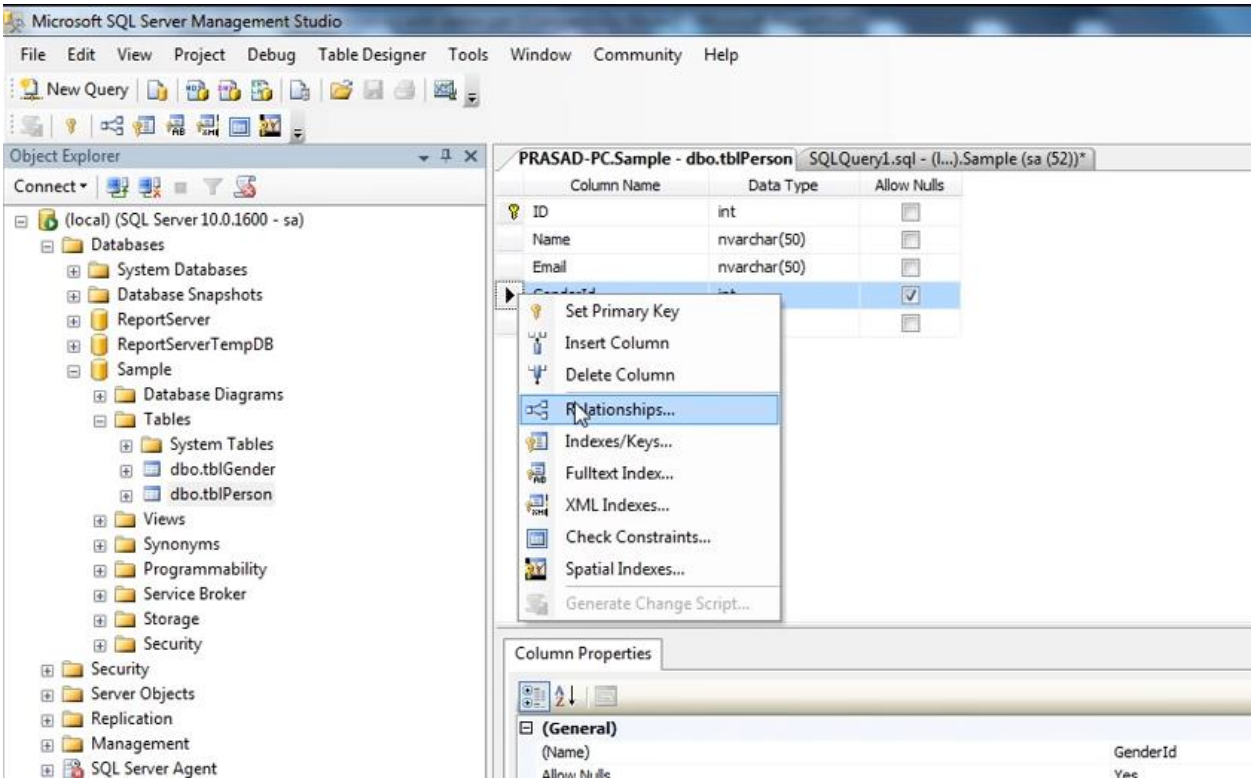
```
Alter table ForeignKeyTable add constraint ForeignKeyTable_ForiegnKeyColumn_FK
FOREIGN KEY (ForiegnKeyColumn) references PrimaryKeyTable (PrimaryKeyColumn)
```

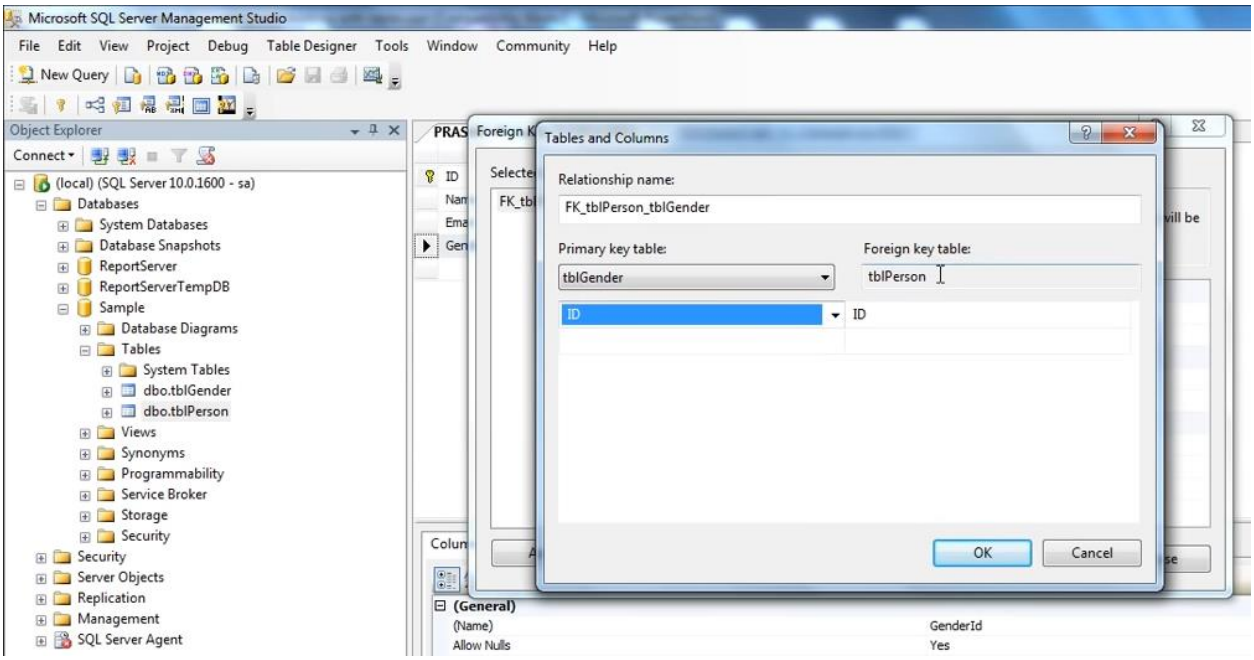
Foreign keys are used to enforce database integrity. In layman's terms, A foreign key in one table points to a primary key in another table. The foreign key constraint prevents invalid data form being inserted into the foreign key column. The values that you enter into the foreign key column, has to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

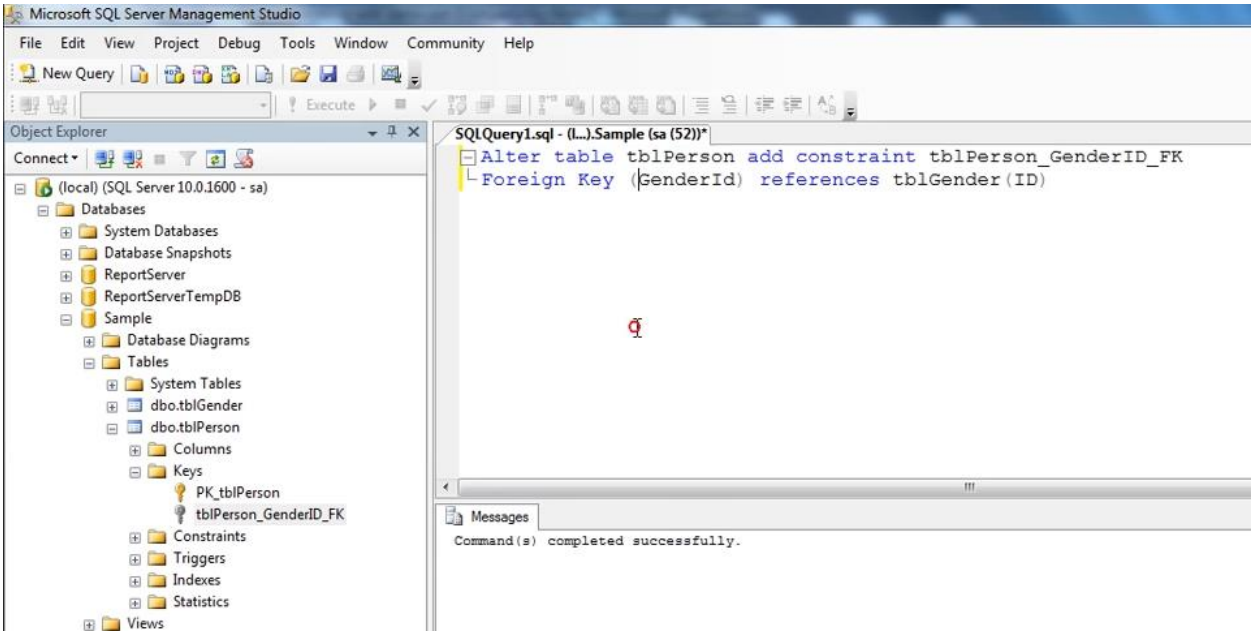


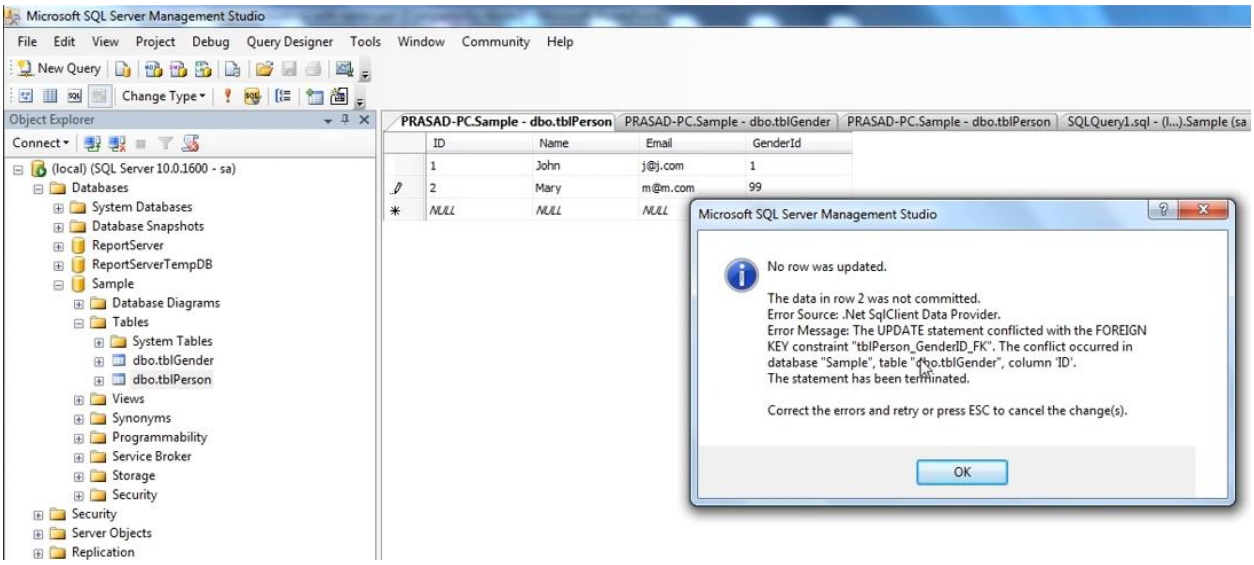

Microsoft SQL Server Management Studio
File Edit View Project Debug Tools Window Community Help
New Query
Execute
Object Explorer
Connect
SQLQuery1.sql - (local).master (sa (52))
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
New Table...
Filter
Start PowerShell
Reports
Refresh
Security
Server Objects
Replication


Microsoft SQL Server Management Studio
File Edit View Project Debug Table Designer Tools Window Community Help
New Query
Object Explorer
Connect
PRASAD-PC.Sample - dbo.tblPerson
SQLQuery1.sql - (local).master (sa (52))
Column Name
Data Type
Allow Nulls
ID
int
Name
nvarchar(50)
Email
nvarchar(50)
GenderId
int
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblPerson
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
SQL Server Agent
Column Properties
(General)
(Name)
GenderId
Allow Nulls
Yes
Data Type
int
Default Value or Binding

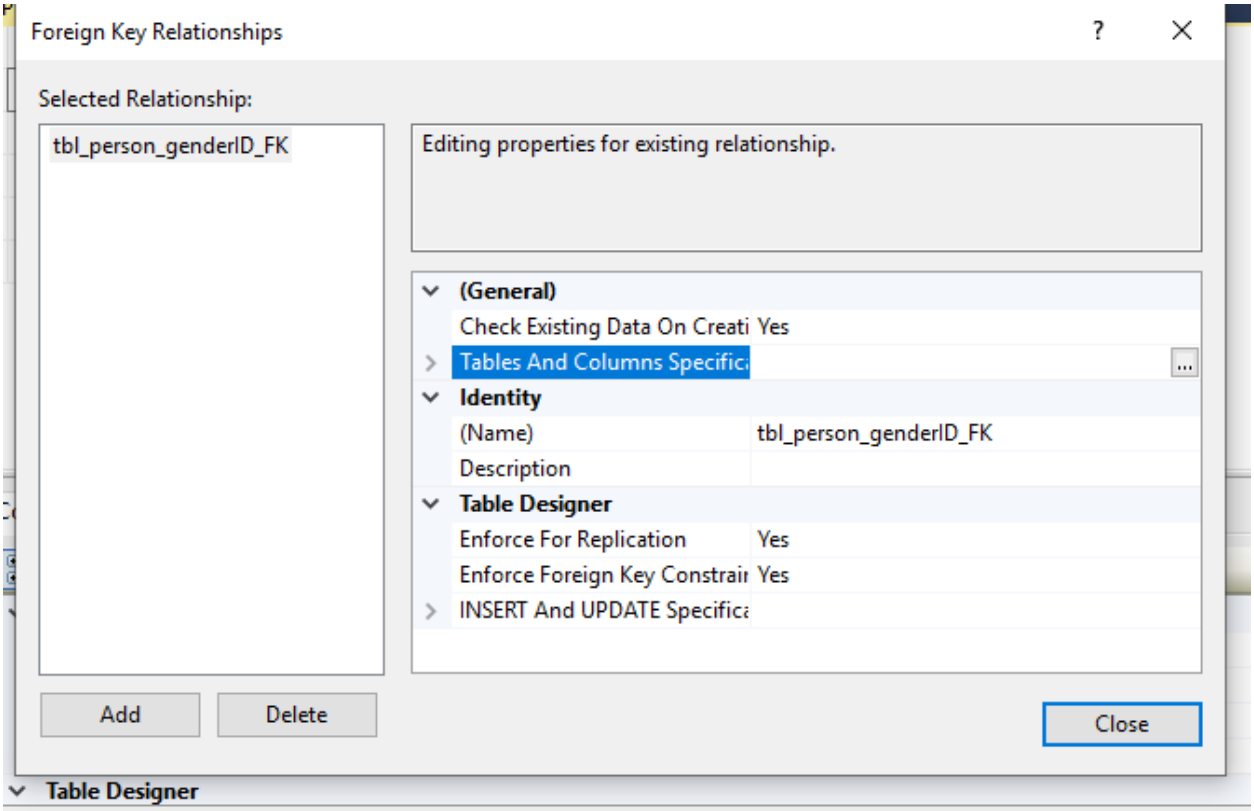

Microsoft SQL Server Management Studio
File Edit View Project Debug Table Designer Tools Window Community Help
New Query
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
SQL Server Agent
PRASAD-PC.Sample - dbo.tblPerson
SQLQuery1.sql - (l...).Sample (sa (52))*
Column Name
Data Type
Allow Nulls
ID
int
Name
nvarchar(50)
Email
nvarchar(50)
Set Primary Key
Insert Column
Delete Column
Relationships...
Indexes/Keys...
Fulltext Index...
XML Indexes...
Check Constraints...
Spatial Indexes...
Generate Change Script...
Column Properties
(General)
(Name)
GenderId

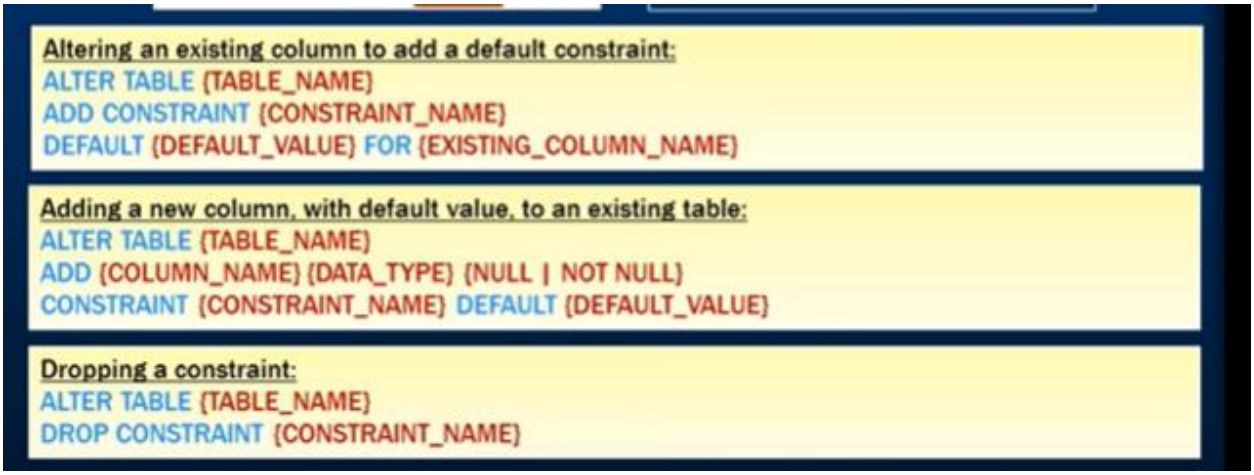

Microsoft SQL Server Management Studio
File Edit View Project Debug Table Designer Tools Window Community Help
New Query
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
SQL Server Agent
Tables and Columns
Relationship name:
FK_tblPerson_tblGender
Primary key table:
tblGender
Foreign key table:
tblPerson
ID
ID
OK
Cancel
(General)
(Name)
GenderId
Allow Nulls
Yes

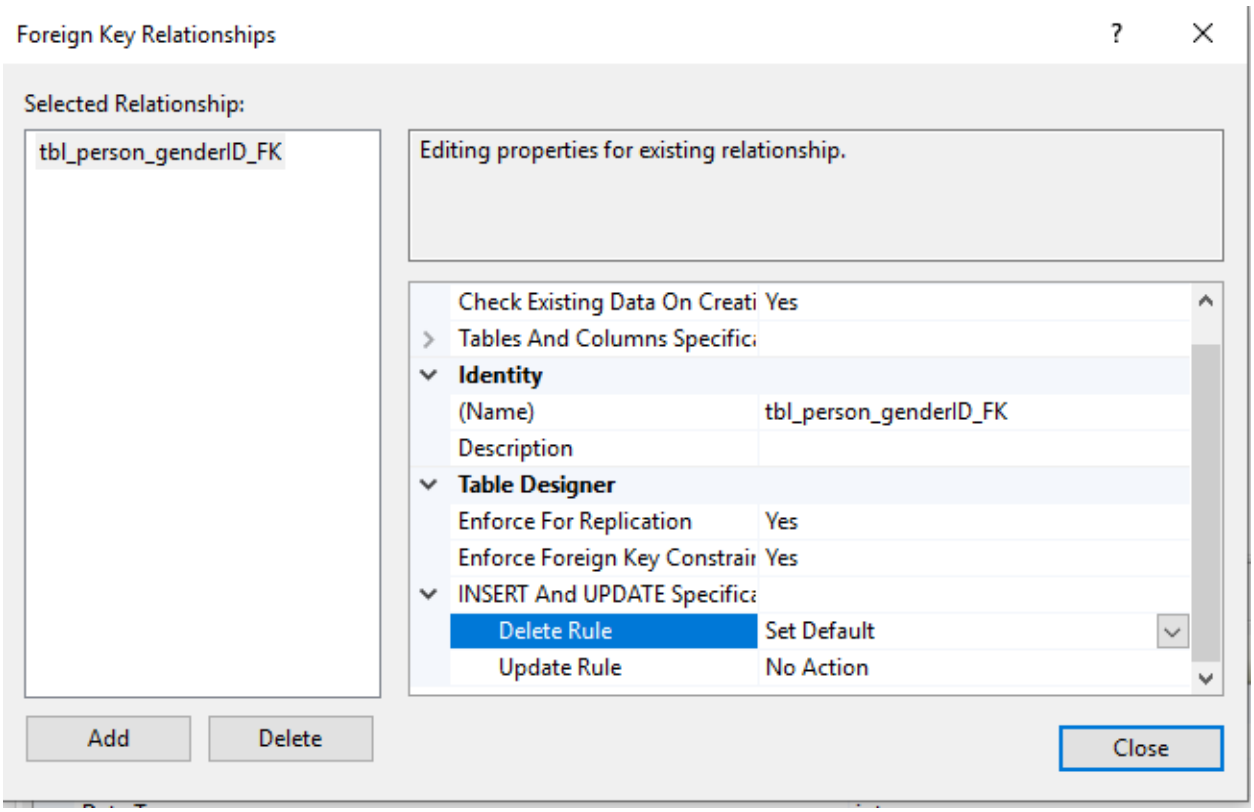




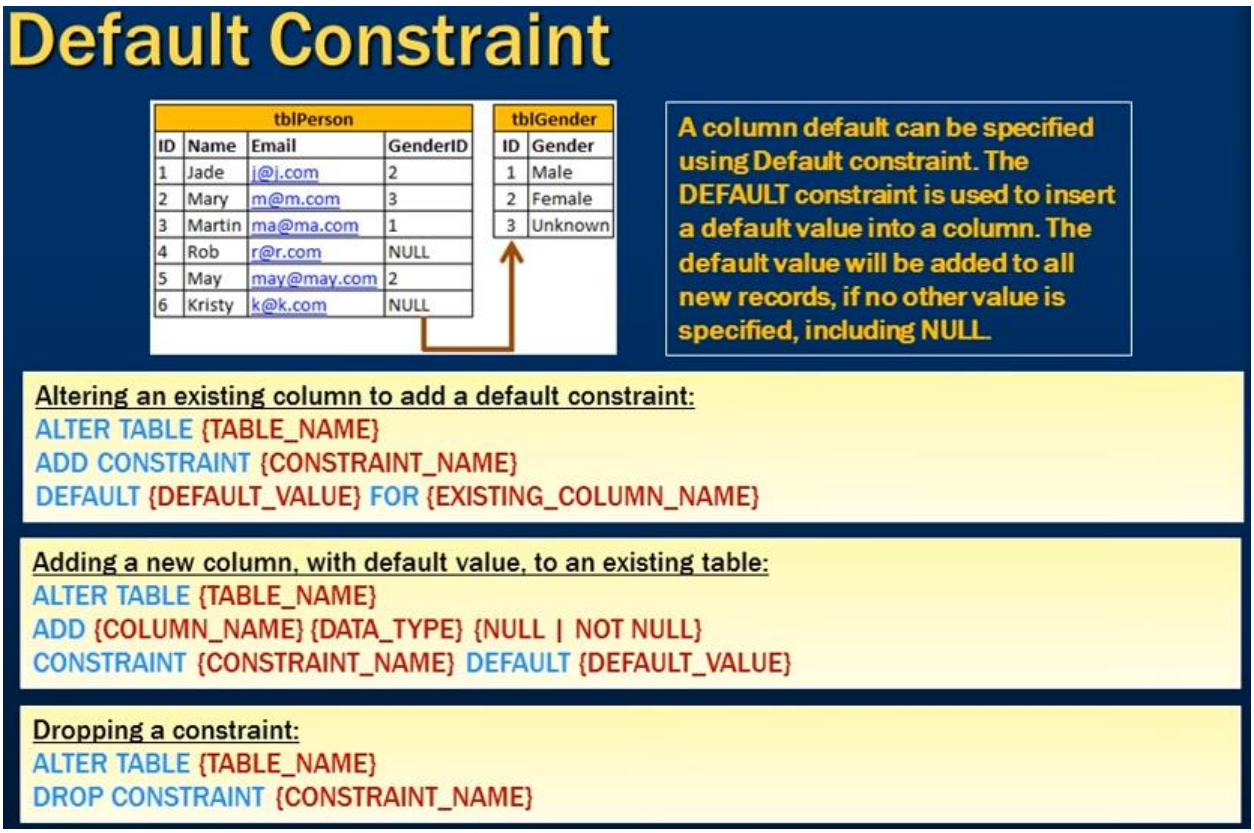
Foreign keys are used to enforce database integrity. In layman's terms, A foreign key in one table points to a primary key in another table. The foreign key constraint prevents invalid data form being inserted into the foreign key column. The values that you enter into the foreign key column, has to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

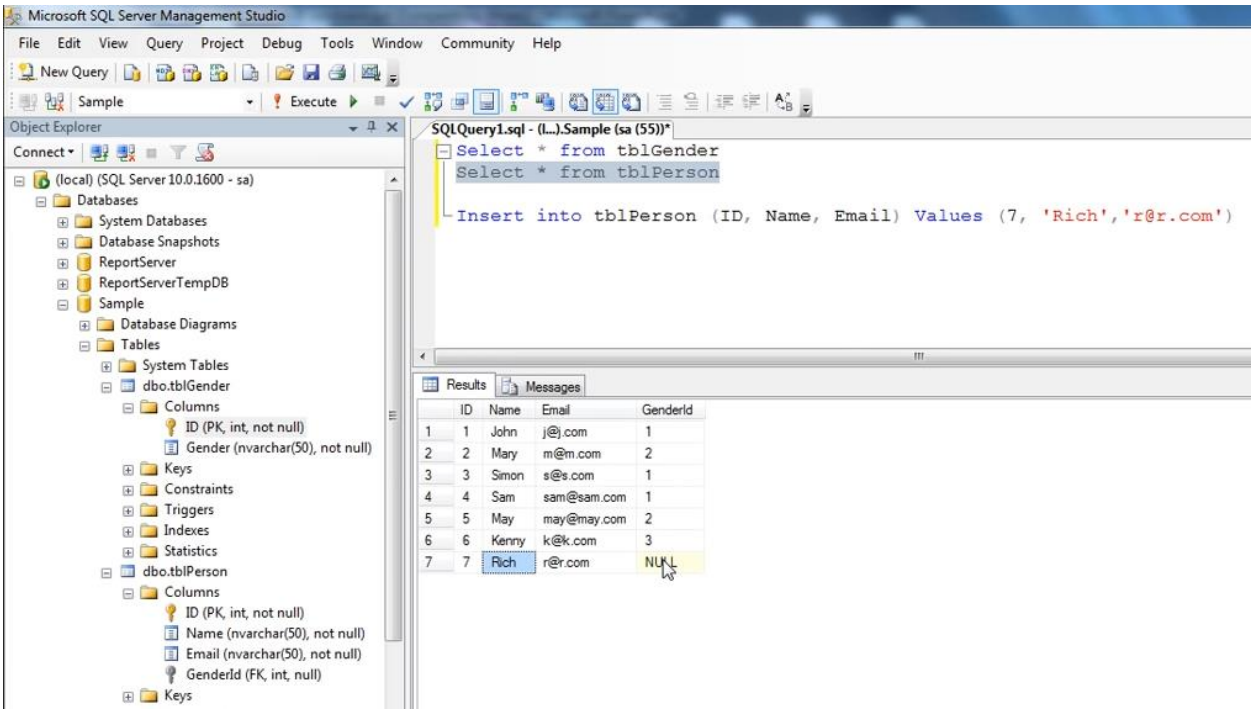

Foreign Key Relationships
Selected Relationship:
tbl_person_genderID_FK
Editing properties for existing relationship.
(General)
Check Existing Data On Creati Yes
Tables And Columns Specifica
Identity
(Name) tbl_person_genderID_FK
Description
Table Designer
Enforce For Replication Yes
Enforce Foreign Key Constrai Yes
INSERT And UPDATE Specifica
Add
Delete
Close
Table Designer

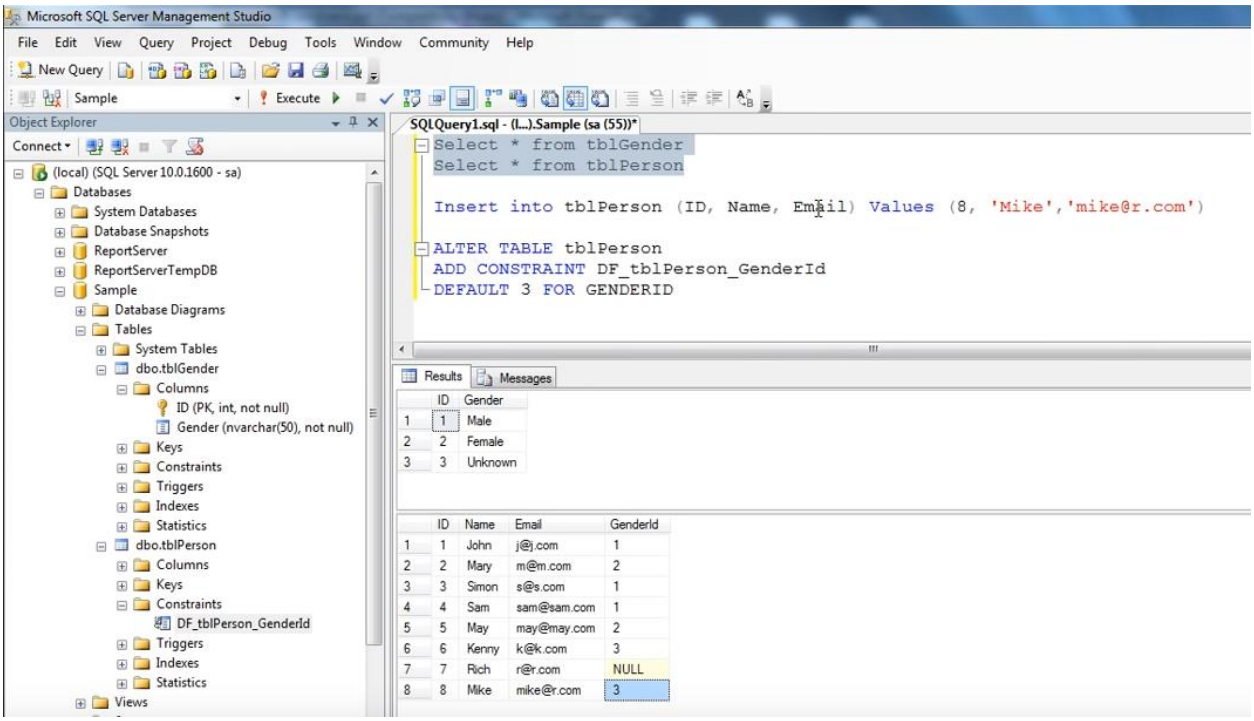

Altering an existing column to add a default constraint:
ALTER TABLE {TABLE_NAME}
ADD CONSTRAINT {CONSTRAINT_NAME}
DEFAULT {DEFAULT_VALUE} FOR {EXISTING_COLUMN_NAME}
Adding a new column, with default value, to an existing table:
ALTER TABLE {TABLE_NAME}
ADD {COLUMN_NAME} {DATA_TYPE} {NULL | NOT NULL}
CONSTRAINT {CONSTRAINT_NAME} DEFAULT {DEFAULT_VALUE}
Dropping a constraint:
ALTER TABLE {TABLE_NAME}
DROP CONSTRAINT {CONSTRAINT_NAME}

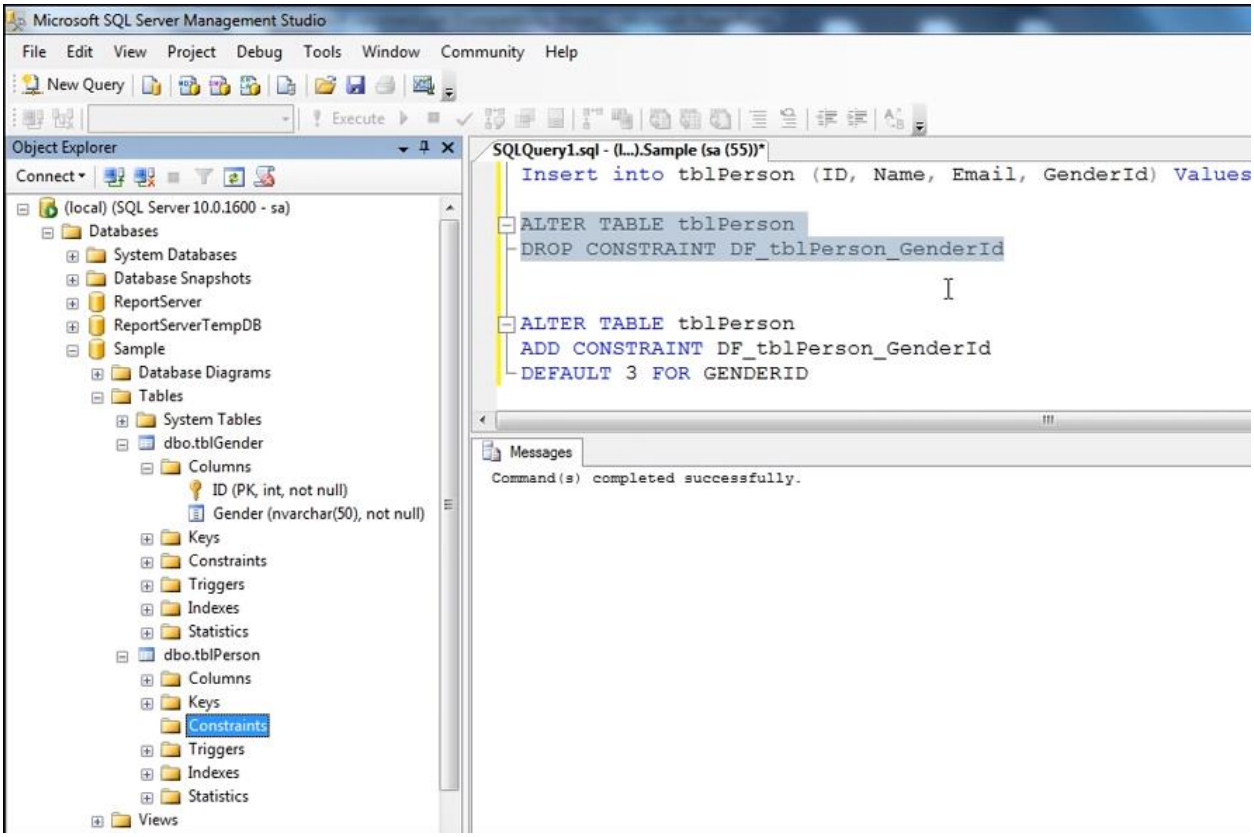


Default Constraint

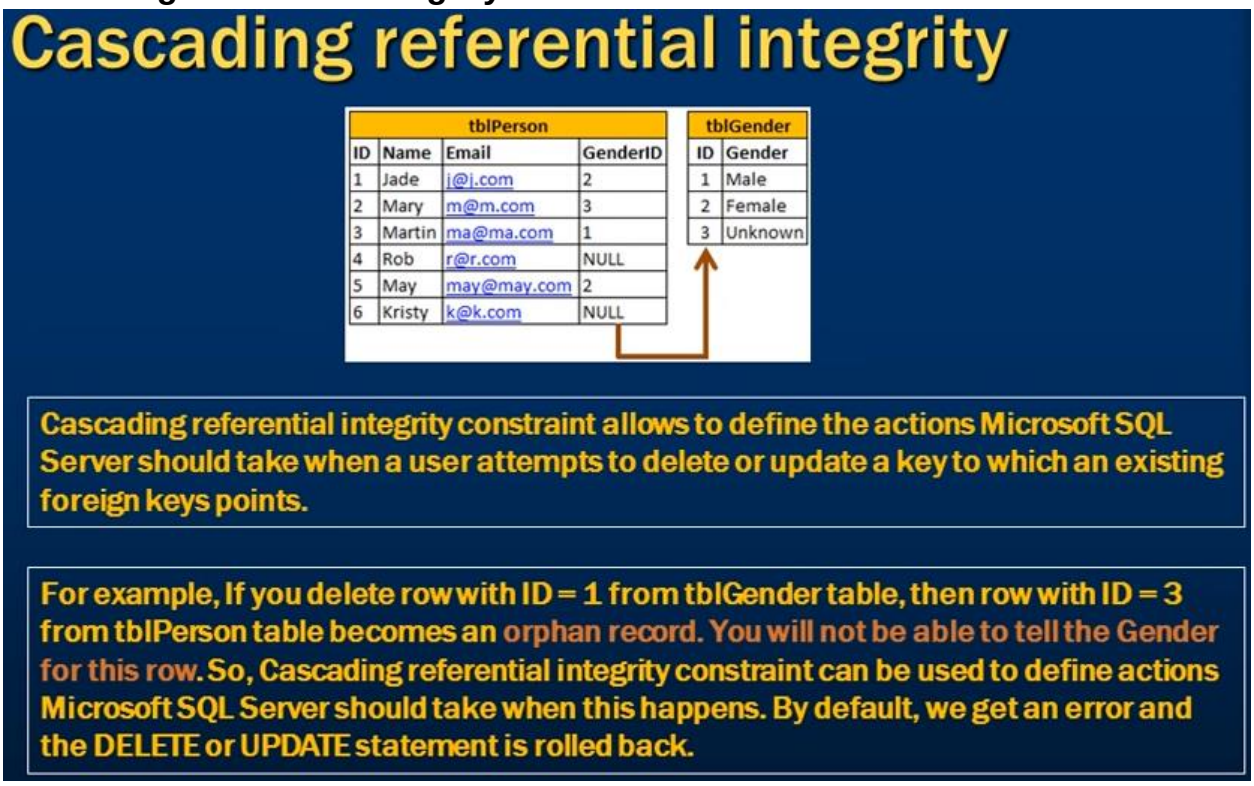


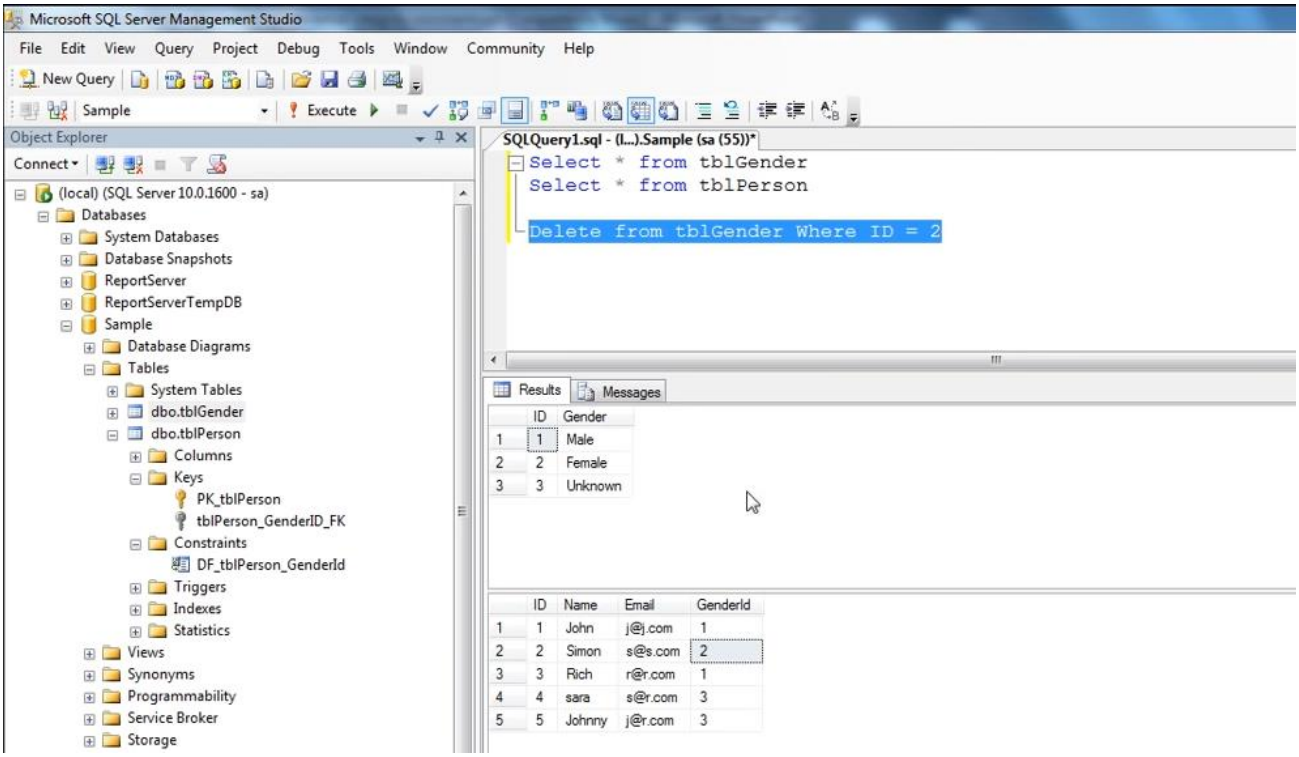

Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
Sample
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
Columns
ID (PK, int, not null)
Gender (nvarchar(50), not null)
Keys
Constraints
Triggers
Indexes
Statistics
dbo.tblPerson
Columns
ID (PK, int, not null)
Name (nvarchar(50), not null)
Email (nvarchar(50), not null)
GenderId (FK, int, null)
Keys
SQLQuery1.sql - (l...).Sample (sa (55))*
Select * from tblGender
Select * from tblPerson
Insert into tblPerson (ID, Name, Email) Values (7, 'Rich','r@r.com')
Results
Messages
ID Name Email GenderId
1 1 John j@j.com 1
2 2 Mary m@m.com 2
3 3 Simon s@s.com 1
4 4 Sam sam@sam.com 1
5 5 May may@may.com 2
6 6 Kenny k@k.com 3
7 7 Rich r@r.com NULL

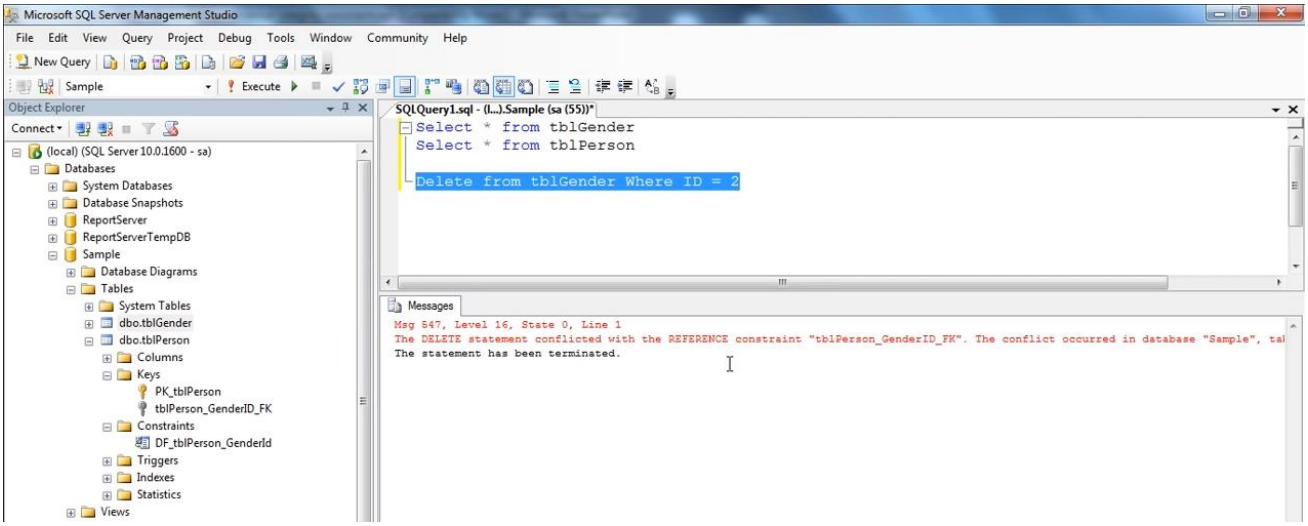

Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
Sample
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
Columns
ID (PK, int, not null)
Gender (nvarchar(50), not null)
Keys
Constraints
Triggers
Indexes
Statistics
dbo.tblPerson
Columns
Keys
Constraints
DF_tblPerson_GenderId
Triggers
Indexes
Statistics
Views
SQLQuery1.sql - (l...).Sample (sa (55))*
Select * from tblGender
Select * from tblPerson
Insert into tblPerson (ID, Name, Email) Values (8, 'Mike','mike@r.com')
ALTER TABLE tblPerson
ADD CONSTRAINT DF_tblPerson_GenderId
DEFAULT 3 FOR GENDERID
Results
Messages
ID Gender
1 1 Male
2 2 Female
3 3 Unknown
ID Name Email GenderId
1 1 John j@j.com 1
2 2 Mary m@m.com 2
3 3 Simon s@s.com 1
4 4 Sam sam@sam.com 1
5 5 May may@may.com 2
6 6 Kenny k@k.com 3
7 7 Rich r@r.com NULL
8 8 Mike mike@r.com 3



Cascading referential integrity constraint




Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
Sample
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Columns
Keys
PK_tblPerson
tblPerson_GenderID_FK
Constraints
DF_tblPerson_GenderId
Triggers
Indexes
Statistics
Views
Synonyms
Programmability
Service Broker
Storage
SQLQuery1.sql - (l...).Sample (sa (55))*
Select * from tblGender
Select * from tblPerson
Delete from tblGender Where ID = 2
Results
Messages
ID Gender
1 1 Male
2 2 Female
3 3 Unknown
ID Name Email GenderId
1 1 John j@j.com 1
2 2 Simon s@s.com 2
3 3 Rich r@r.com 1
4 4 sara s@r.com 3
5 5 Johnny j@r.com 3


Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
Sample
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Columns
Keys
PK_tblPerson
tblPerson_GenderID_FK
Constraints
DF_tblPerson_GenderId
Triggers
Indexes
Statistics
Views
SQLQuery1.sql - (l...).Sample (sa (55))*
Select * from tblGender
Select * from tblPerson
Delete from tblGender Where ID = 2
Messages
Msg 547, Level 16, State 0, Line 1
The DELETE statement conflicted with the REFERENCE constraint "tblPerson_GenderID_FK". The conflict occurred in database "Sample", tal
The statement has been terminated.

Cascading referential integrity

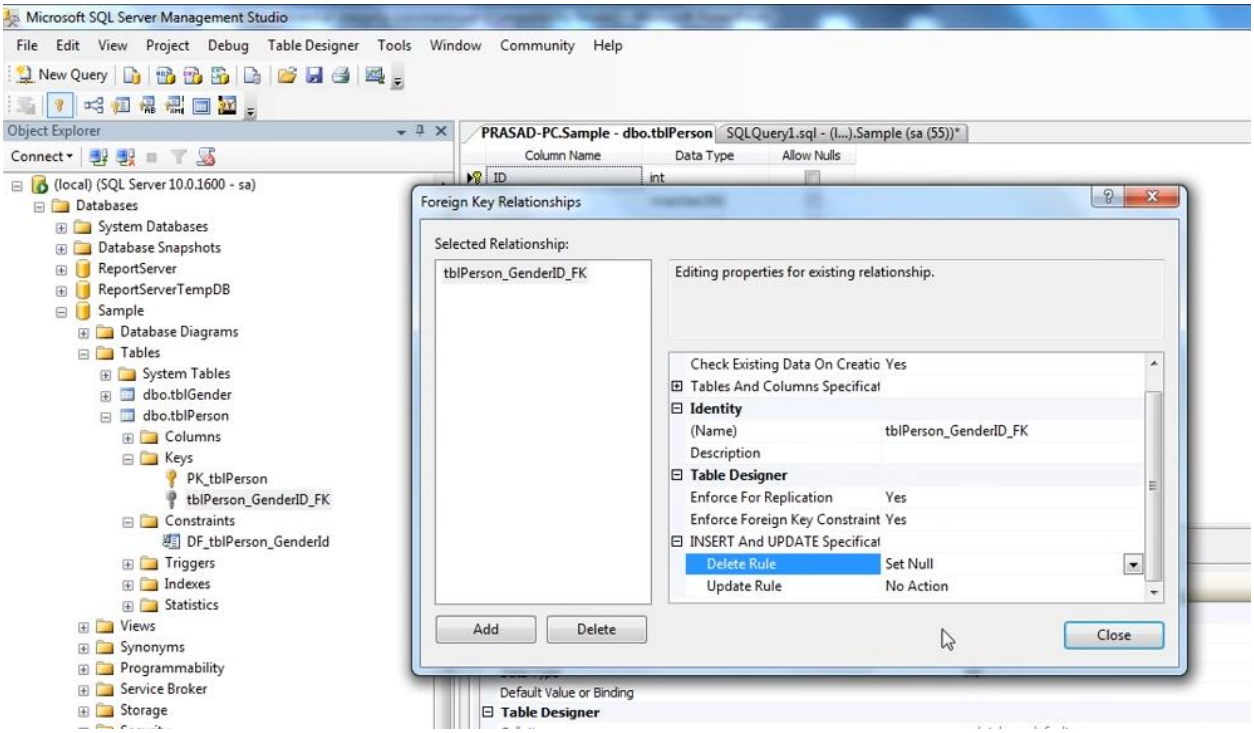
Options when setting up Cascading referential integrity constraint:

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.

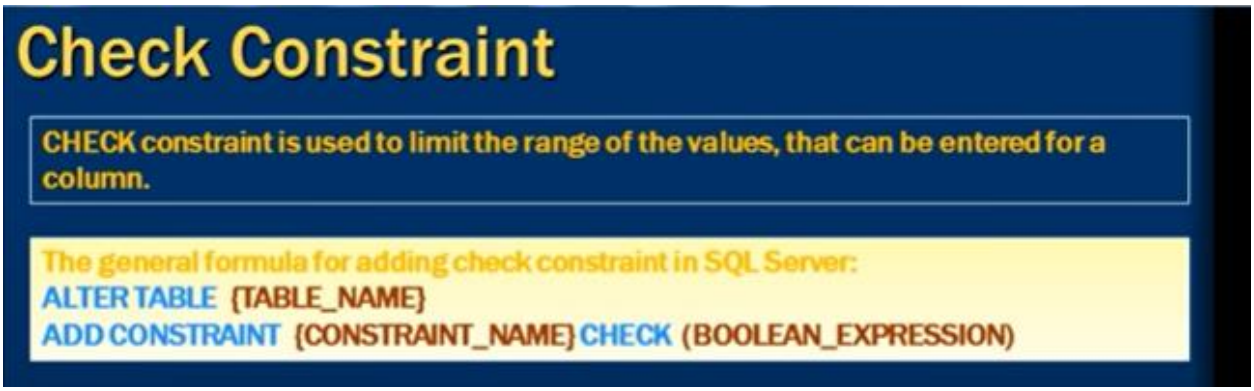
3. Set NULL: Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to NULL.

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.

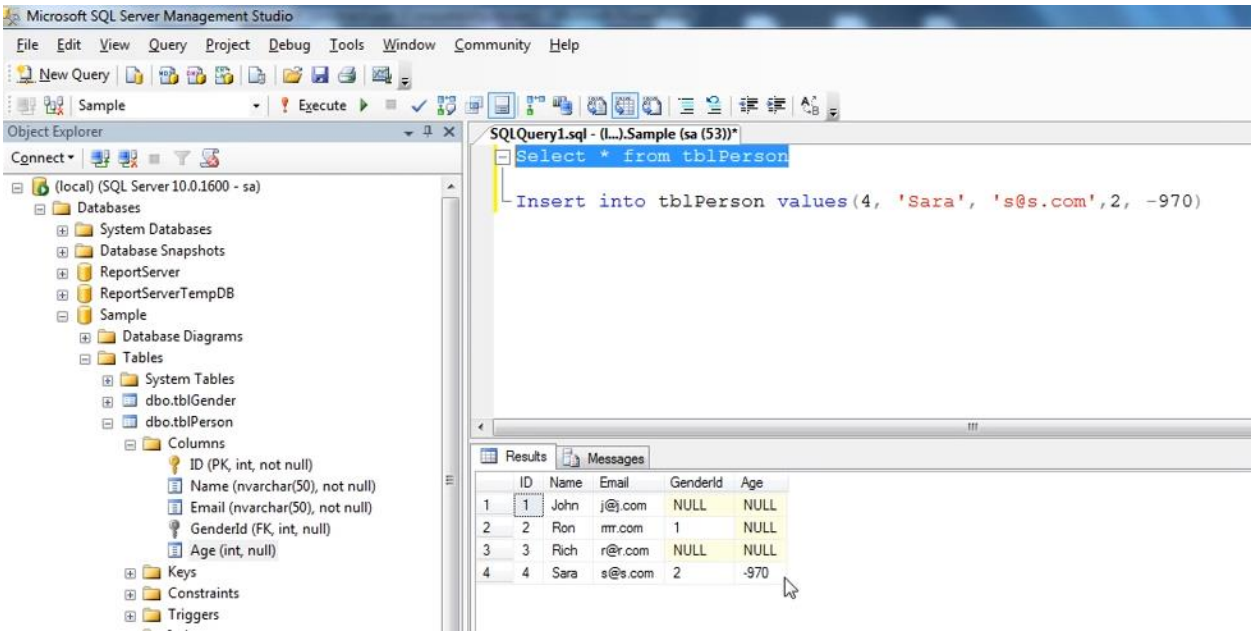


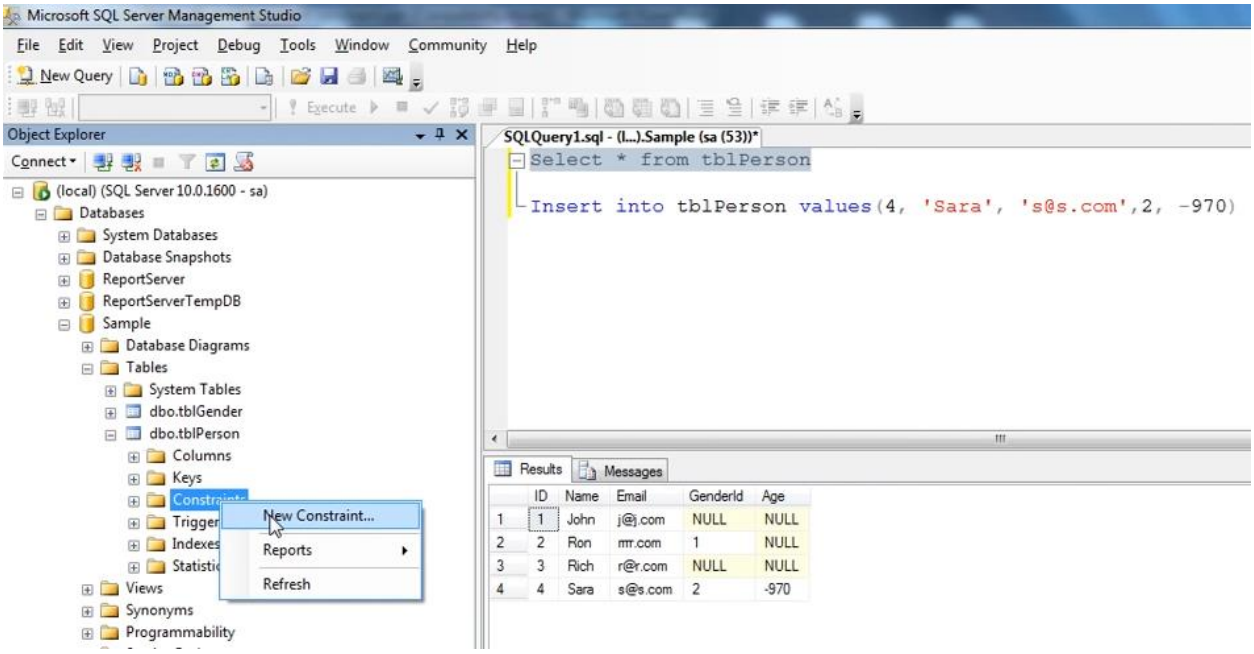
Check Constraint

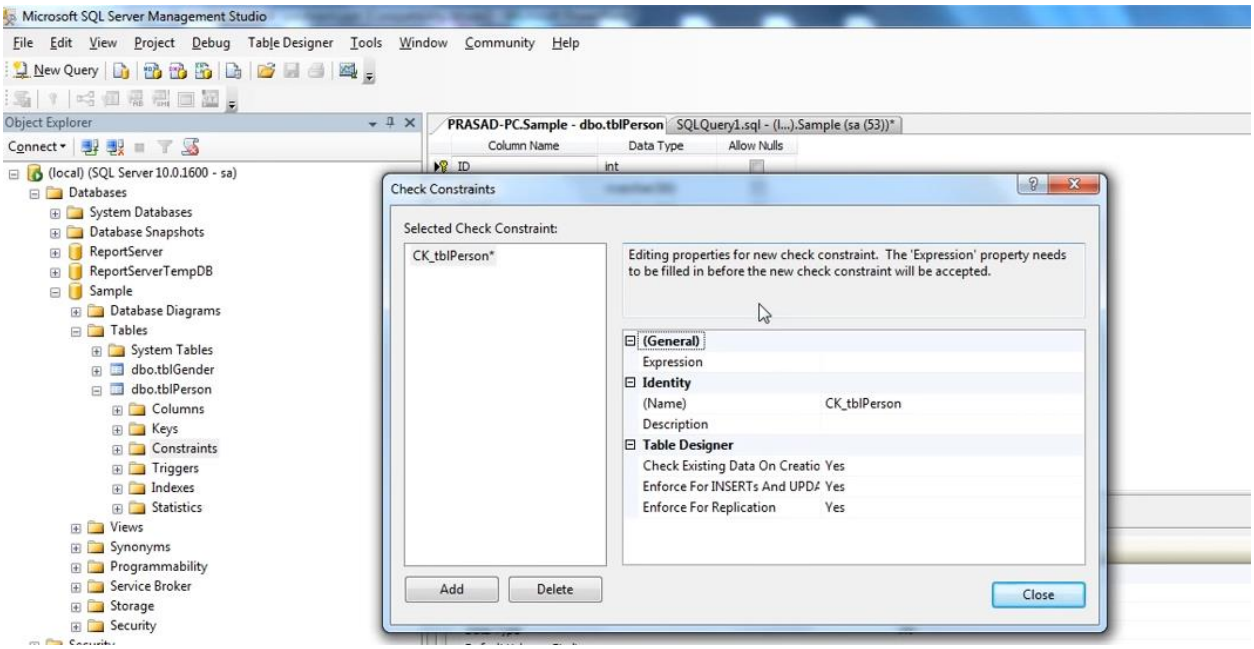
The **CHECK constraint** is used to limit the value range that can be placed in a column. If you define a **CHECK constraint** on a single column it allows only certain values for this column. If you define a **CHECK constraint** on a table it can limit the values in certain columns based on values in other columns in the row.

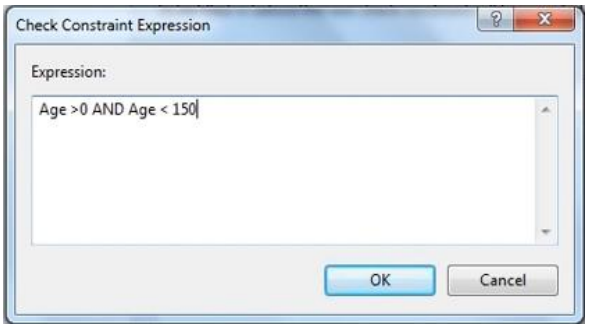


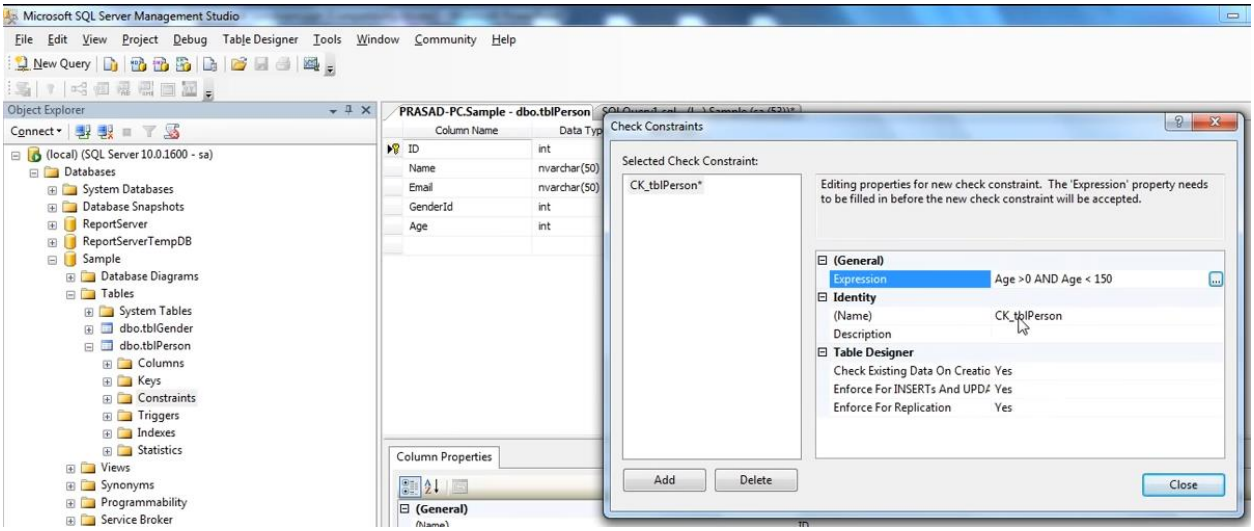
In the following statement Age is entered as -970, we prevent such mistakes using check constraint.

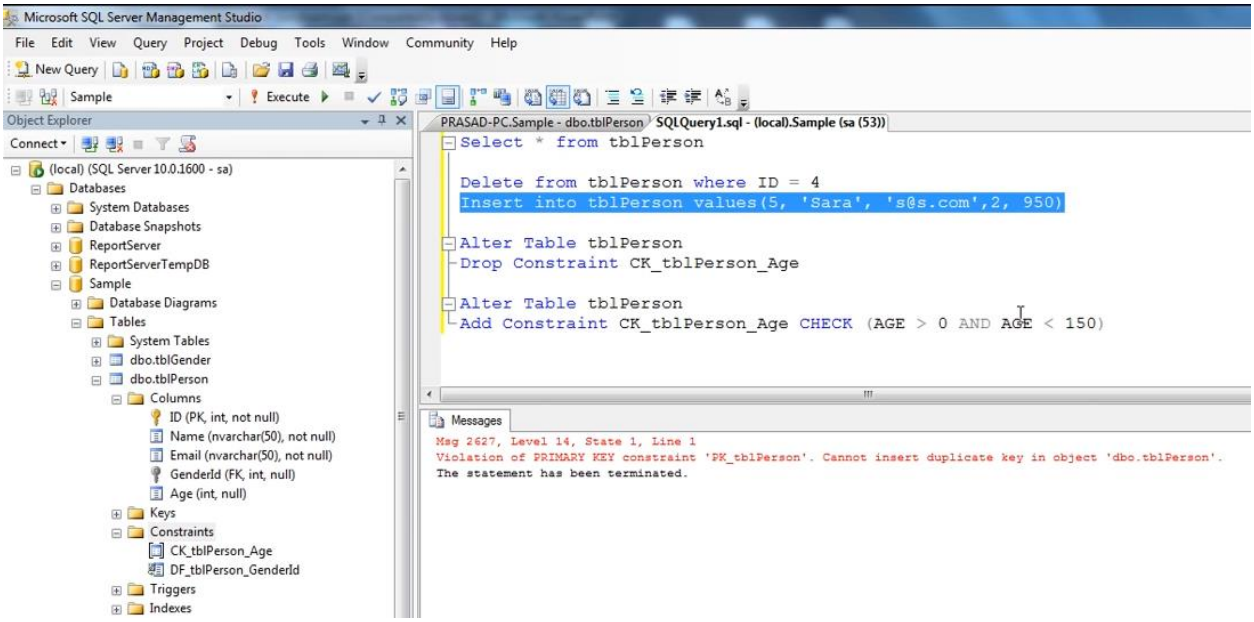


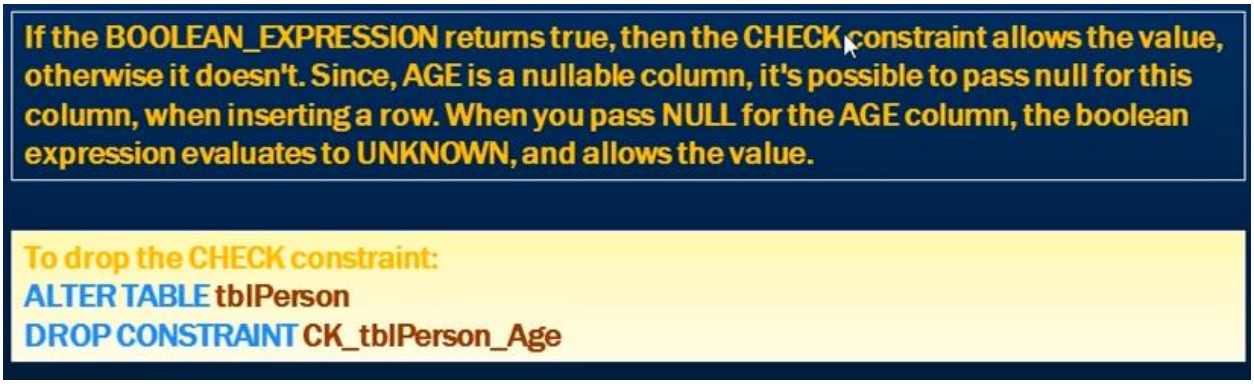

Microsoft SQL Server Management Studio
File Edit View Project Debug Tools Window Community Help
New Query
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Columns
Keys
Constraints
Triggers
Indexes
Statistics
Views
Synonyms
Programmability
New Constraint...
Reports
Refresh
SQLQuery1.sql - (l...).Sample (sa (53))*
Select * from tblPerson
Insert into tblPerson values(4, 'Sara', 's@s.com',2, -970)
Results
Messages
ID Name Email GenderId Age
1 1 John j@j.com NULL NULL
2 2 Ron mm.com 1 NULL
3 3 Rich r@r.com NULL NULL
4 4 Sara s@s.com 2 -970


Microsoft SQL Server Management Studio
File Edit View Project Debug Table Designer Tools Window Community Help
New Query
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Columns
Keys
Constraints
Triggers
Indexes
Statistics
Views
Synonyms
Programmability
Service Broker
Storage
Security
PRASAD-PC.Sample - dbo.tblPerson
SQLQuery1.sql - (l...).Sample (sa (53))*
Column Name Data Type Allow Nulls
ID int
Check Constraints
Selected Check Constraint:
CK_tblPerson*
Editing properties for new check constraint. The 'Expression' property needs to be filled in before the new check constraint will be accepted.
(General)
Expression
Identity
(Name) CK_tblPerson
Description
Table Designer
Check Existing Data On Creatio Yes
Enforce For INSERTs And UPDA Yes
Enforce For Replication Yes
Add
Delete
Close


Check Constraint Expression
Expression:
Age >0 AND Age < 150
OK
Cancel


Microsoft SQL Server Management Studio
File Edit View Project Debug Table Designer Tools Window Community Help
New Query
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Columns
Keys
Constraints
Triggers
Indexes
Statistics
Views
Synonyms
Programmability
Service Broker
PRASAD-PC.Sample - dbo.tblPerson
Column Name
ID int
Name nvarchar(50)
Email nvarchar(50)
GenderId int
Age int
Column Properties
(General)
Check Constraints
Selected Check Constraint:
CK_tblPerson*
Editing properties for new check constraint. The 'Expression' property needs to be filled in before the new check constraint will be accepted.
(General)
Expression Age >0 AND Age < 150
Identity
(Name) CK_tblPerson
Description
Table Designer
Check Existing Data On Creatio Yes
Enforce For INSERTs And UPDA Yes
Enforce For Replication Yes
Add
Delete
Close


Microsoft SQL Server Management Studio
File Edit View Query Project Debug Tools Window Community Help
New Query
Sample
Execute
Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblGender
dbo.tblPerson
Columns
ID (PK, int, not null)
Name (nvarchar(50), not null)
Email (nvarchar(50), not null)
GenderId (FK, int, null)
Age (int, null)
Keys
Constraints
CK_tblPerson_Age
DF_tblPerson_GenderId
Triggers
Indexes
PRASAD-PC.Sample - dbo.tblPerson
SQLQuery1.sql - (local).Sample (sa (53))
Select * from tblPerson
Delete from tblPerson where ID = 4
Insert into tblPerson values(5, 'Sara', 's@s.com',2, 950)
Alter Table tblPerson
Drop Constraint CK_tblPerson_Age
Alter Table tblPerson
Add Constraint CK_tblPerson_Age CHECK (AGE > 0 AND AGE < 150)
Messages
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK_tblPerson'. Cannot insert duplicate key in object 'dbo.tblPerson'.
The statement has been terminated.


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

- What is Identity column
- Creating an Identity column
- Supply explicit values for identity columns using IDENTITY_INSERT
- Reset identity value using DBCC CHECKIDENT command

Identity column of a table is a column whose value increases automatically. The value in an identity column is created by the server. A user generally cannot insert a value into an identity column. Identity column can be used to uniquely identify the rows in the table.

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.

```
Create Table tblPerson
(
   PersonId int Identity(1,1) Primary Key,
   Name nvarchar(20)
)
```

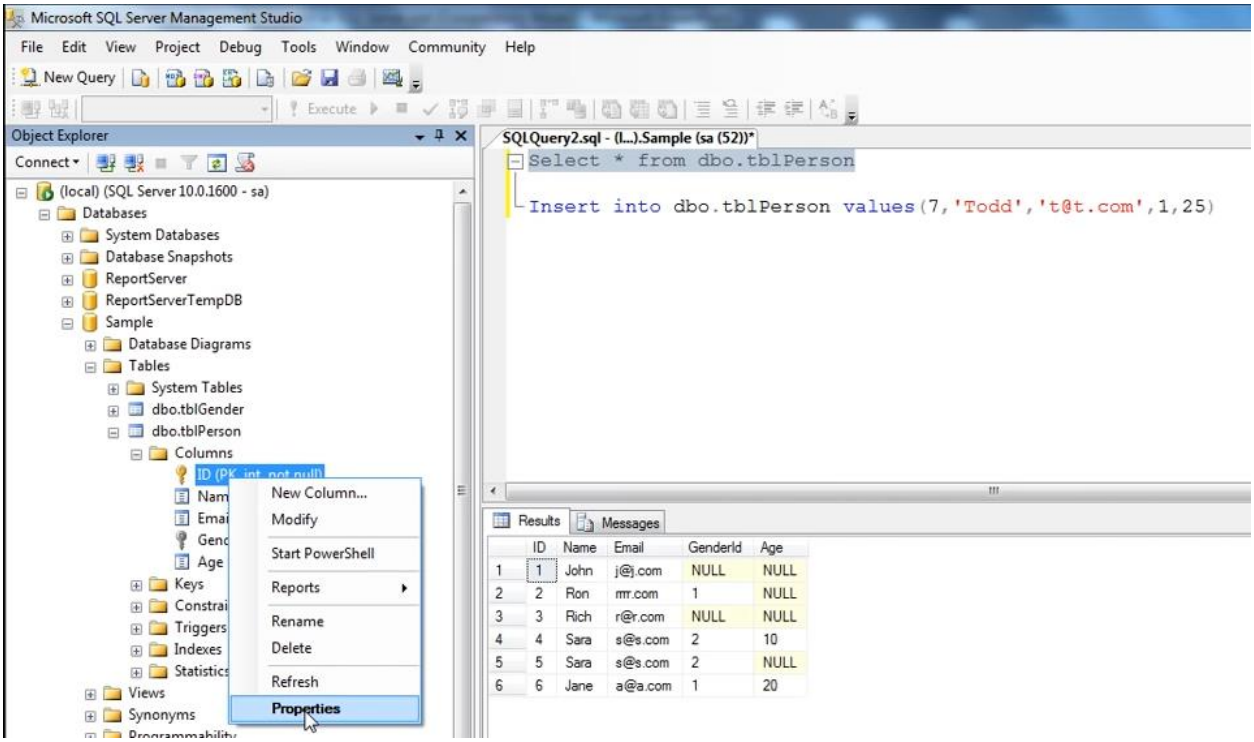
Note: Seed and Increment values are optional. If you don't specify the identity and seed they both default to 1.

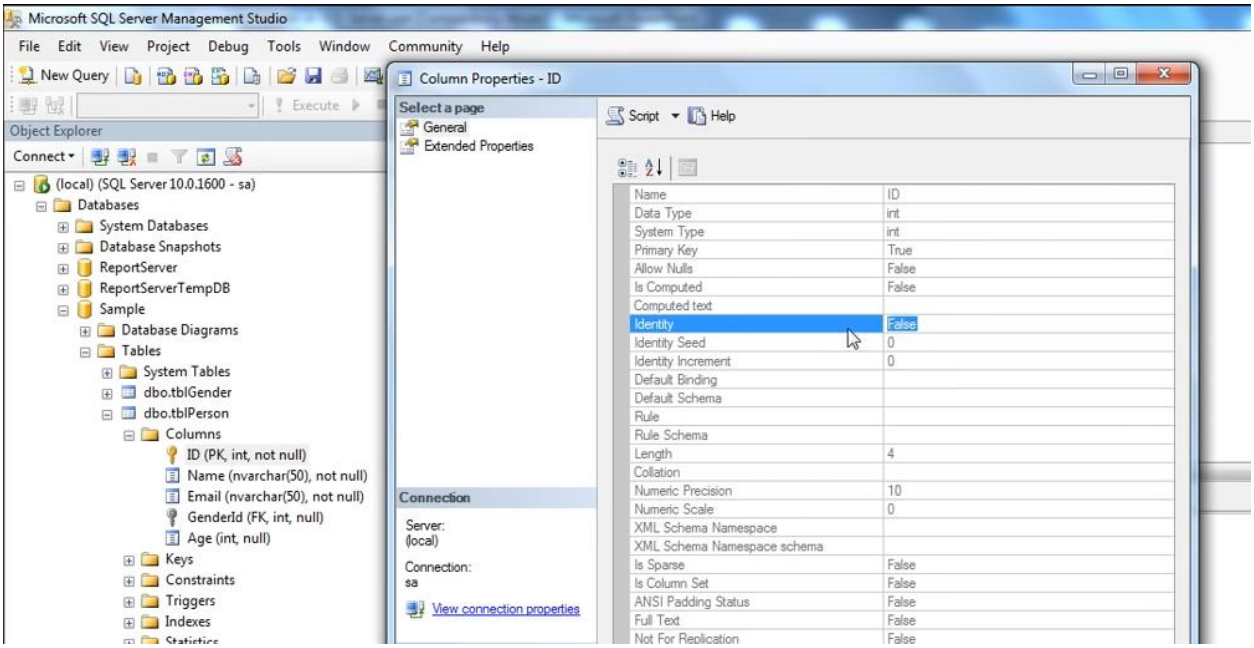
To explicitly supply a value for identity column
1. First turn on identity insert - SET Identity_Insert tblPerson ON
2. In the insert query specify the column list
   Insert into tblPerson(PersonId, Name) values(2, 'John')

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

```
DBCC CHECKIDENT('tblPerson', RESEED, 0)
```

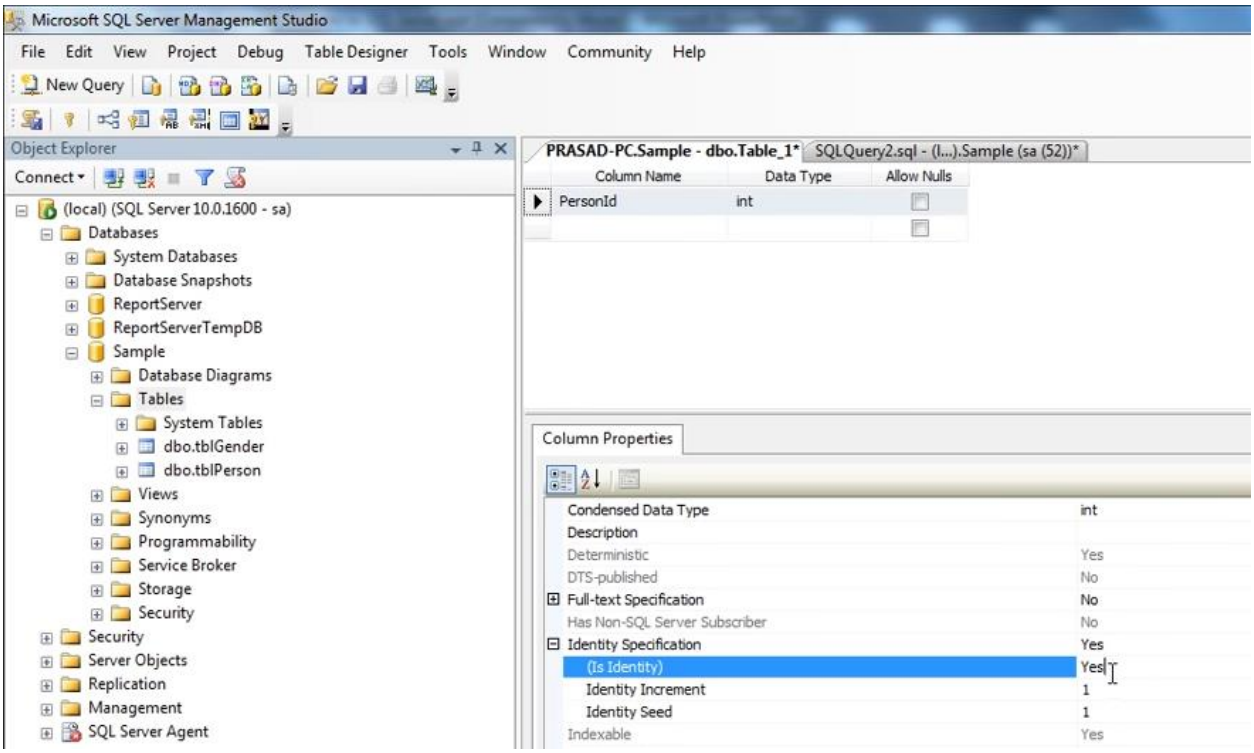
How to check if a column is an Identity Column?



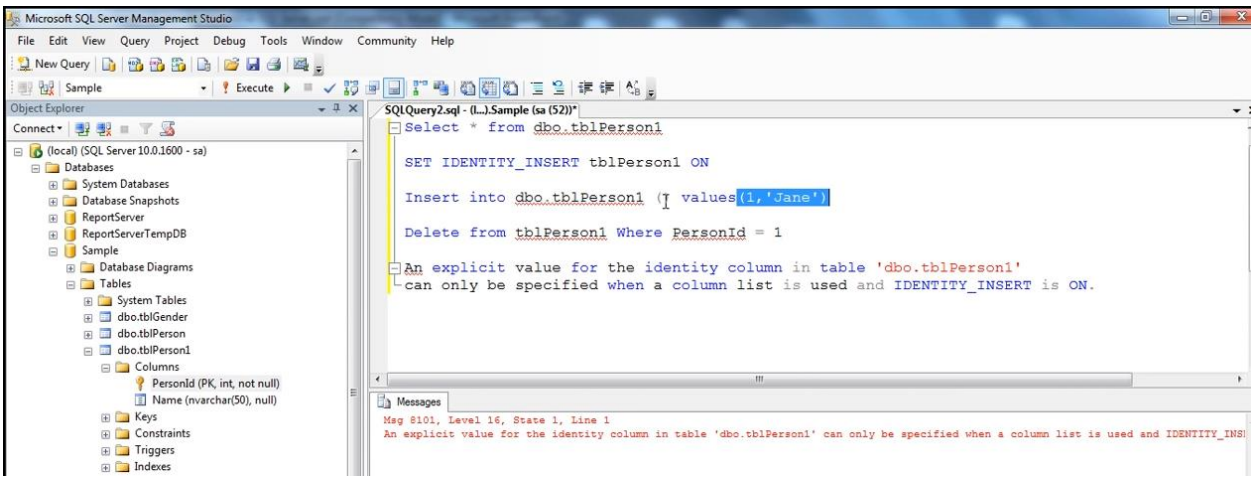


Since ID is not an Identity Column, we will have to supply the value of ID every time we need to insert a new record in this table.

How to mark an column as Identity Column? At the time of table creation, when a column is added set its property 'Is Identity' to Yes.



After Is Identity property is set to Yes, there is no need to supply value for this column when entering a record. However, if one wishes to supply an explicit value to an Identity column then IDENTITY_INSERT has to be set to ON.



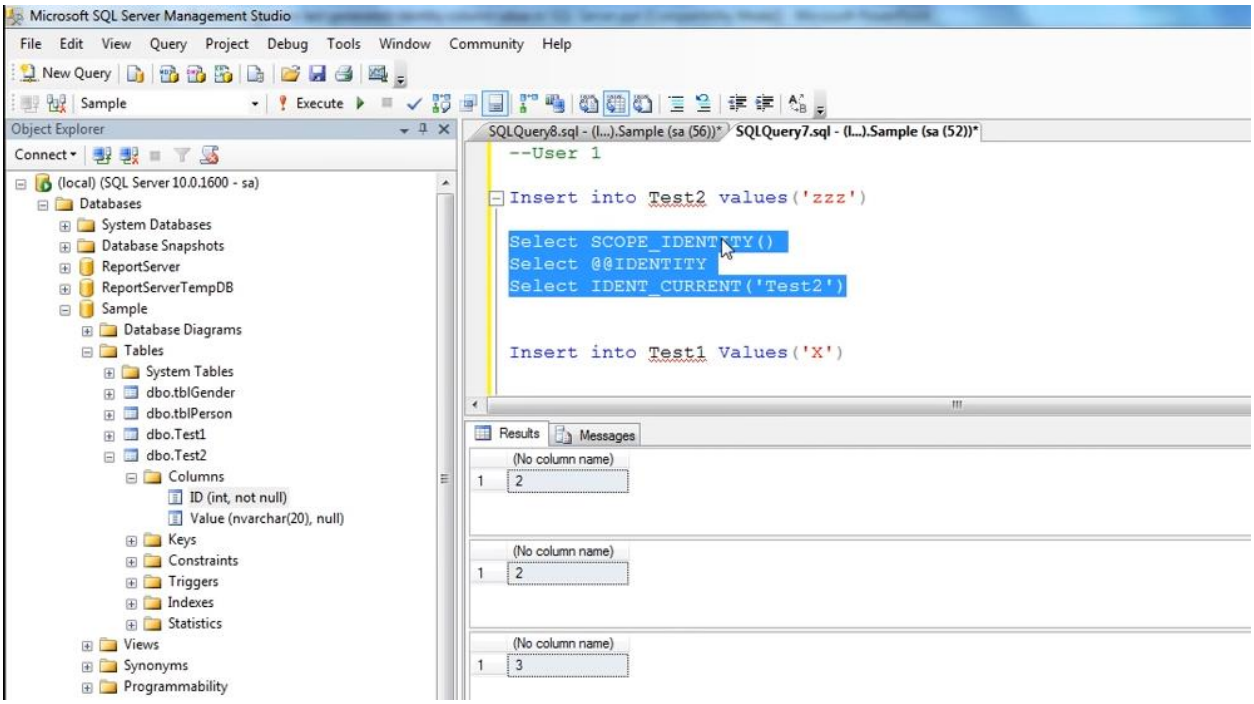
Retrieve Identity Column Value

- How to get the last generated identity column value
- Difference between SCOPE_IDENTITY(), @@Identity and IDENT_CURRENT('TableName')

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.

Note: You can also use @@IDENTITY and IDENT_CURRENT('TableName')

Difference:
SCOPE_IDENTITY() - Same session and the same scope.
@@IDENTITY - Same session and across any scope.
IDENT_CURRENT('TableName') - Specific table across any session and any scope.




```
Select IDENT_CURRENT('Test2') -- returns the last generated identity value in any session and any scope
Select SCOPE_IDENTITY() --returns the last identity value generated in the same scope and session
Select @@IDENTITY -- returns the last identity value generated in the same session but any scope
```

Unique key constraint

- What is unique key constraint
- When do you use unique key constraint over primary key constraint
- Difference between unique key constraint and primary key constraint

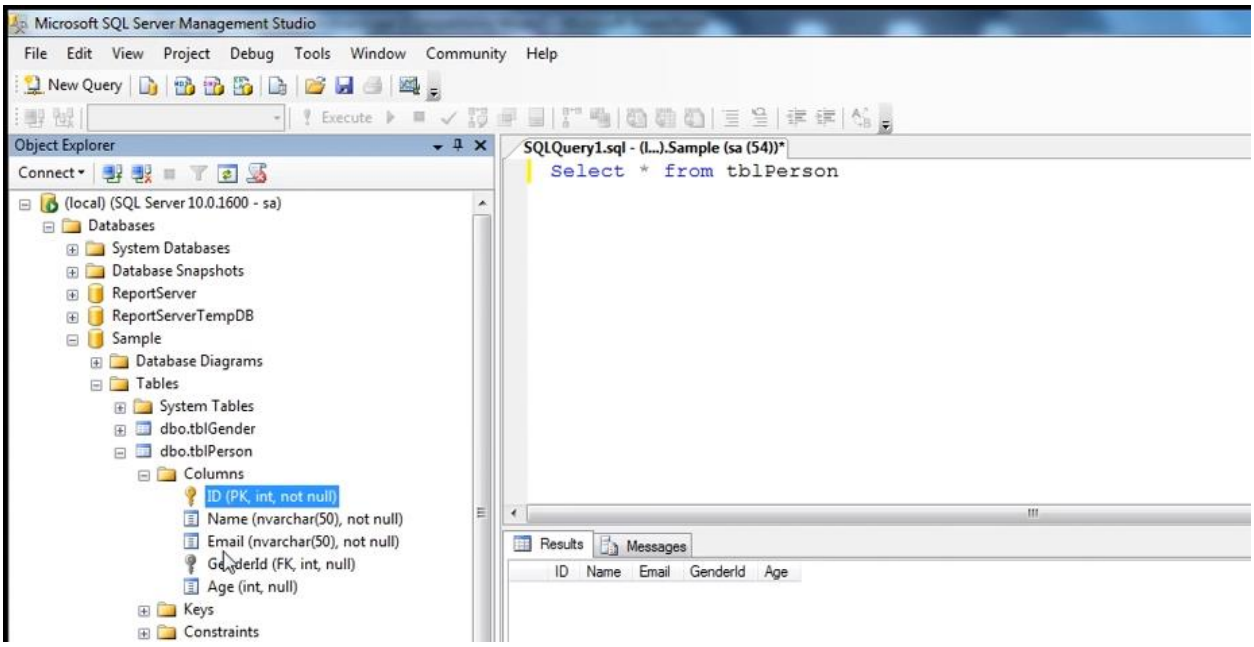
We use UNIQUE constraint to enforce uniqueness of a column i.e the column shouldn't allow any duplicate values. We can add a Unique constraint thru the designer or using a query.

To create the unique key using a query:
Alter Table Table_Name
Add Constraint Constraint_Name Unique(Column_Name)

Bot 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?
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

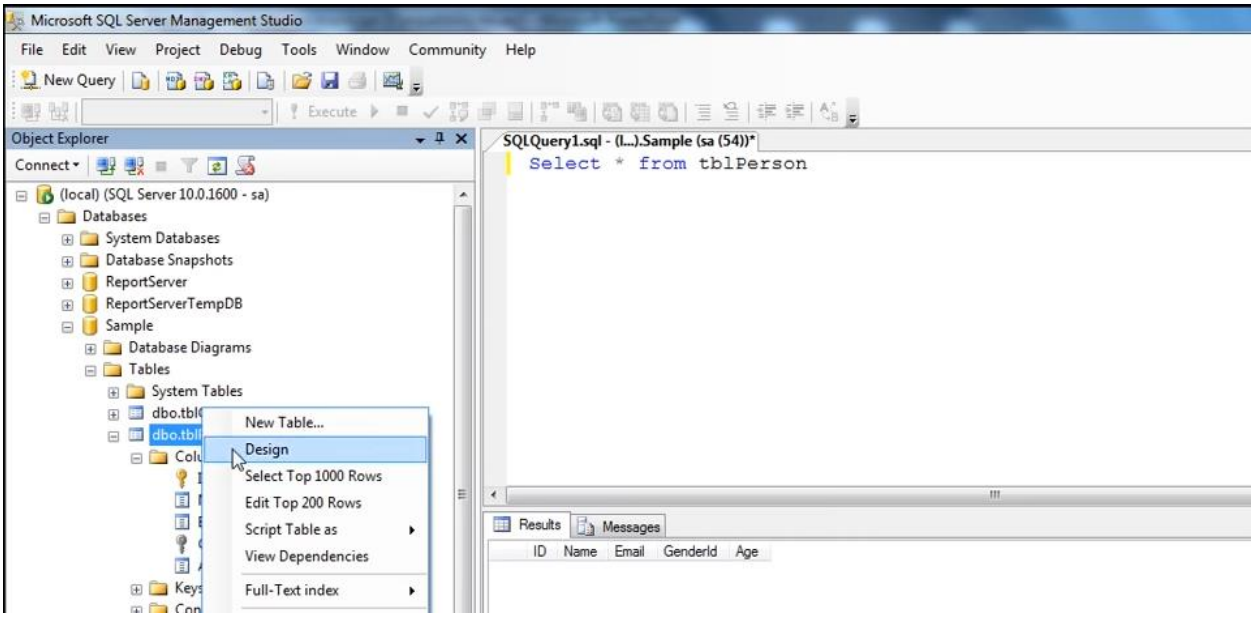
In the following table we have a primary key and we cannot mark another column as primary key.



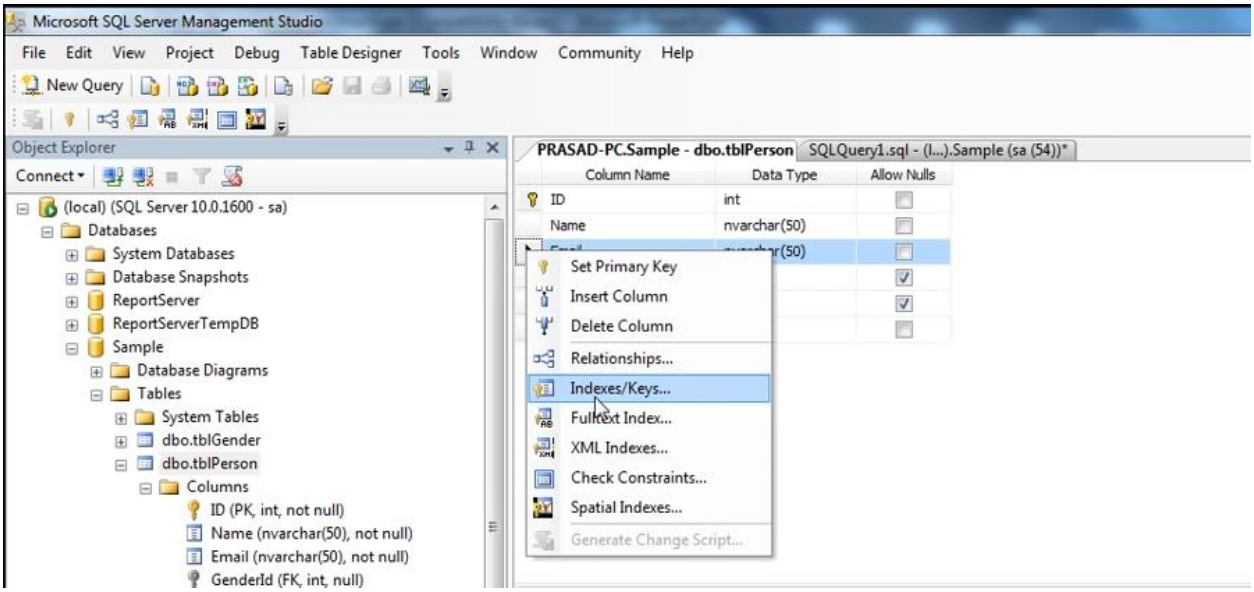
Thus even though we know that Email is always unique but we cannot mark it as another primary key.

However, we can mark Email as Unique Key. To do so:

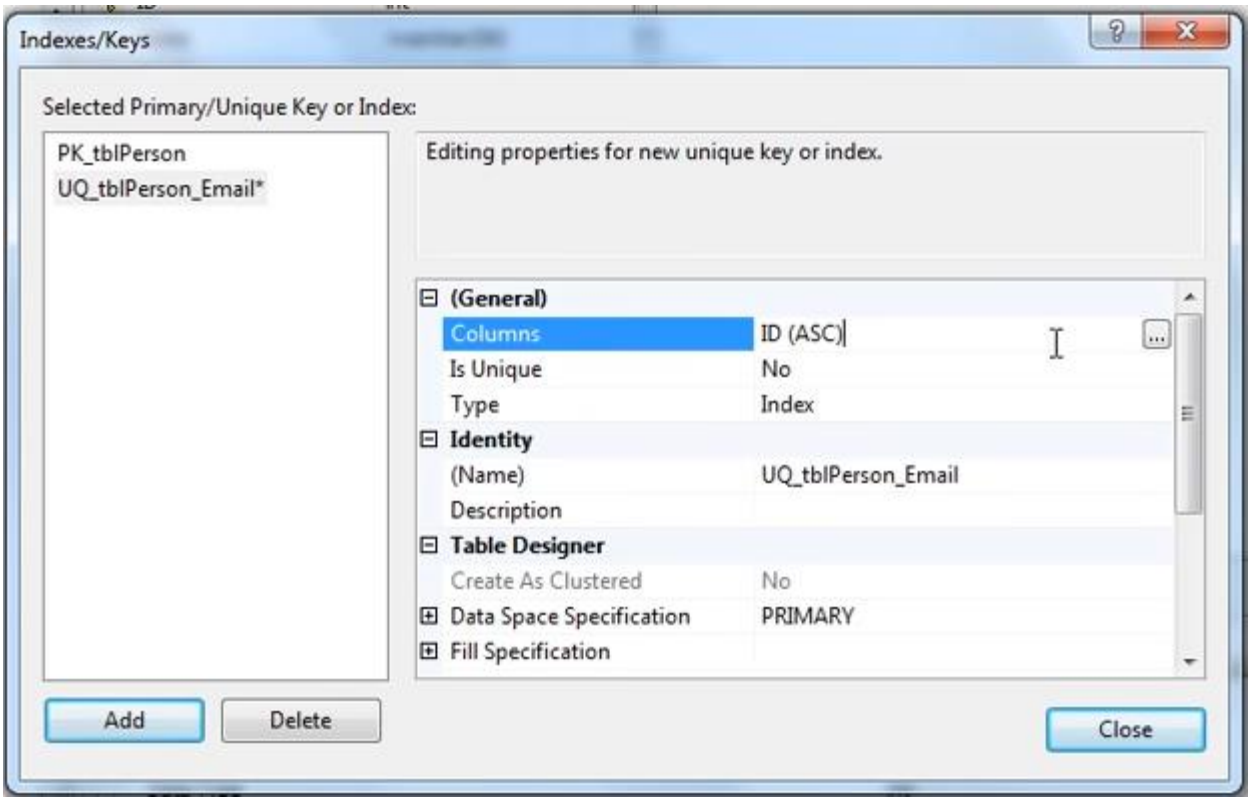
Right click on the table in which we wish to mark a column as unique key, select Design:



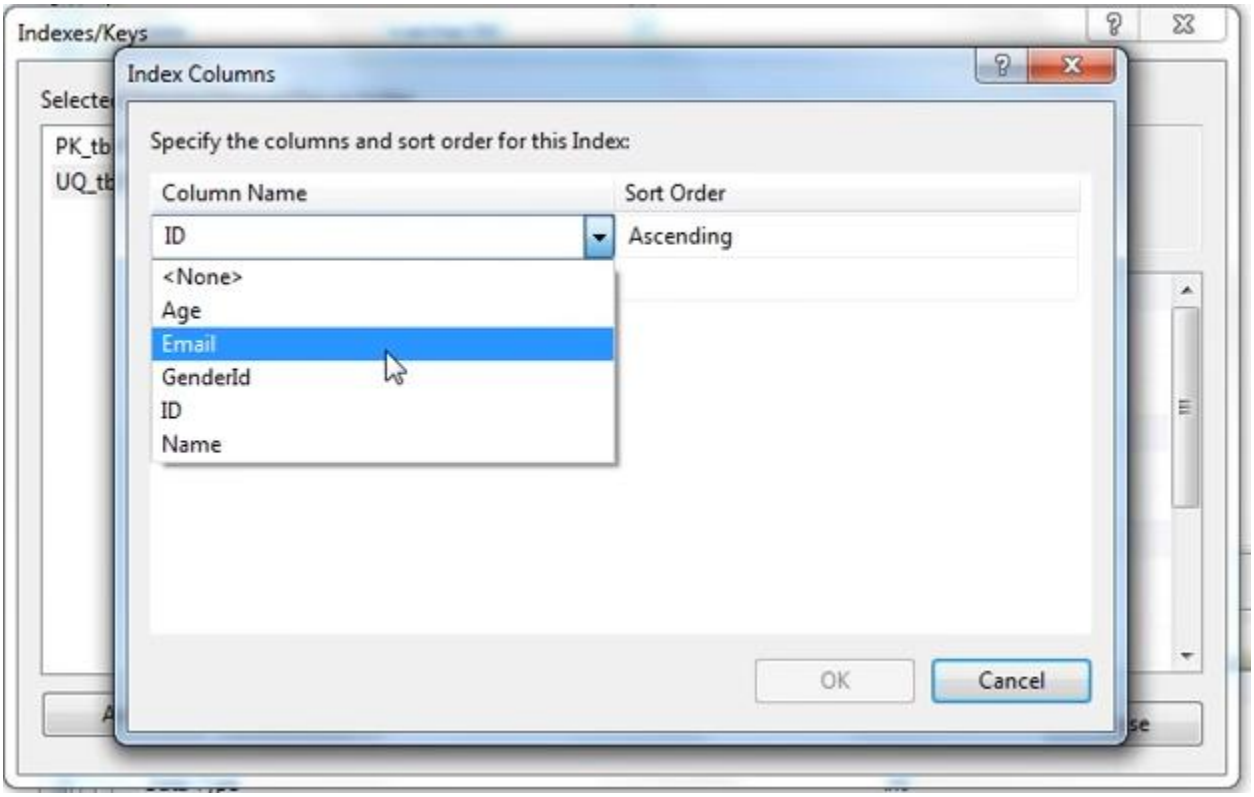
In the table designer select the column that we wish to mark as Unique Key, right click and then select option Indexes/Keys:



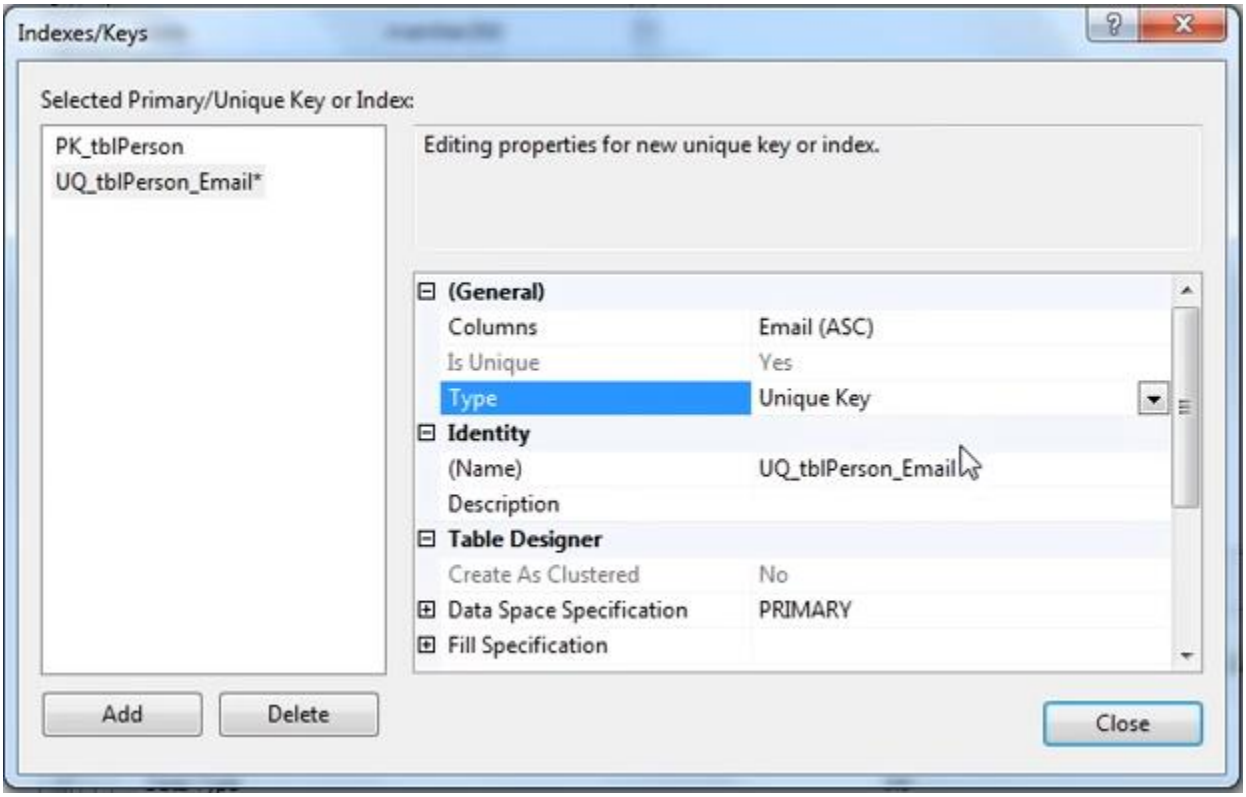
This brings up a pop up in which a Unique Key can be marked by adding the column and then giving name to the unique key, which normally begins with UQ_



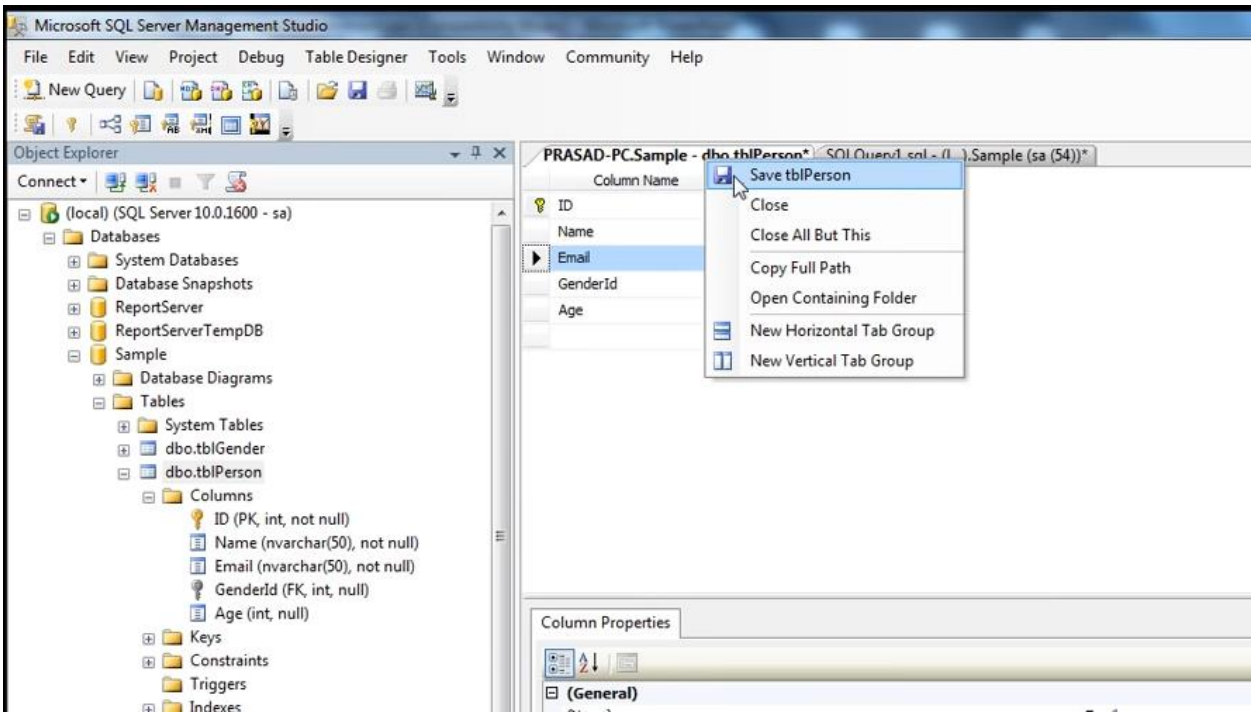
Select the ellipsis button of Columns, which brings up a pop up in which you select the column, e.g. select Email if you want to maintain uniqueness of Email column.



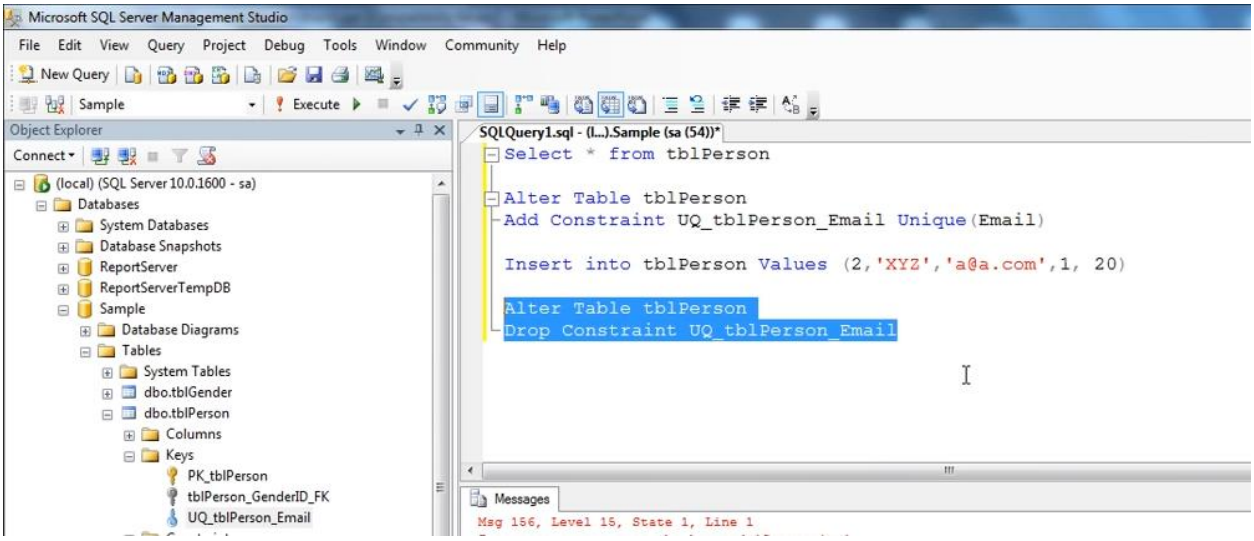
And then Edit properties of new unique key as follows:



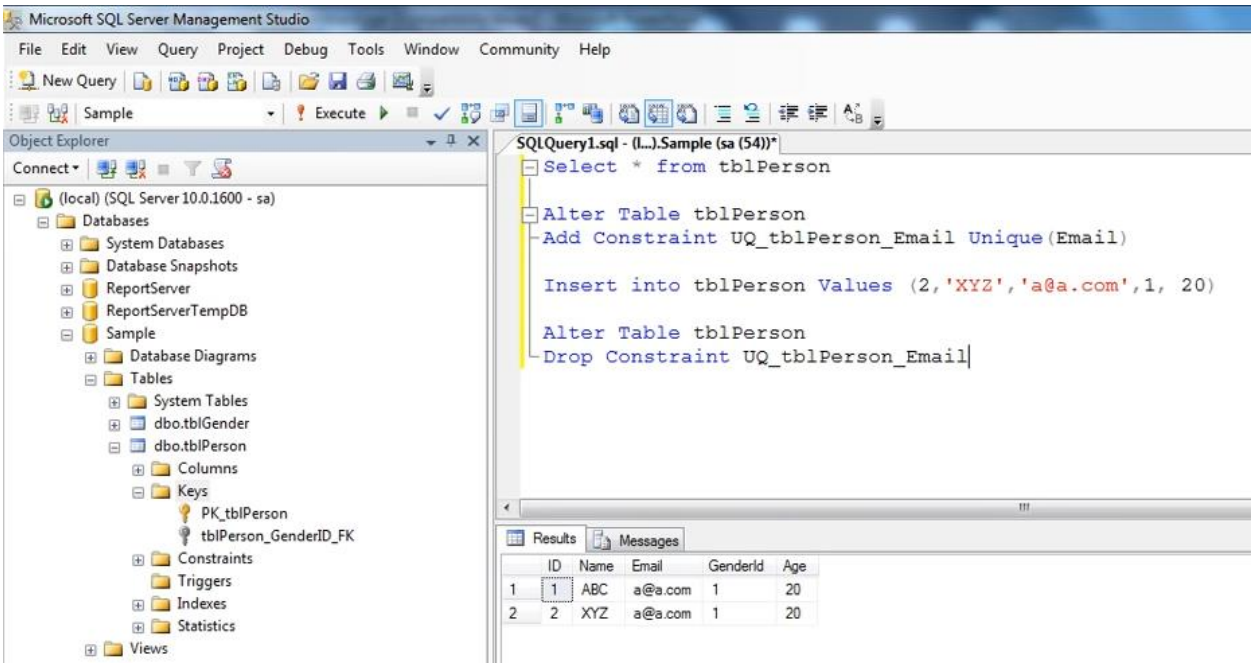
And the save the table.



Adding or Dropping a Unique Key using a query:



We can see below that from Keys list the unique key has been dropped.



Select statement

- Select specific or all columns
- Distinct rows
- Filtering with where clause
- Wild Cards in SQL Server
- Joining multiple conditions using AND and OR operators
- Sorting rows using order by
- Selecting top n or top n percentage of rows

```
Select * from tblPerson
```

```
SELECT [ID]
      ,[Name]
      ,[Email]
      ,[GenderId]
      ,[Age]
      ,[City]
  FROM [Sample].[dbo].[tblPerson]
GO
```

	ID	Name	Email	GenderId	Age	City
1	1	Tom	t@t.com	1	23	London
2	2	John	j@j.com	1	20	New York
3	3	Mary	mary.mary.com	2	21	Sydney
4	4	Josh	josh.j@dell.com	1	29	London
5	5	Sara	sara@abc.com	2	25	Mumbai

```
Select DISTINCT City from tblPerson
```

	City
1	London
2	Mumbai
3	New York
4	Sydney

```
Select DISTINCT Name, City from tblPerson
```

	Name	City
1	John	New York
2	Josh	London
3	Mary	Sydney
4	Sara	Mumbai
5	Tom	London

Filtering the records with Where Clause: E.g. to get people residing only in London:

```
Select * from tblPerson Where City = 'London'
```

	ID	Name	Email	GenderId	Age	City
1	1	Tom	t@t.com	1	23	London
2	4	Josh	josh.j@dell.com	1	29	London

Filtering the records to get people NOT residing in London:

```
Select * from tblPerson Where City <> 'London'
```

	ID	Name	Email	GenderId	Age	City
1	2	John	j@j.com	1	20	New York
2	3	Mary	mary.mary.com	2	21	Sydney
3	5	Sara	sara@abc.com	2	25	Mumbai

Operators and Wild Cards

Operator	Purpose
=	Equal to
!= or <>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
IN	Specify a list of values
BETWEEN	Specify a range
LIKE	Specify a pattern
NOT	Not in a list, range etc...

Operator	Purpose
%	Specifies zero or more characters
_	Specifies exactly one character
[]	Any character with in the brackets
[^]	Not any character with in the brackets

SQLQuery17.sql - L...B9U5G9\Gauri (52))*

```
select * from tbl_persontbl Where Email LIKE '_@_.com'

select * from tbl_persontbl Where Age in (20,23,26)

select * from tbl_persontbl Where Age Between 20 AND 26

--Returns all people whose name starts with M or S or T
-- [] is to specify list of characters
-- ^ symbol specify to return all those people whose name does not start with M or S or T

select * from tbl_persontbl Where Name LIKE '[MST]%'

select * from tbl_persontbl Where Name LIKE '[^MST]%'
```

We use the LIKE operator **to find a pattern**. And with LIKE operator we use something like Wild Cards, E.g. in the statement below % is a wild card, which stands for "some characters".

```
Select * from tblPerson Where Email NOT LIKE '%@%'
```

How can we find an Email that has **exactly one character before @ symbol and one character after @ symbol?**

```
Select * from tblPerson Where Email NOT LIKE '_@_.com'
```

IMPORTANT: Remember any time you are using Character or Text data type that has to be specified within single quotes.

What if we want that selected name **should start with** either M or S or T and records with other names are not required?

```
Select * from tblPerson Where Name LIKE '[MST]%'
```

	ID	Name	Email	GenderId	Age	City
1	1	Tom	t@t.com	1	23	London
2	3	Mary	mary.mary.com	2	21	Sydney
3	5	Sa	sara@abc.com	2	25	Mumbai

What if we want that selected name **should NOT start with** either M or S or T and records with other names are not required?

```
Select * from tblPerson Where Name LIKE '[^MST]%'
```

	ID	Name	Email	GenderId	Age	City
1	2	John	j@j.com	1	20	New York
2	4	Josh	josh.j@dell.com	1	29	London

```
Select * from tblPerson Where (City = 'London' or City  = 'Mumabi') AND Age > 25
```

	ID	Name	Email	GenderId	Age	City
1	4	Josh	josh.j@dell.com	1	29	London

Sorting by name in descending order then sorting by age in ascending order:

```
Select * from tblPerson order by Name DESC, Age ASC
```

	ID	Name	Email	GenderId	Age	City
1	1	Tom	t@t.com	1	23	London
2	5	Sara	sara@abc.com	2	25	Mumbai
3	3	Mary	mary.mary.com	2	21	Sydney
4	4	Josh	josh.j@dell.com	1	29	London
5	2	John	j@j.com	1	20	New York

Top keyword is especially useful if table is large. If we want to see only top 10 or 20 rows of a table then it is very useful. Since we don't have 10 records, this will return all records.

```
Select top 10 * from tblPerson
```

	ID	Name	Email	GenderId	Age	City
1	1	Tom	t@t.com	1	23	London
2	2	John	j@j.com	1	20	New York
3	3	Mary	mary.mary.com	2	21	Sydney
4	4	Josh	josh.j@dell.com	1	29	London
5	5	Sara	sara@abc.com	2	25	Mumbai

We can also use Percent or top n specific columns:

```
Select top 1 Percent * from tblPerson
Select top 2 Name, Age from tblPerson
```

Group by

- Grouping rows using GROUP BY
- Filtering Groups
- Difference between WHERE and HAVING

GROUP BY

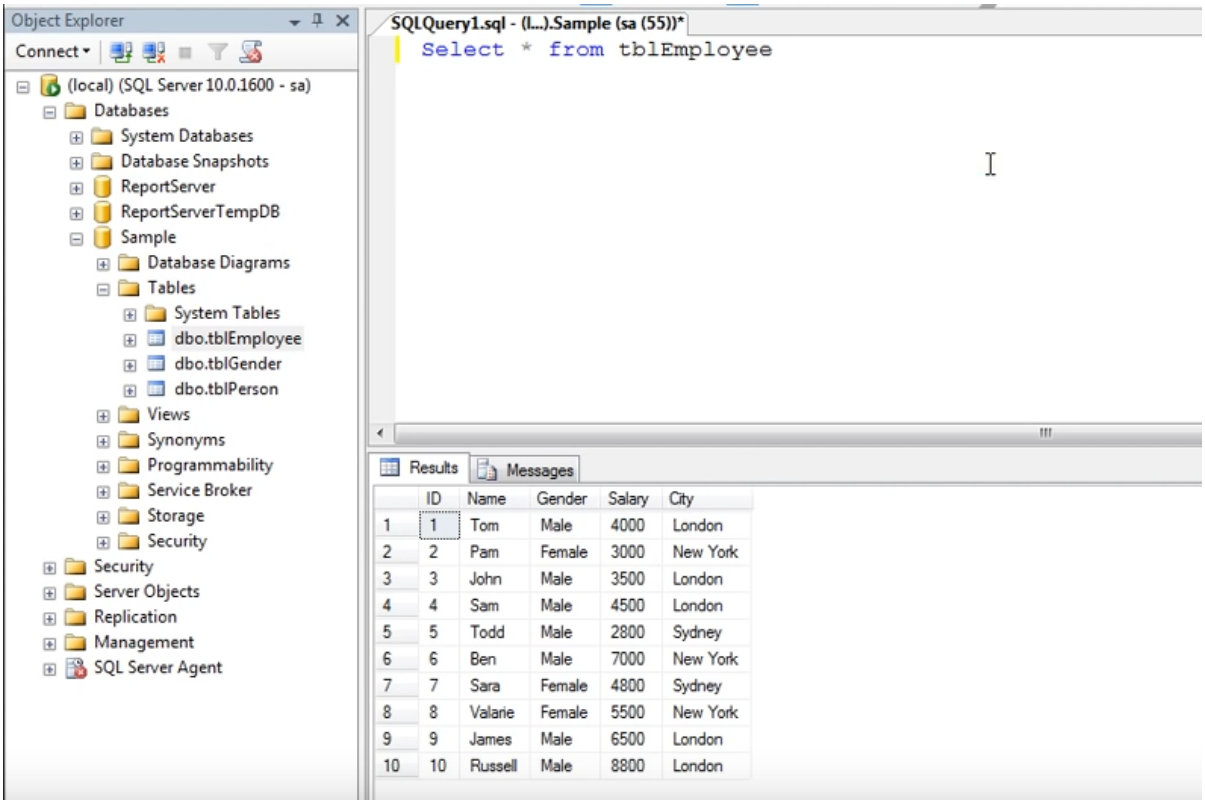
ID	Name	Gender	Salary	City
1	Tom	Male	4000	London
2	Pam	Female	3000	New York
3	John	Male	3500	London
4	Sam	Male	4500	London
5	Todd	Male	2800	Sydney
6	Ben	Male	7000	New York
7	Sara	Female	4800	Sydney
8	Valarie	Female	5500	New York
9	James	Male	6500	London
10	Russell	Male	8800	London

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.

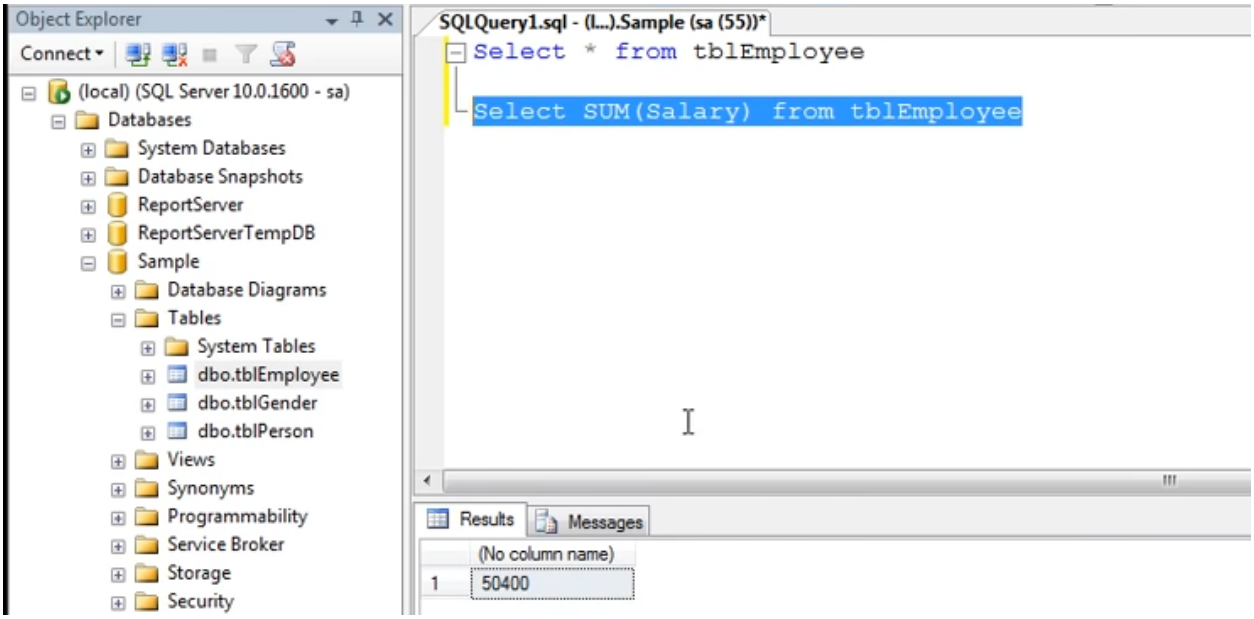
City	TotalSalary
London	27300
New York	15500
Sydney	7600

```
SELECT City, SUM(Salary) AS TotalSalary
FROM tblEmployee GROUP BY City
```

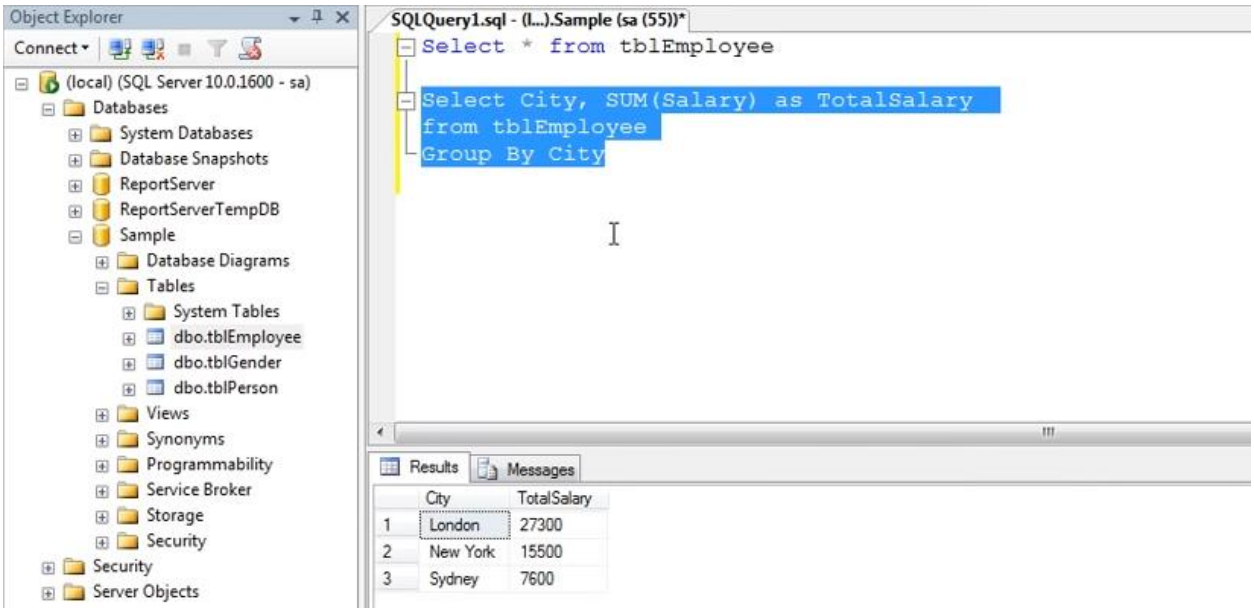
Note: If you omit, the GROUP BY clause and try to execute the query, you get an error
Column 'tblEmployee.City' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.



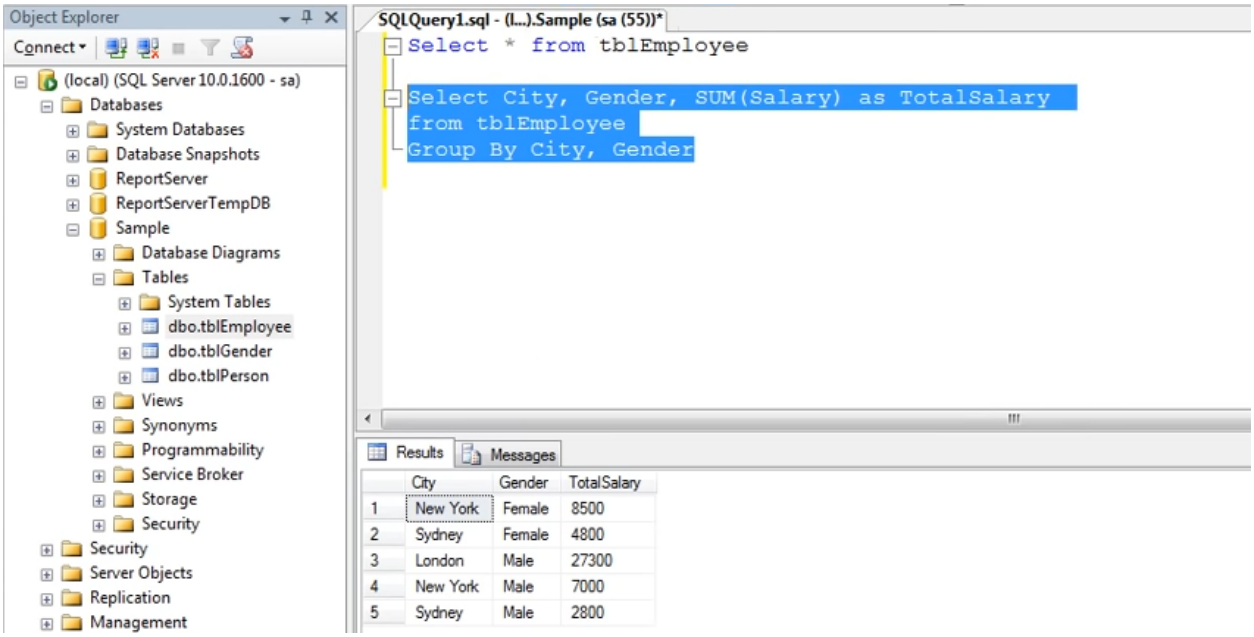
In the above table if I want to know total salary being paid, then:



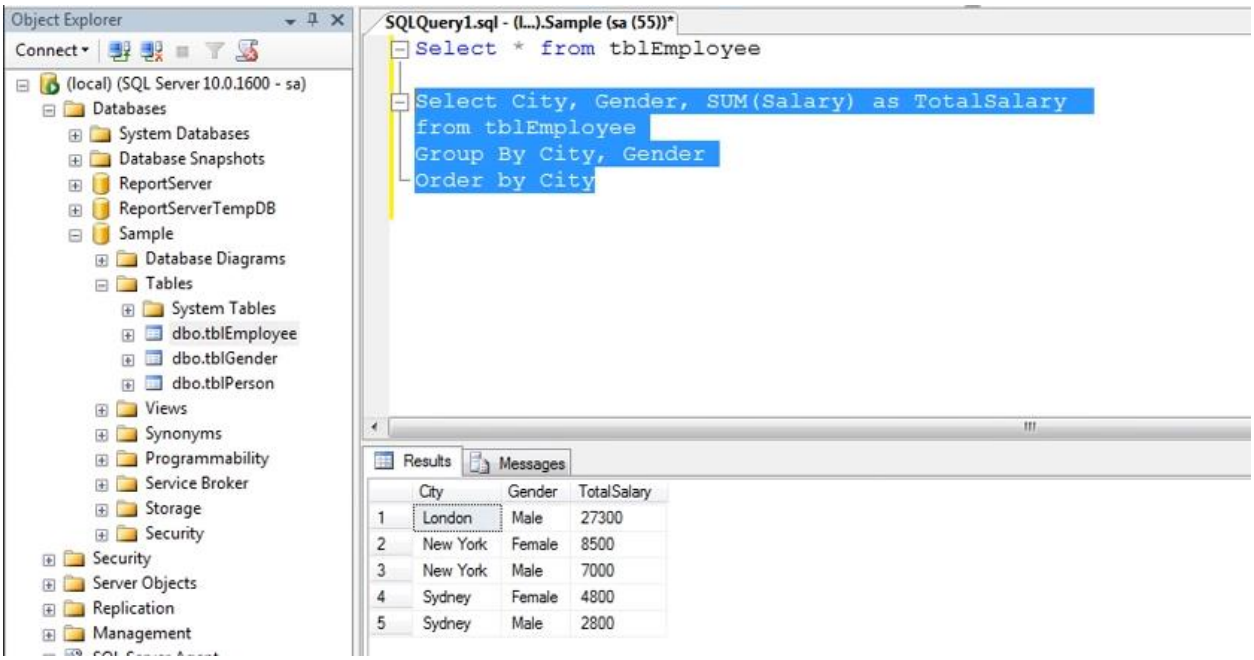
There are several aggregate functions in SQL Server such as MAX, MIN, SUM etc. Group By is always used with an aggregate function. E.g. I want to know the salary being paid by the City.



What if we want to further break down by gender also i.e. grouping first by City and then by Gender.



We can further Order By city:



Instead, we can also sort first by gender and then by city:

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

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

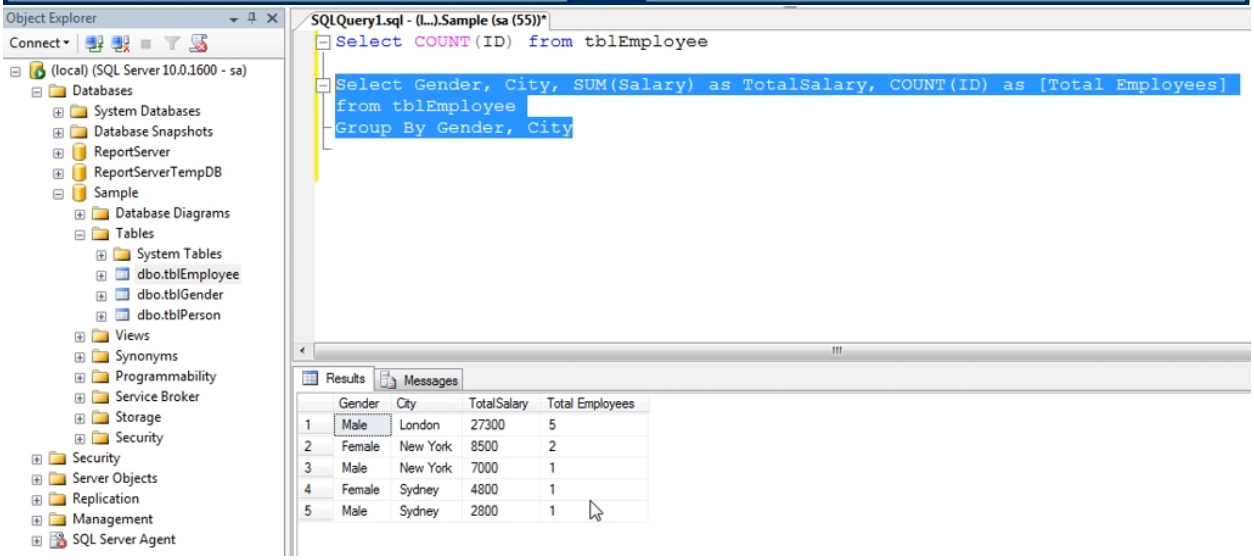
What if we also want Total Number of Employees?

GROUPING BY multiple columns

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

Using multiple aggregate functions

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



How can we filter the rows that only have Male employees?

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.

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

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

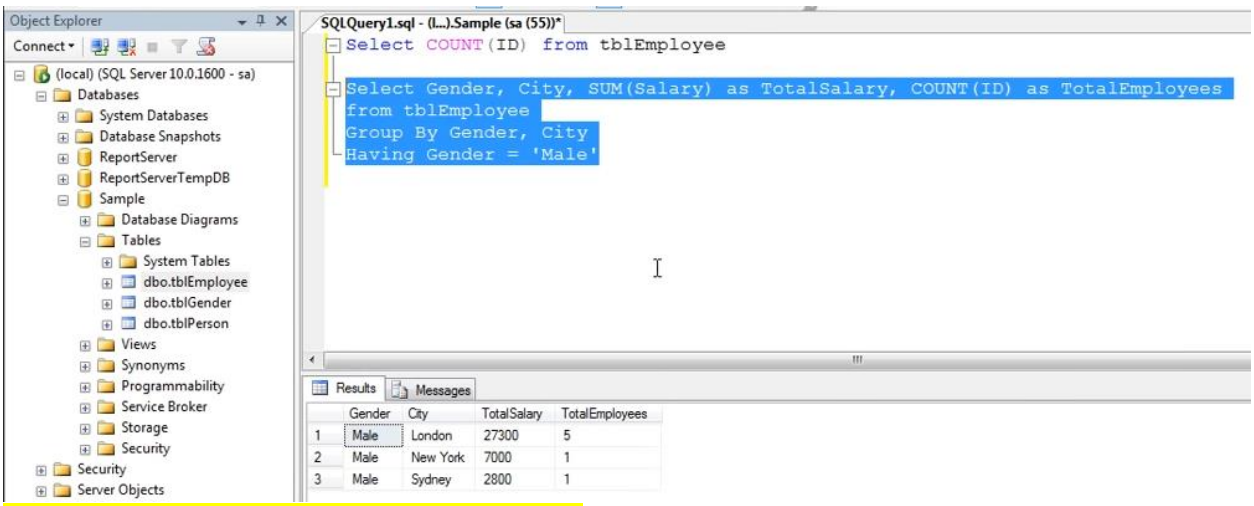
Note: From a performance standpoint, you cannot say that one method is less efficient than the other. Sql server optimizer analyses each statement and selects an efficient way of executing it. As a best practice, use the syntax that clearly describes the desired result. Try to eliminate rows that you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

```
Select Gender, City, SUM(Salary) as TotalSalary, COUNT(ID) as TotalEmployees
from tblEmployee
Where Gender = 'Male'
Group By Gender, City
```

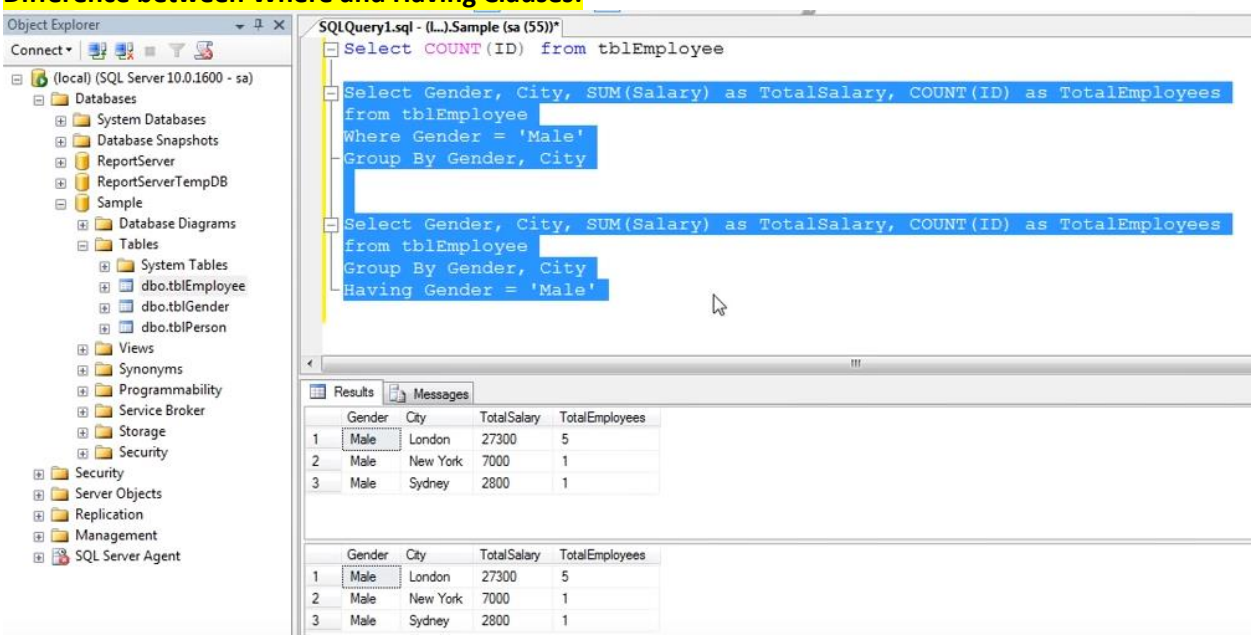
	Gender	City	TotalSalary	TotalEmployees
1	Male	London	27300	5
2	Male	New York	7000	1
3	Male	Sydney	2800	1

Having Clause

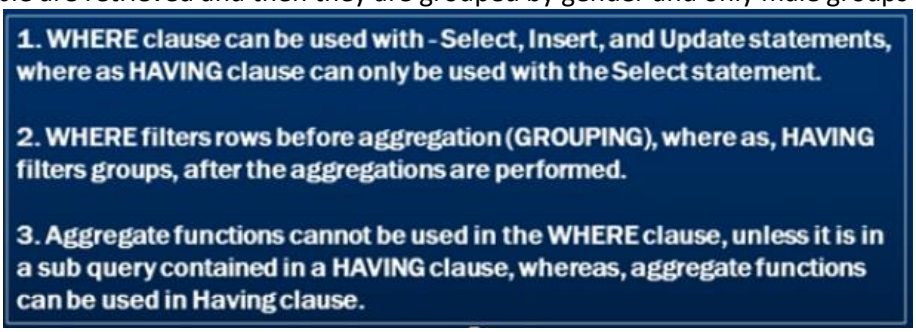
Instead of Where clause, we can use Having clause but Having clause should come after Group By clause.



Difference between Where and Having Clauses:



Difference between Where and Having is that when Where clause is used from the table only male records are retrieved and then they are grouped. But when Having clause is used then all the records from table are retrieved and then they are grouped by gender and only male groups are shown.



Joins

The two tables below on the right hand side are related to each other by a foreign key "DepartmentId" in top table, which is Id in the bottom table. The two tables are related and we want an output like the table on the left hand side in which columns from both the tables are being displayed. This requires joining two tables.

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

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

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

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

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

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

Inner Join: A **join** that returns all rows from both the participating tables where the key record of one table is equal to the key records of another table. This type of **join** required a comparison operator to match rows from the participating tables based on a common field or column of both the tables.

Inner Join or Join

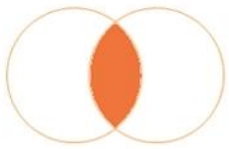
ID	Name	Gender	Salary	DepartmentId
1	Tom	Male	4000	1
2	Pam	Female	3000	3
3	John	Male	3500	1
4	Sam	Male	4500	2
5	Todd	Male	2800	2
6	Ben	Male	7000	1
7	Sara	Female	4800	3
8	Valarie	Female	5500	1
9	James	Male	6500	NULL
10	Russell	Male	8800	NULL

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

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

INNER JOIN or JOIN

Matching Rows only – Non matching rows eliminated



Employees - Departments

INNER JOIN returns only the matching rows between both the tables. Non matching rows are eliminated.

```
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: For Inner Join we can write simply Join also. But writing Inner Join is preferred.

Left Outer Join or Left Join: In case we want all matching rows plus all non-matching rows also from left table.

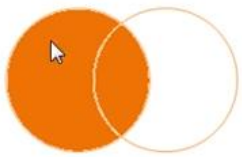
Left Outer Join or Left Join

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

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

LEFT OUTER JOIN or LEFT JOIN

Matching Rows + Non matching rows from the left table



Employees - Departments

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

LEFT JOIN returns all the matching rows + non matching rows from the left table.

```
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
```

PRAGIM Technologies | www.pragimtech.com | 9900113931

Right Outer Join or Right Join: In case we want all matching rows plus all non-matching rows also from right table.

Right Outer Join or Right Join

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

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

RIGHT OUTER JOIN or RIGHT JOIN

Matching Rows + Non matching rows from the right table

Employees - Departments

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

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

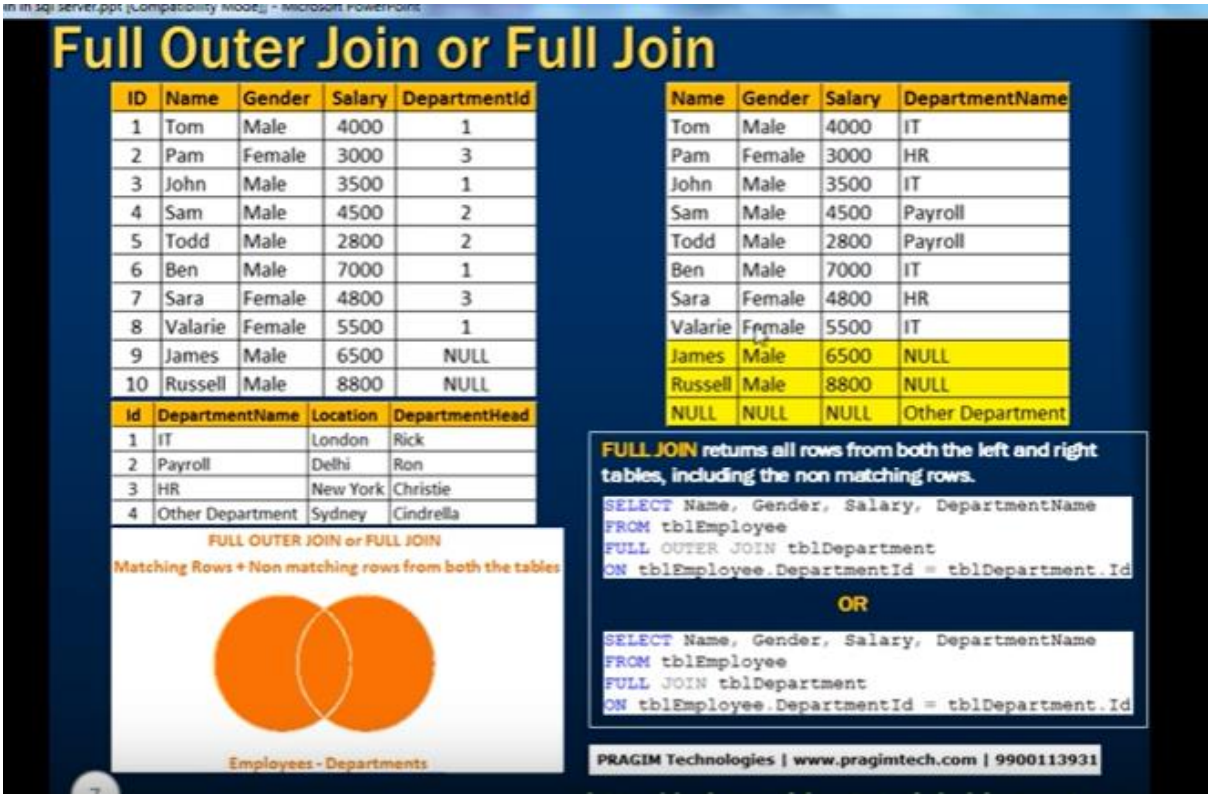
```
SELECT Name, Gender, Salary, DepartmentName
FROM tblEmployee
RIGHT OUTER JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.Id
```

OR

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

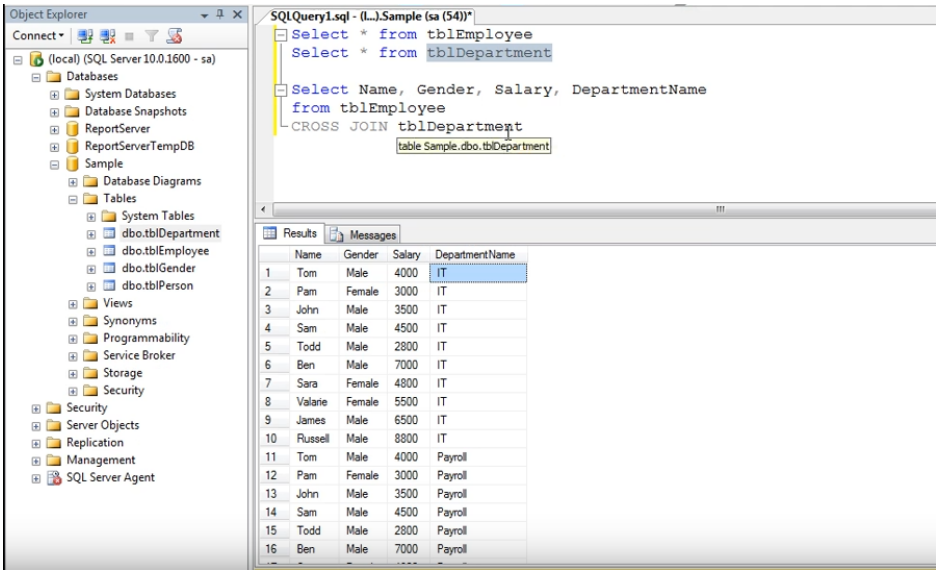
PRAGIM Technologies | www.pragimtech.com | 9900113931

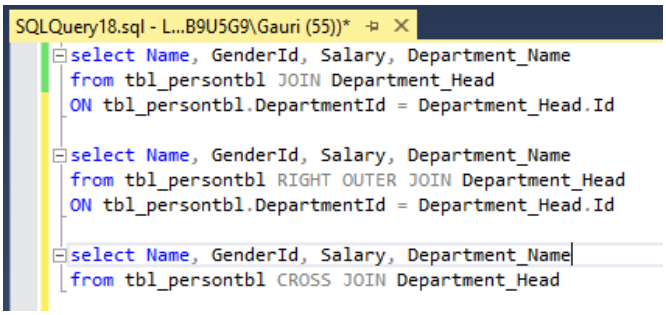
Full Outer Join or Full Join: Matching rows plus non-matching rows from both the tables.

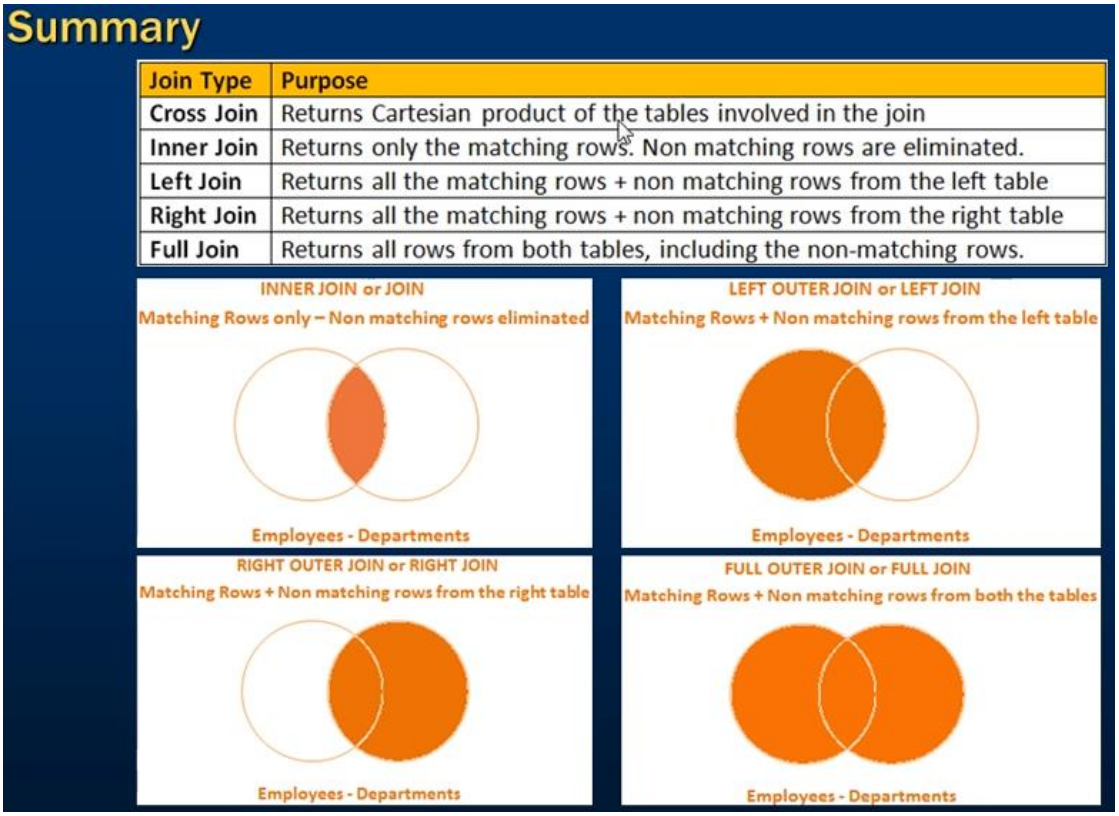


Cross Join: **CROSS JOIN** is used to combine each row of the first table with each row of the second table.
Note: A Cross Join does not have an ON clause.

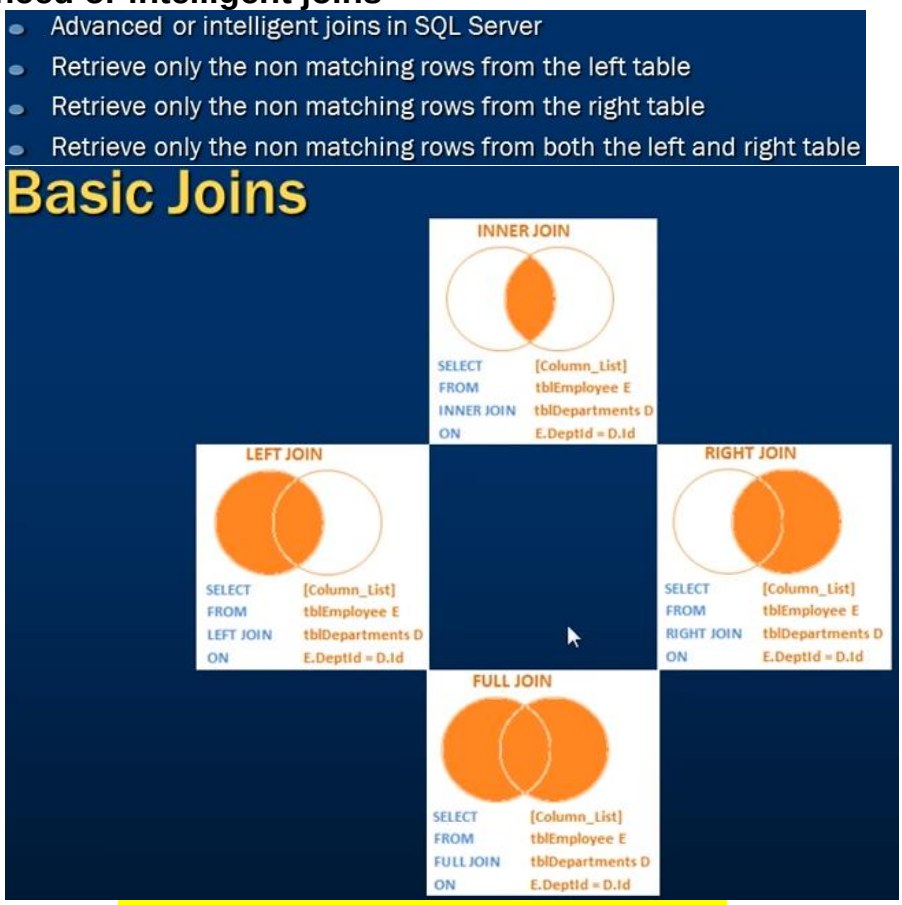
For example, if one table has 10 rows and the other 4 then cross join outputs 40 rows as below.


Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblDepartment
dbo.tblEmployee
dbo.tblGender
dbo.tblPerson
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
SQL Server Agent
SQLQuery1.sql - (l...).Sample (sa (54))*
Select * from tblEmployee
Select * from tblDepartment
Select Name, Gender, Salary, DepartmentName
from tblEmployee
CROSS JOIN tblDepartment
table Sample.dbo.tblDepartment
Results
Messages
Name Gender Salary DepartmentName
1 Tom Male 4000 IT
2 Pam Female 3000 IT
3 John Male 3500 IT
4 Sam Male 4500 IT
5 Todd Male 2800 IT
6 Ben Male 7000 IT
7 Sara Female 4800 IT
8 Valarie Female 5500 IT
9 James Male 6500 IT
10 Russell Male 8800 IT
11 Tom Male 4000 Payroll
12 Pam Female 3000 Payroll
13 John Male 3500 Payroll
14 Sam Male 4500 Payroll
15 Todd Male 2800 Payroll
16 Ben Male 7000 Payroll

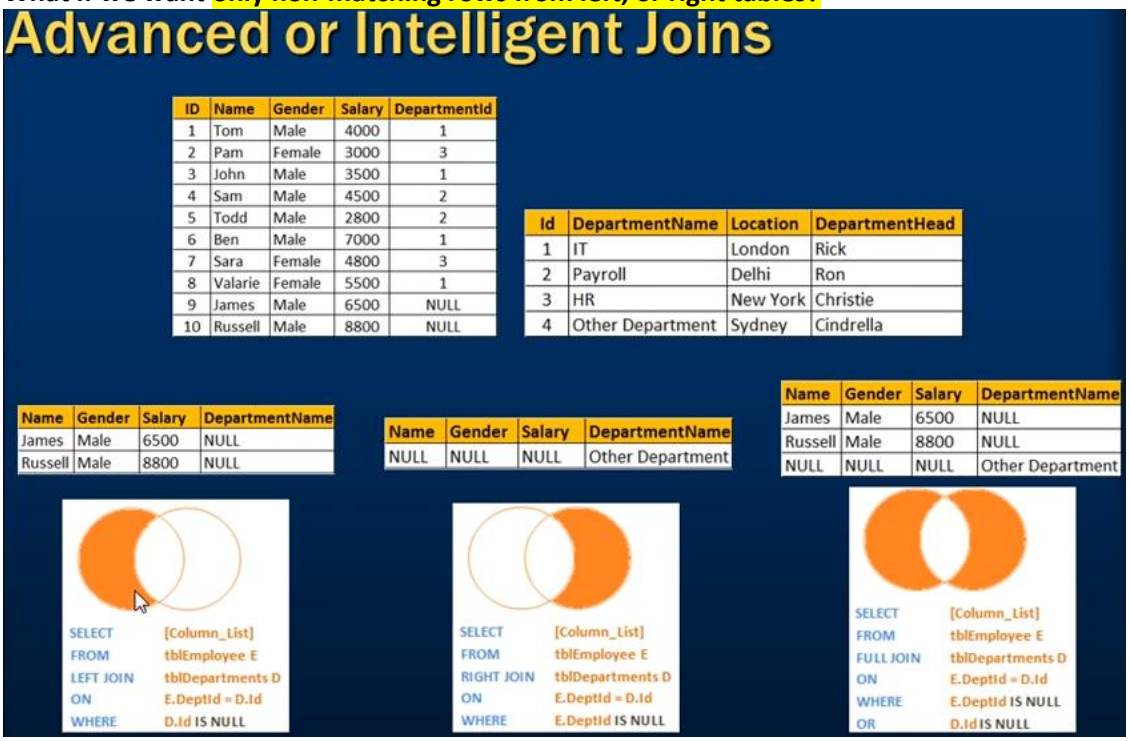

SQLQuery18.sql - L...B9U5G9\Gauri (55))*
select Name, GenderId, Salary, Department_Name
from tbl_persontbl JOIN Department_Head
ON tbl_persontbl.DepartmentId = Department_Head.Id
select Name, GenderId, Salary, Department_Name
from tbl_persontbl RIGHT OUTER JOIN Department_Head
ON tbl_persontbl.DepartmentId = Department_Head.Id
select Name, GenderId, Salary, Department_Name
from tbl_persontbl CROSS JOIN Department_Head


Summary
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.
INNER JOIN or JOIN
Matching Rows only – Non matching rows eliminated
Employees - Departments
LEFT OUTER JOIN or LEFT JOIN
Matching Rows + Non matching rows from the left table
Employees - Departments
RIGHT OUTER JOIN or RIGHT JOIN
Matching Rows + Non matching rows from the right table
Employees - Departments
FULL OUTER JOIN or FULL JOIN
Matching Rows + Non matching rows from both the tables
Employees - Departments

Advanced or intelligent joins



What if we want only non-matching rows from left, or right tables?



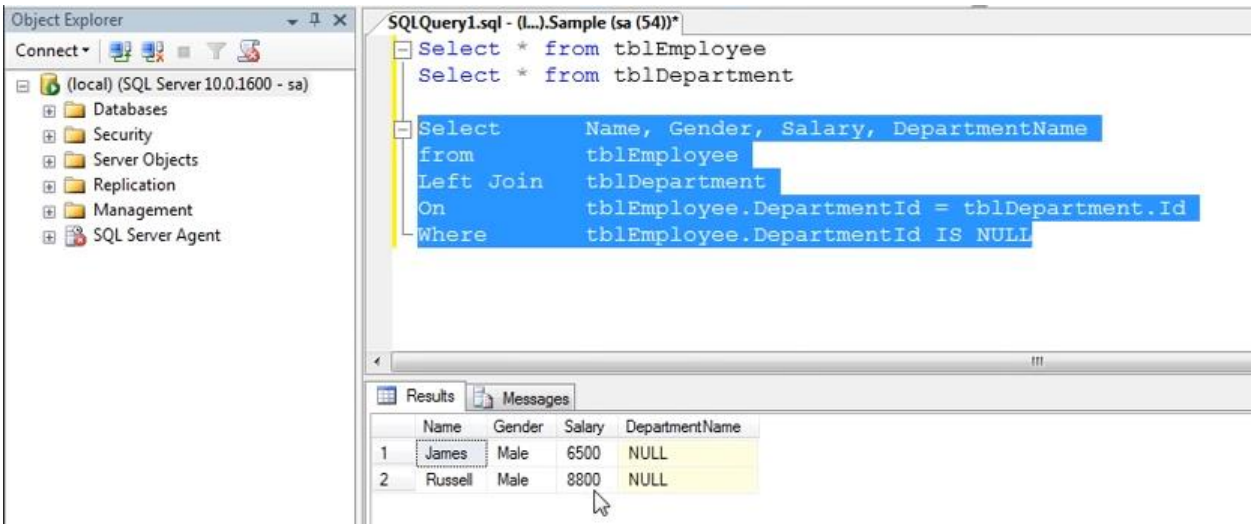
ID	Name	Gender	Salary	DepartmentId
1	Tom	Male	4000	1
2	Pam	Female	3000	3
3	John	Male	3500	1
4	Sam	Male	4500	2
5	Todd	Male	2800	2
6	Ben	Male	7000	1
7	Sara	Female	4800	3
8	Valarie	Female	5500	1
9	James	Male	6500	NULL
10	Russell	Male	8800	NULL

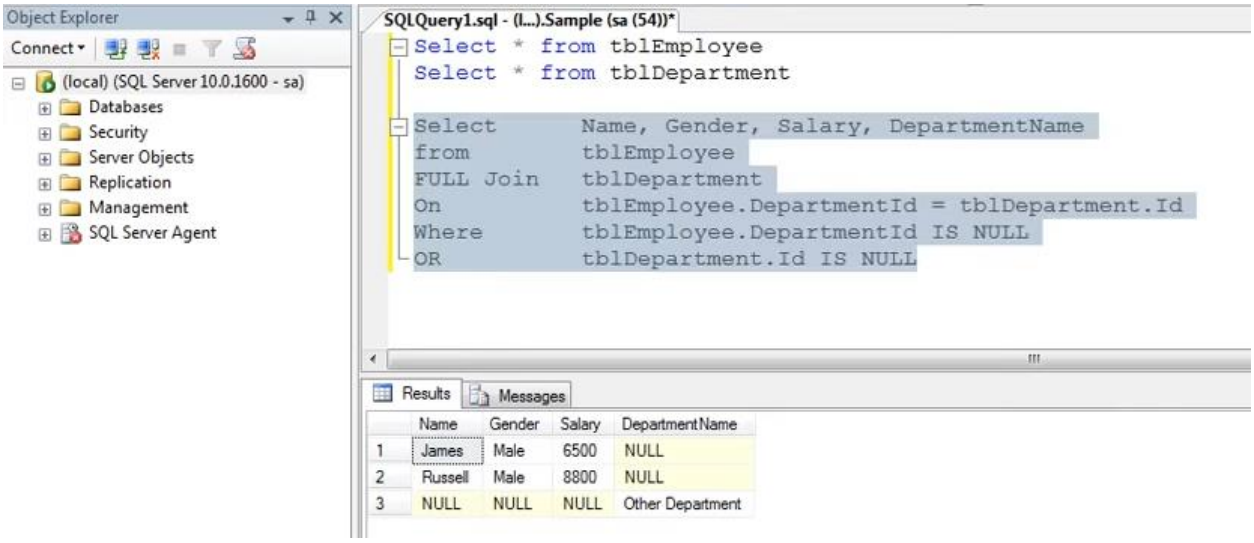
Id	DepartmentName	Location	DepartmentHead
1	IT	London	Rick
2	Payroll	Delhi	Ron
3	HR	New York	Christie
4	Other Department	Sydney	Cindrella

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

Name	Gender	Salary	DepartmentName
NULL	NULL	NULL	Other Department

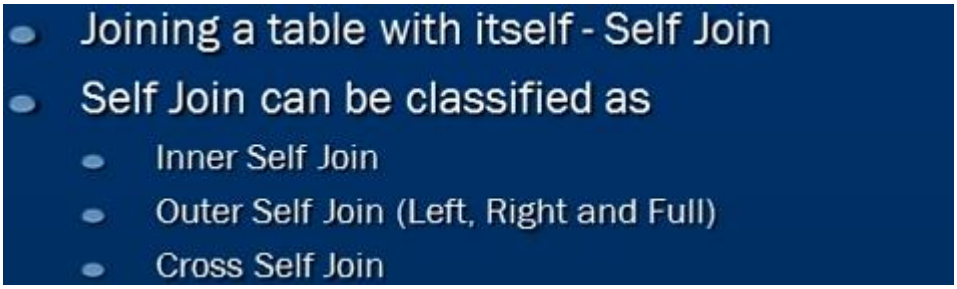
Name	Gender	Salary	DepartmentName
James	Male	6500	NULL
Russell	Male	8800	NULL
NULL	NULL	NULL	Other Department





Self Join

A **self join** is a **join** in which a table is **joined** with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY. To **join** a table itself means that each row of the table is combined with itself and with every other row of the table.



Self Join

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.

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

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

```
-- Left Outer Self Join
SELECT      E.Name AS Employee, M.Name AS Manager
FROM        tblEmployee E
LEFT JOIN   tblEmployee M
ON          E.ManagerId = M.EmployeeId
```

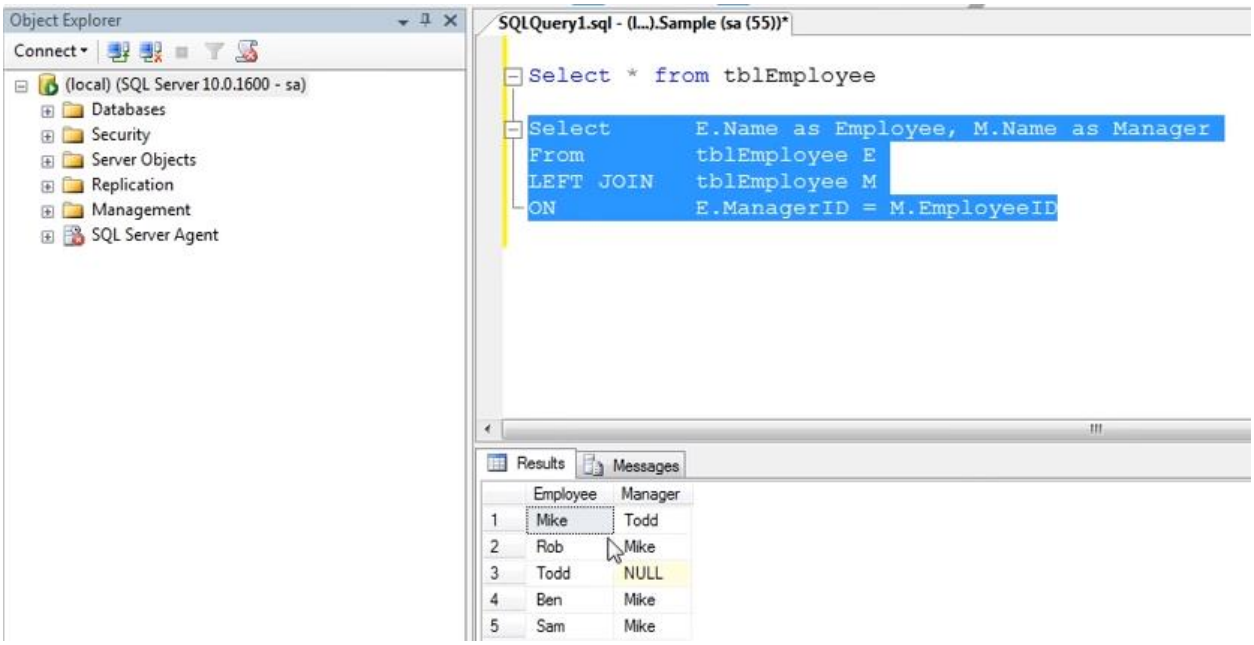
```
-- Inner Self Join
SELECT      E.Name AS Employee, M.Name AS Manager
FROM        tblEmployee E
INNER JOIN  tblEmployee M
ON          E.ManagerId = M.EmployeeId
```

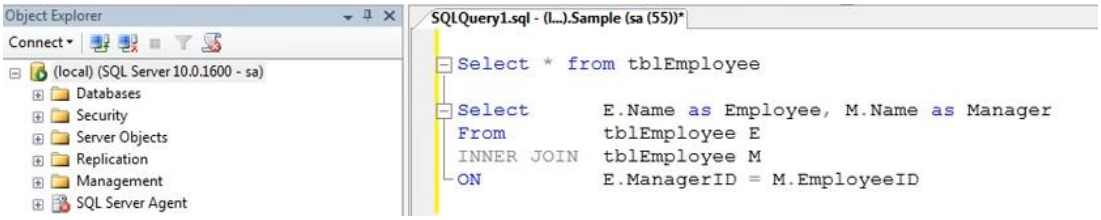
```
-- Cross Self Join
SELECT      E.Name AS Employee, M.Name AS Manager
FROM        tblEmployee E
CROSS JOIN  tblEmployee M
```

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:
1. INNER,
2. OUTER (Left, Right, Full)
3. CROSS Joins.





Replacing NULL

- Different ways to replace NULL values
  - ISNULL() function
  - CASE statement
  - COALESCE() function

Replacing NULL values with some Text such as 'No Manager'

Replacing NULL Values

Three ways to replace NULL values - ISNULL() Function, CASE Statement & COALESCE() Function

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

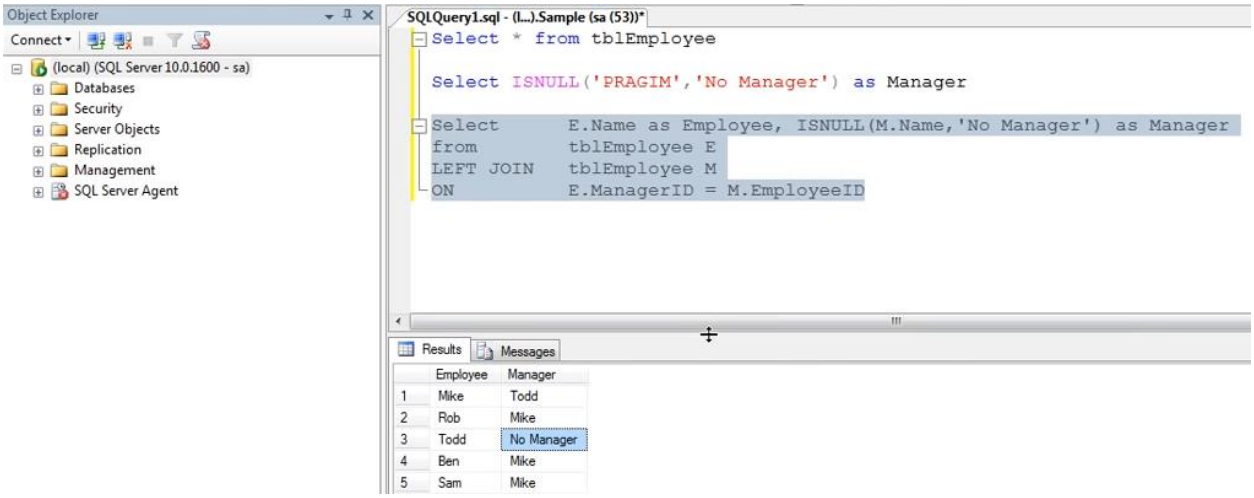
Employee	Manager
Mike	Todd
Rob	Mike
Todd	NULL
Ben	Mike
Sam	Mike

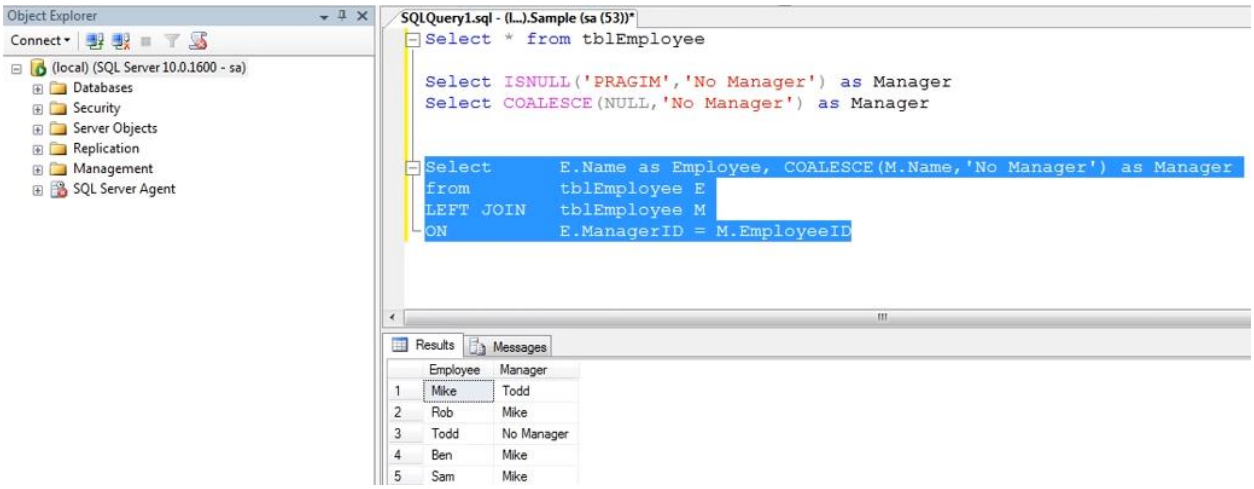
Employee	Manager
Mike	Todd
Rob	Mike
Todd	No Manager
Ben	Mike
Sam	Mike

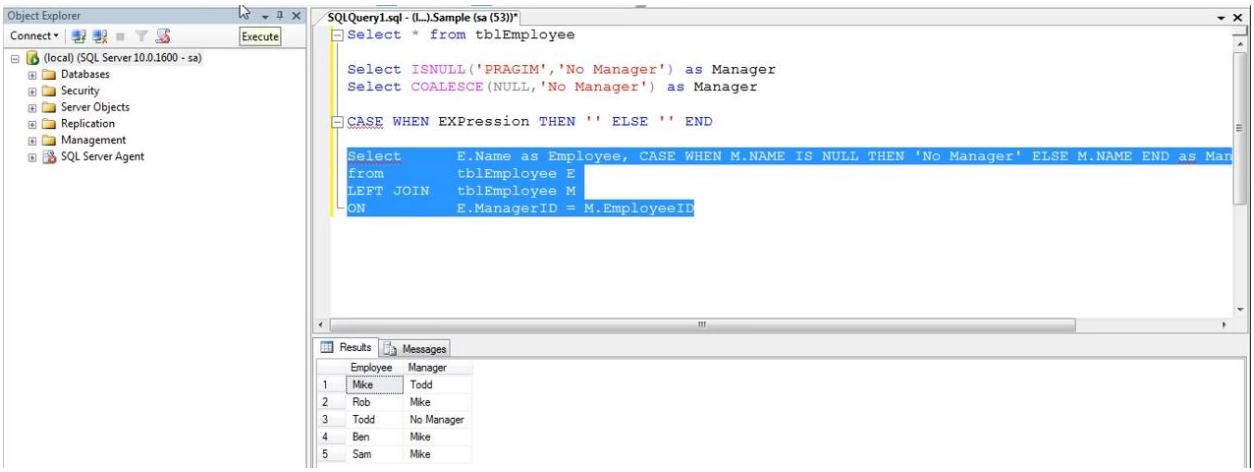
```
SELECT      E.Name as Employee, ISNULL(M.Name, 'No Manager') as Manager
FROM        tblEmployee E
LEFT JOIN   tblEmployee M
ON          E.ManagerID = M.EmployeeID
```

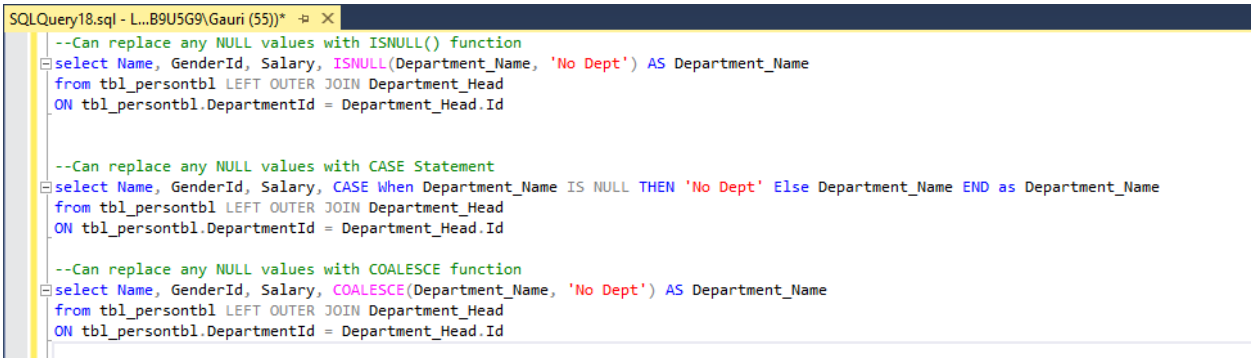
```
SELECT      E.Name as Employee,
            CASE
                WHEN M.Name IS NULL THEN 'No Manager' ELSE M.Name
            END
            as Manager
FROM        tblEmployee E
LEFT JOIN   tblEmployee M
ON          E.ManagerID = M.EmployeeID
```

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




Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
Security
Server Objects
Replication
Management
SQL Server Agent
SQLQuery1.sql - (l...).Sample (sa (53))*
Select * from tblEmployee
Select ISNULL('PRAGIM','No Manager') as Manager
Select COALESCE(NULL,'No Manager') as Manager
Select E.Name as Employee, COALESCE(M.Name,'No Manager') as Manager
from tblEmployee E
LEFT JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
Results
Messages
Employee Manager
1 Mike Todd
2 Rob Mike
3 Todd No Manager
4 Ben Mike
5 Sam Mike

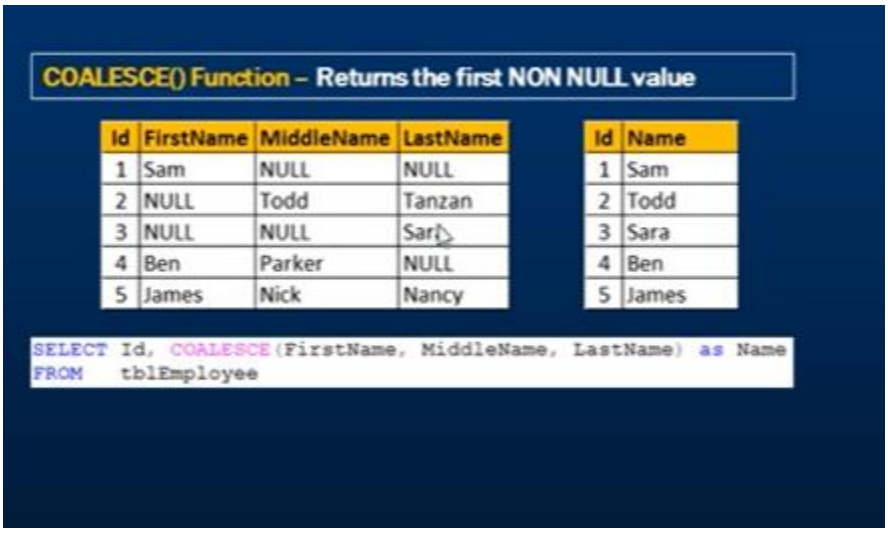

Object Explorer
Connect
Execute
(local) (SQL Server 10.0.1600 - sa)
Databases
Security
Server Objects
Replication
Management
SQL Server Agent
SQLQuery1.sql - (l...).Sample (sa (53))*
Select * from tblEmployee
Select ISNULL('PRAGIM','No Manager') as Manager
Select COALESCE(NULL,'No Manager') as Manager
CASE WHEN EXPression THEN '' ELSE '' END
Select E.Name as Employee, CASE WHEN M.NAME IS NULL THEN 'No Manager' ELSE M.NAME END as Man
from tblEmployee E
LEFT JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
Results
Messages
Employee Manager
1 Mike Todd
2 Rob Mike
3 Todd No Manager
4 Ben Mike
5 Sam Mike

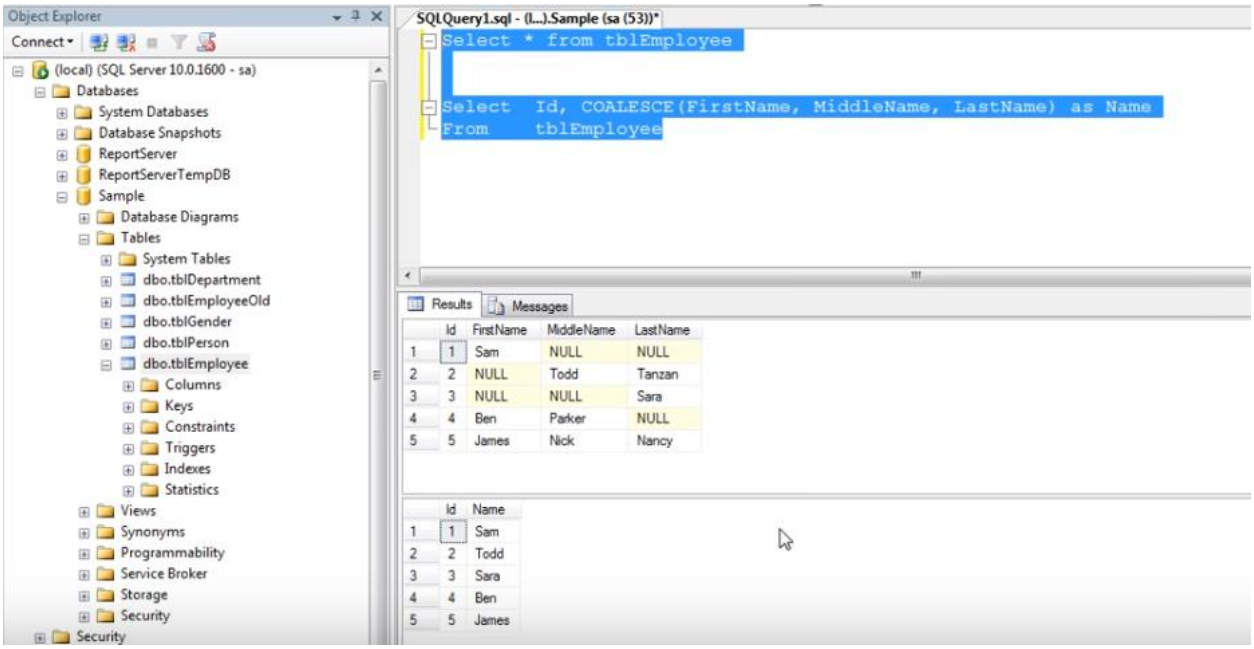

SQLQuery18.sql - L...B9U5G9\Gauri (55))*
--Can replace any NULL values with ISNULL() function
select Name, GenderId, Salary, ISNULL(Department_Name, 'No Dept') AS Department_Name
from tbl_persontbl LEFT OUTER JOIN Department_Head
ON tbl_persontbl.DepartmentId = Department_Head.Id
--Can replace any NULL values with CASE Statement
select Name, GenderId, Salary, CASE When Department_Name IS NULL THEN 'No Dept' Else Department_Name END as Department_Name
from tbl_persontbl LEFT OUTER JOIN Department_Head
ON tbl_persontbl.DepartmentId = Department_Head.Id
--Can replace any NULL values with COALESCE function
select Name, GenderId, Salary, COALESCE(Department_Name, 'No Dept') AS Department_Name
from tbl_persontbl LEFT OUTER JOIN Department_Head
ON tbl_persontbl.DepartmentId = Department_Head.Id

Coalesce function

In **SQL** Server (Transact-**SQL**), the **COALESCE** function returns the first non-null expression in the list. If all expressions evaluate to null, then the **COALESCE** function will return null.

E.g. in example below Coalesce returns first non NULL value out of FirstName, MiddleName, LastName.



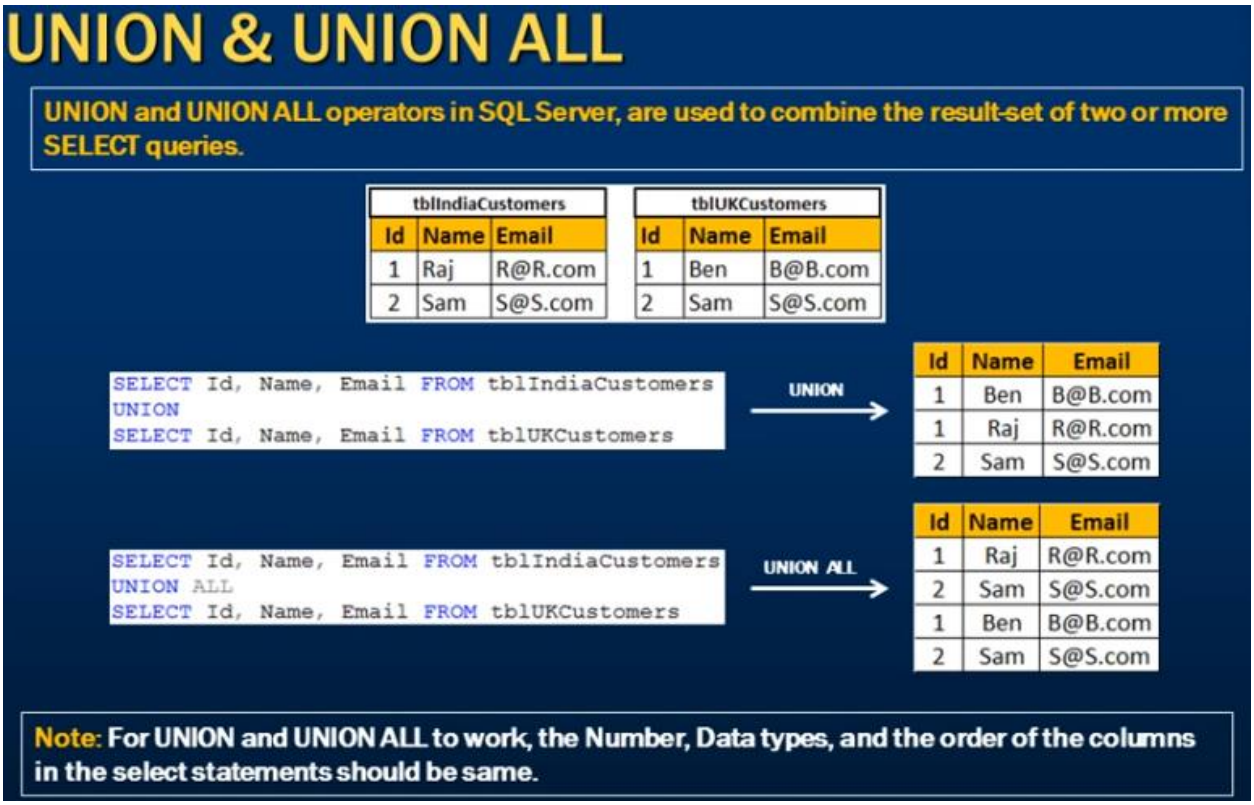


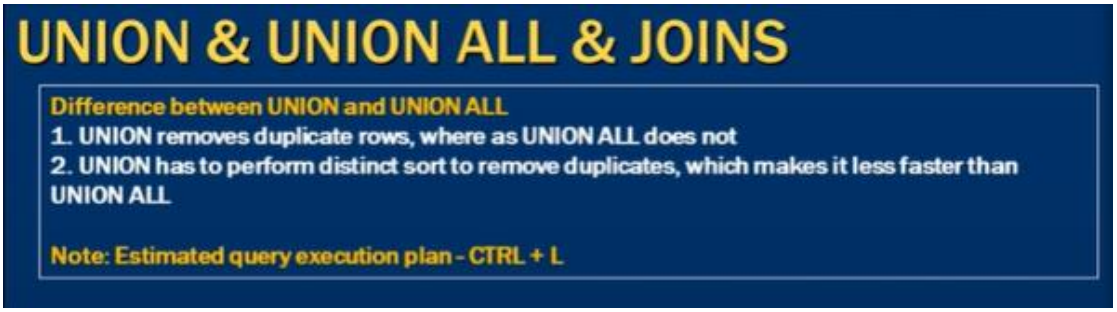
Union and union all

- Purpose of UNION and UNION ALL
- Difference between UNION and UNION ALL
- Difference between JOIN and UNION

The **UNION** command combines the result set of two or more SELECT statements (only distinct values). The following SQL UNION statement returns the Name and Email (**only distinct values**) from both the "tblIndiaCustomers" and the "tblUKCustomers" tables.

The **UNION ALL** command combines the result set of two or more SELECT statements (allows duplicate values). The following SQL statement returns the cities (**duplicate values also**) from both the "tblIndiaCustomers" and the "tblUKCustomers" tables.

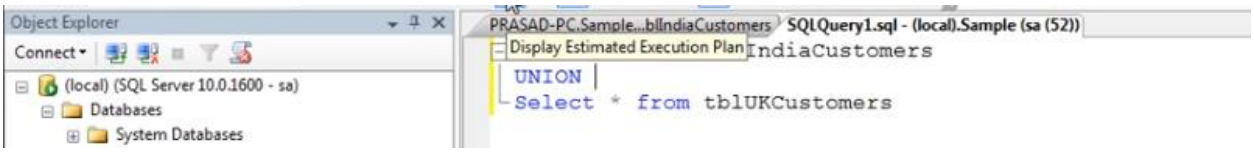




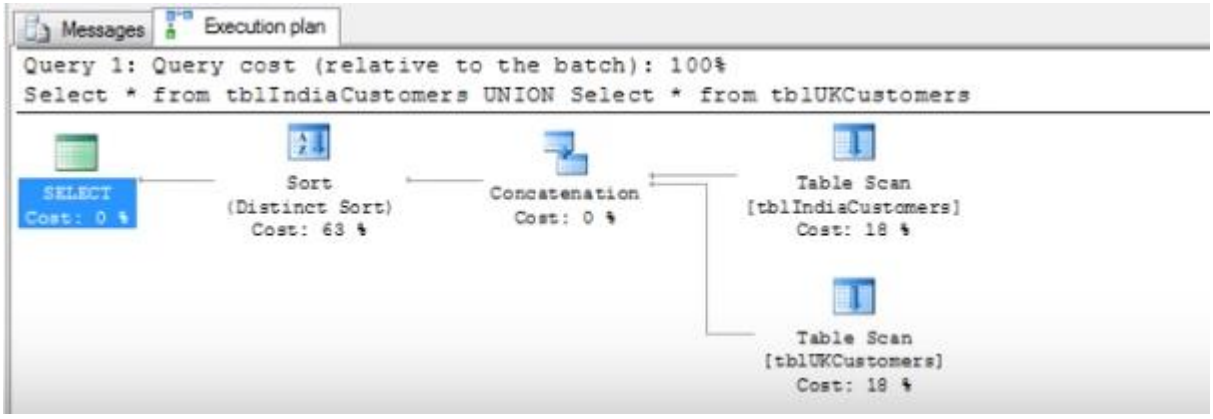
Note: The sequence of columns in both Select statements MUST be same data types.

Estimated Query Execution Plan

To view Estimated Query Execution Plan either use CTRL+L or SSMS menu;

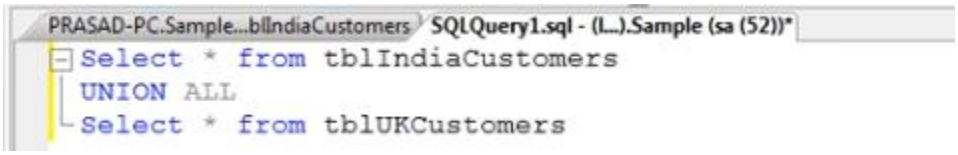


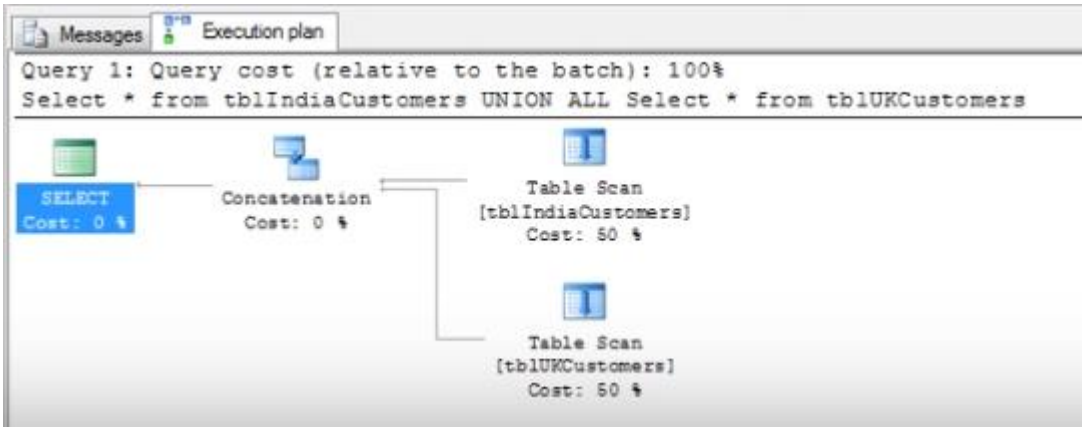
And the plan is displayed:



As we see in the plan above in case of UNION clause the Distinct Sort is taking 63% of execution time.

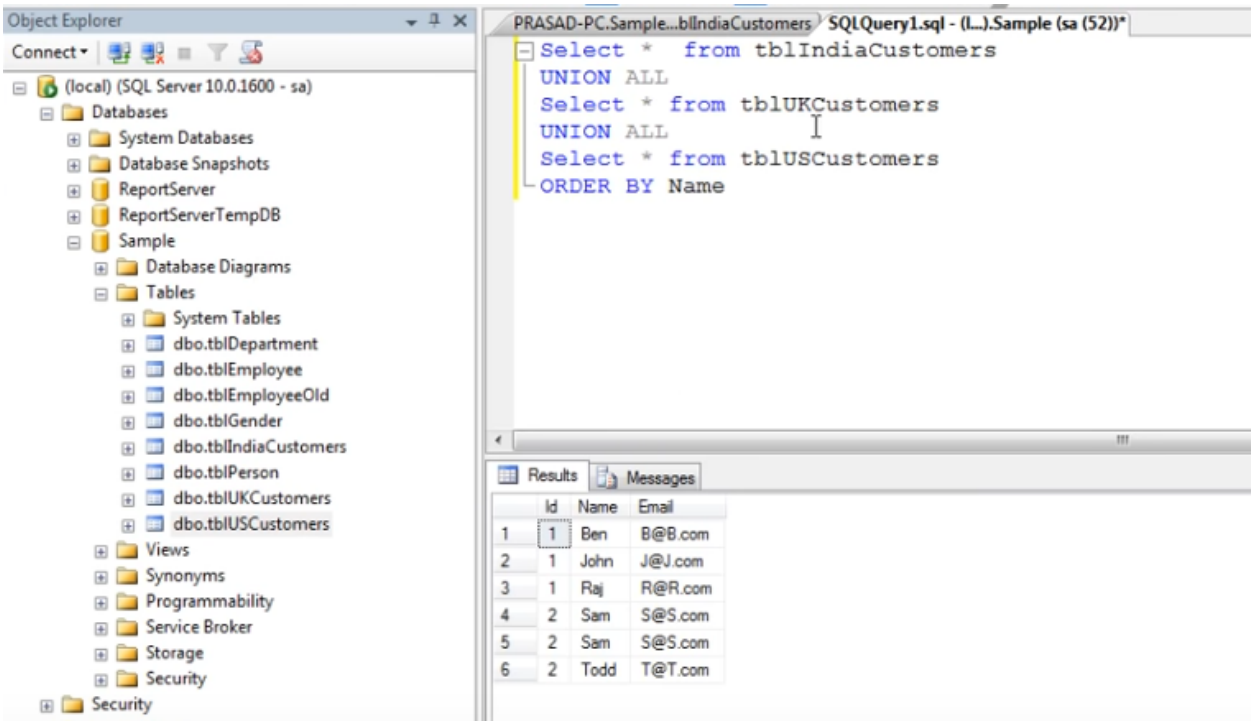
Whereas in UNION ALL clause as shown below there is no Distinct Sort. Hence, UNION is a little slower than UNION ALL.





Sorting the results of UNION and UNION ALL

Sorting results of a UNION or UNION ALL
ORDER BY clause should be used only on the last SELECT statement in the UNION query



Difference between UNION and JOIN
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

A **stored procedure in SQL** is a type of code in **SQL** that can be **stored** for later use and can be used many times. So, whenever you need to execute the query, instead of calling it you can just call the **stored procedure**. Values can be passed through **stored procedures**. Such procedures are stored in the database data dictionary. Uses for stored procedures include data-validation or access-control mechanisms.

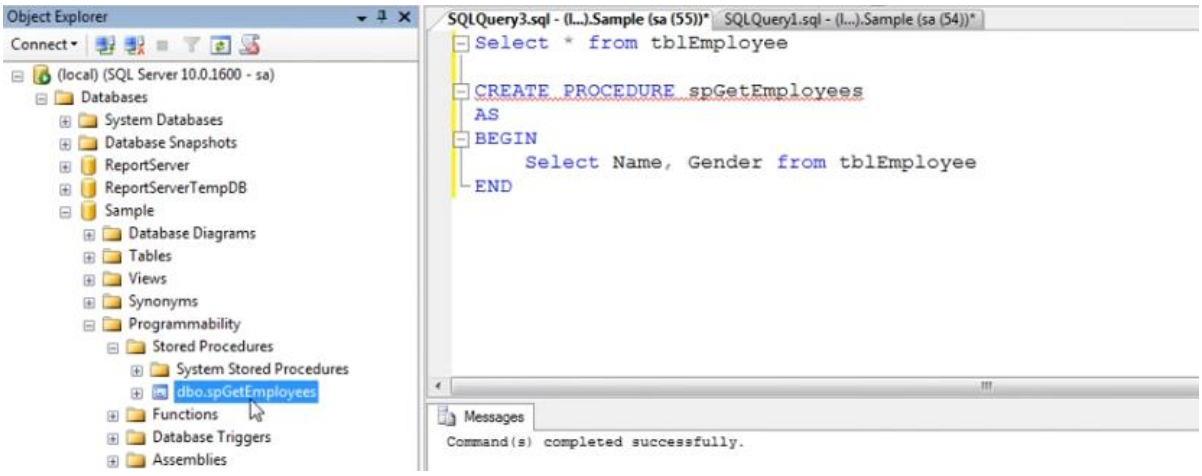
- What is a stored procedure
- Stored Procedure example
- Creating a stored procedure with parameters
- Altering SP
- Viewing the text of the SP
- Dropping the SP

Stored Procedures

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.

1. Use CREATE PROCEDURE or CREATE PROC statement to create SP

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.



Executing Stored Procedure

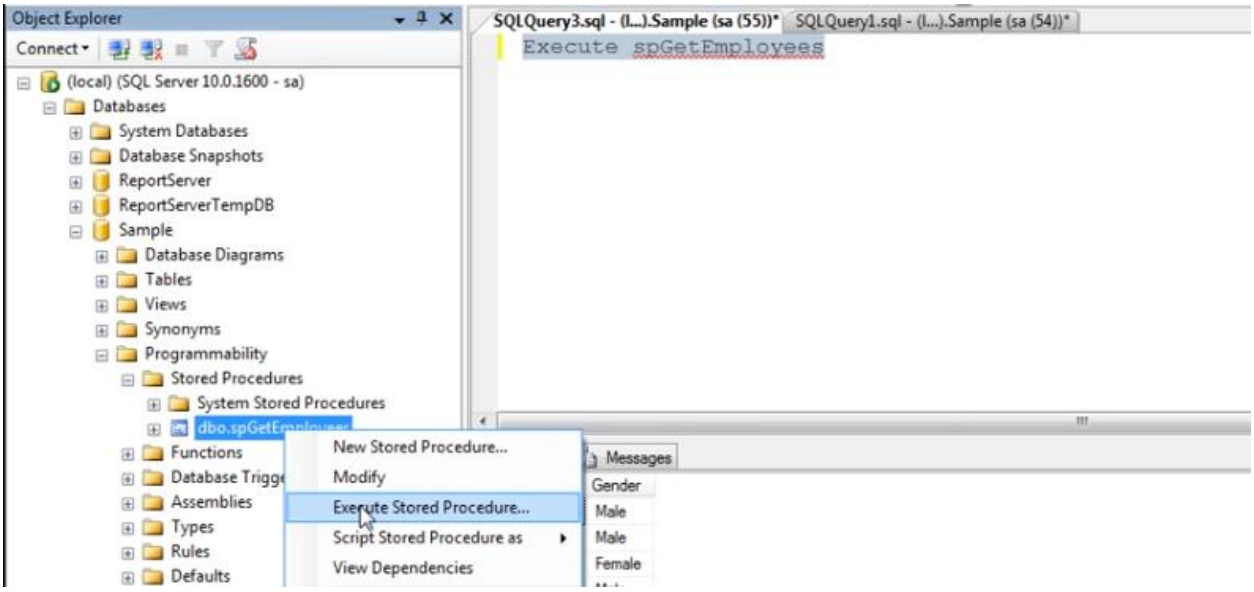
To execute the stored procedure

1. **spGetEmployees**
2. **EXEC spGetEmployees**
3. **Execute spGetEmployees**

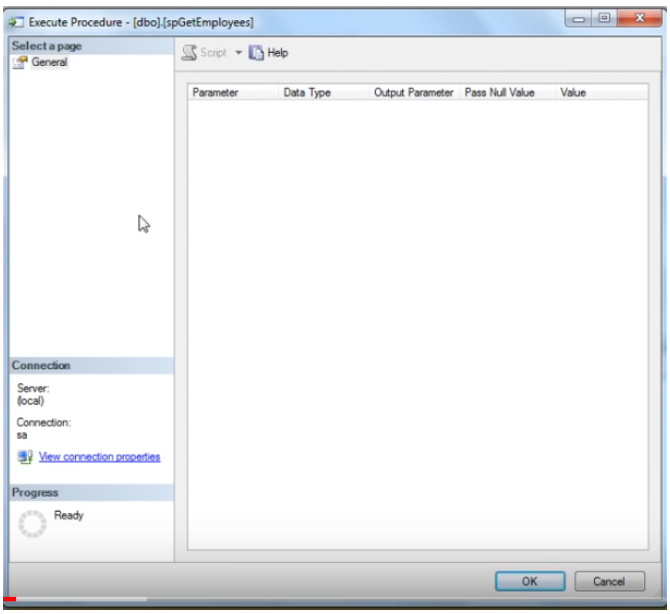
Note: You can also right click on the procedure name, in object explorer in SQL Server Management Studio and select EXECUTE STORED PROCEDURE.

Executing Stored Procedure Graphically using SSMS

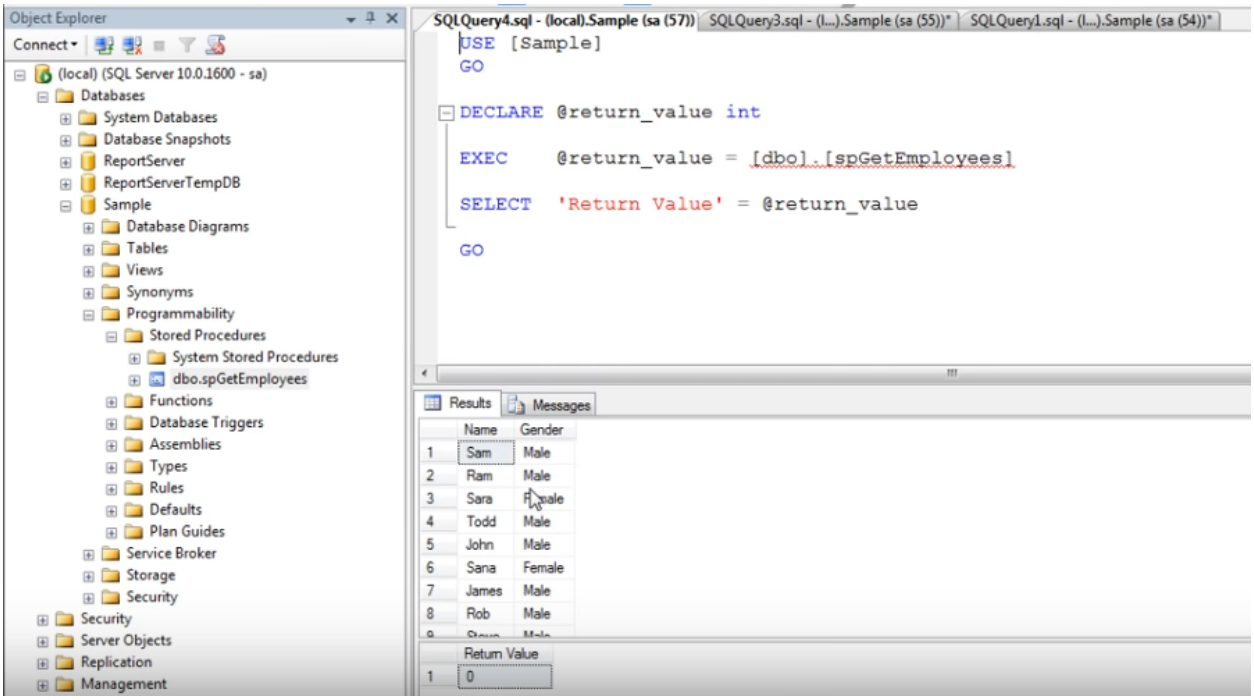
Right click on Stored Procedure then select "Execute Stored Procedure"



Execute Procedure dialog opens up. The procedure that we created does not have any parameter otherwise we will have to supply values for those parameters also.

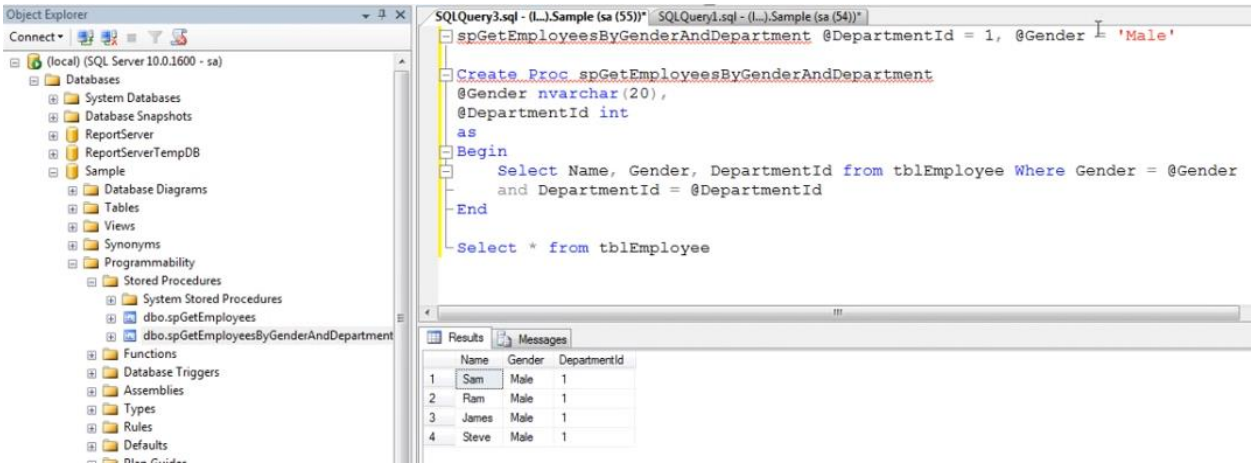


When we click OK, then it executes the stored procedure.



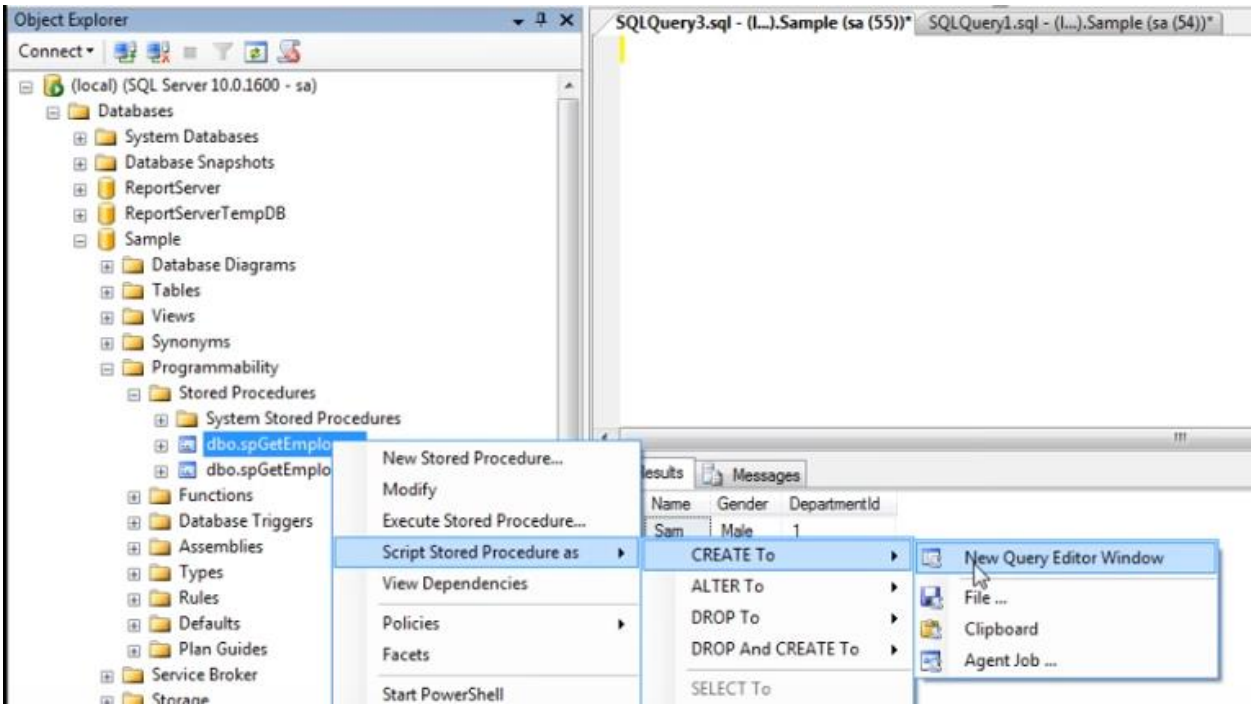
Creating and Executing Stored Procedures that accept Parameters

In the procedure below the user can pass @Gender and @DepartmendId parametes.

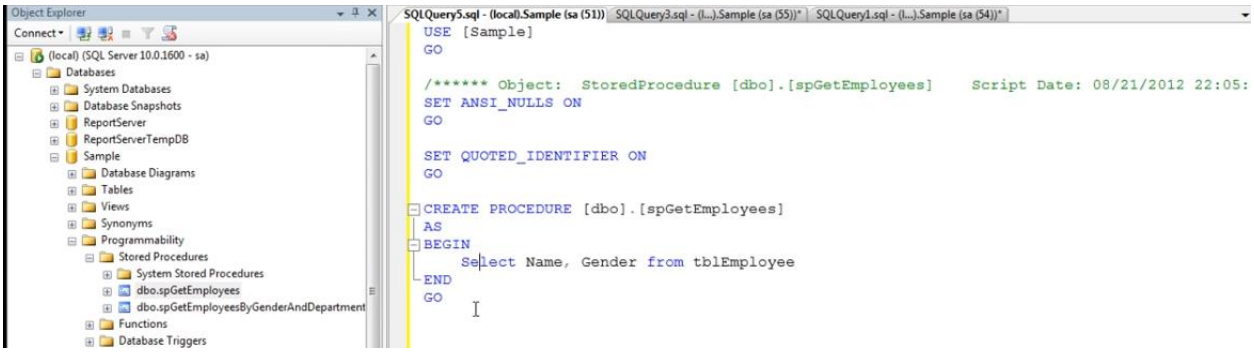


Viewing the Text of Stored Procedure in SSMS

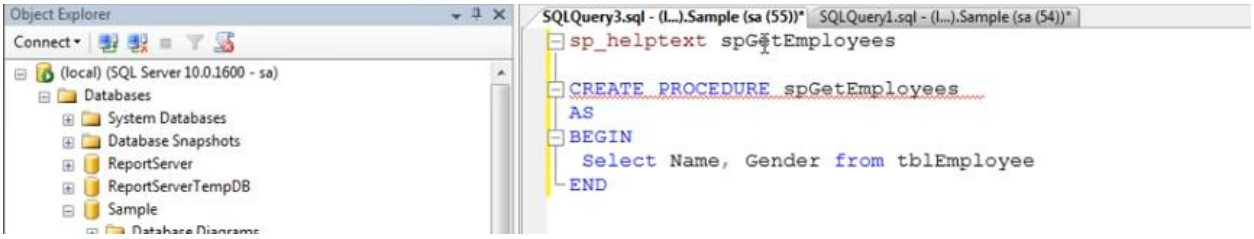
To view the text, of the stored procedure
1. Use system stored procedure sp_helptext 'SPName'
OR
2. Right Click the SP in Object explorer -> Scrip Procedure as -> Create To -> New Query Editor Window



This opens up a new query window and displays the Text of Stored Procedure



The other way is to use query sp_helptext and then stored procedure name



Encrypted Stored Procedure

Once the stored procedure is encrypted one cannot view the Text of the stored procedure.

```
Alter Proc spGetEmployeesByGenderAndDepartment
@Gender nvarchar(20),
@DepartmentId int
WITH Encryption --With Encryption does not allow the user to see the text of stored procedure using sp_helptext
as
BEGIN
 Select Name, Gender, Id from Department_Name Where Gender = @Gender
 and Id = @DepartmentId
END
```

Stored Procedures with Output Parameters

- Creating and executing stored procedures with output parameters
- Learn about some of the useful system stored procedures

Stored Procedures – Output Parameters

To create an SP with output parameter, we use the keywords OUT or OUTPUT.

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

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

To execute the stored procedure with output parameters

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender 'Male', @EmployeeTotal Output
Print   @EmployeeTotal
```

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

If you don't specify the OUTPUT keyword, when executing the stored procedure, the @EmployeeTotal variable will be NULL.

Count result is saved in output parameter @EmployeeCount

SQLQuery8.sql - (l...).Sample (sa (52))*

```
Select * from tblEmployee

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

Declare @TotalCount int
Execute spGetEmployeeCountByGender 'Male', @TotalCount out
if(@TotalCount is null)
    Print '@TotalCount is null'
else
    Print '@TotalCount is not null'

Print @TotalCount
```

```sql
Select * from tblEmployee

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

Declare @TotalCount int
Execute spGetEmployeeCountByGender 'Male', @TotalCount Output
Print @TotalCount
```

```sql
Select * from tblEmployee

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

Declare @TotalCount int
Execute spGetEmployeeCountByGender @EmployeeCount = @TotalCount out, @Gender = 'Male'
Print @TotalCount
```

```sql
--WHEN A STORED PROCEDURE RETURNS A OUTPUT PARAMETER, WE NEED TO CREATE AN VARIABLE
Create proc spGetTotalCount2
@TotalCount int Output
as
Begin
    Select @TotalCount = Count(Id) from Department_Name
End
Declare @Total int
Execute spGetTotalCount2 @Total Output
print @Total
--When A STORED PROCEDURE RETURNS A VALUE, WE MUST CREATE VARIABLE TO STORE THE RETURNED VALUE
Create proc spGetTotalCount3
as
Begin
    return (select count(Id) from Department_Name)
End
Declare @TotalEmp int
Execute @TotalEmp = spGetTotalCount3
print @TotalEmp
```

Useful system stored procedures

sp_help procedure_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 procedure_name - View the Text of the stored procedure

sp_depends procedure_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.

Stored procedure return values

- Understand what are stored procedure return values
- Difference between stored procedure return values and output parameters
- When to use output parameters over return values

Output Parameters or Return Values

Whenever, you execute a stored procedure, it returns an integer status variable. Usually, zero indicates success, and non-zero indicates failure.

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

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

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

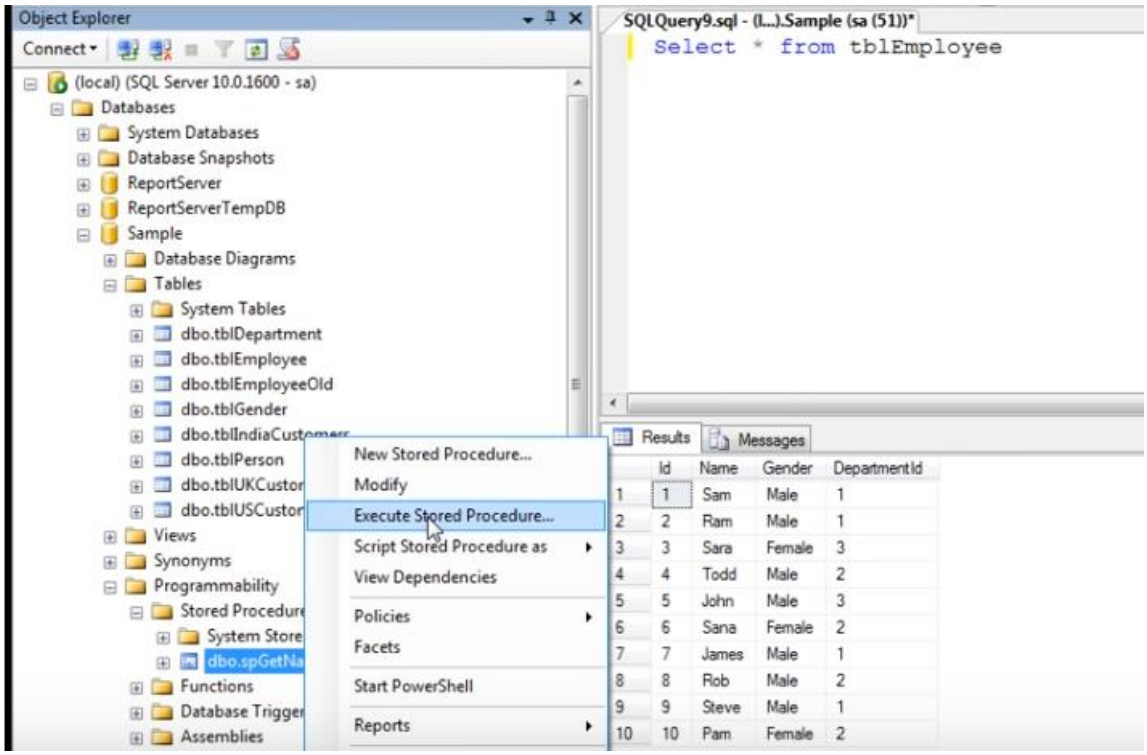
```
Create Procedure spGetTotalCountOfEmployees2
as
Begin
  return (Select COUNT(ID) from Employees)
End
```

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

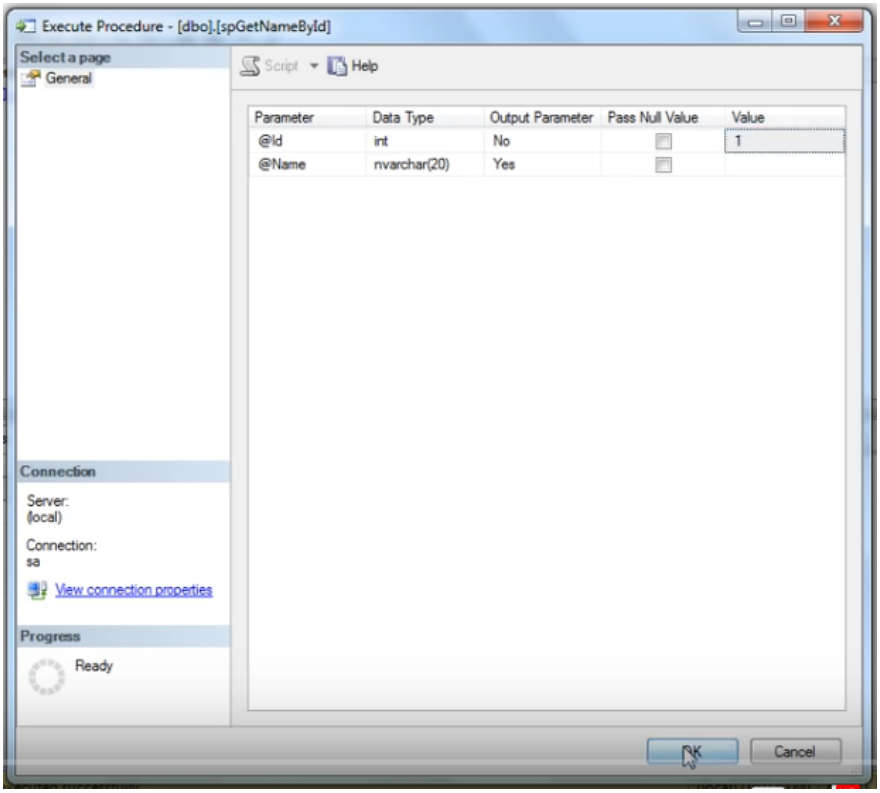
So, we are able to achieve what we want, using output parameters as well as return values.

Now, let's look at example, where return status variables cannot be used, but Output parameters can be used.

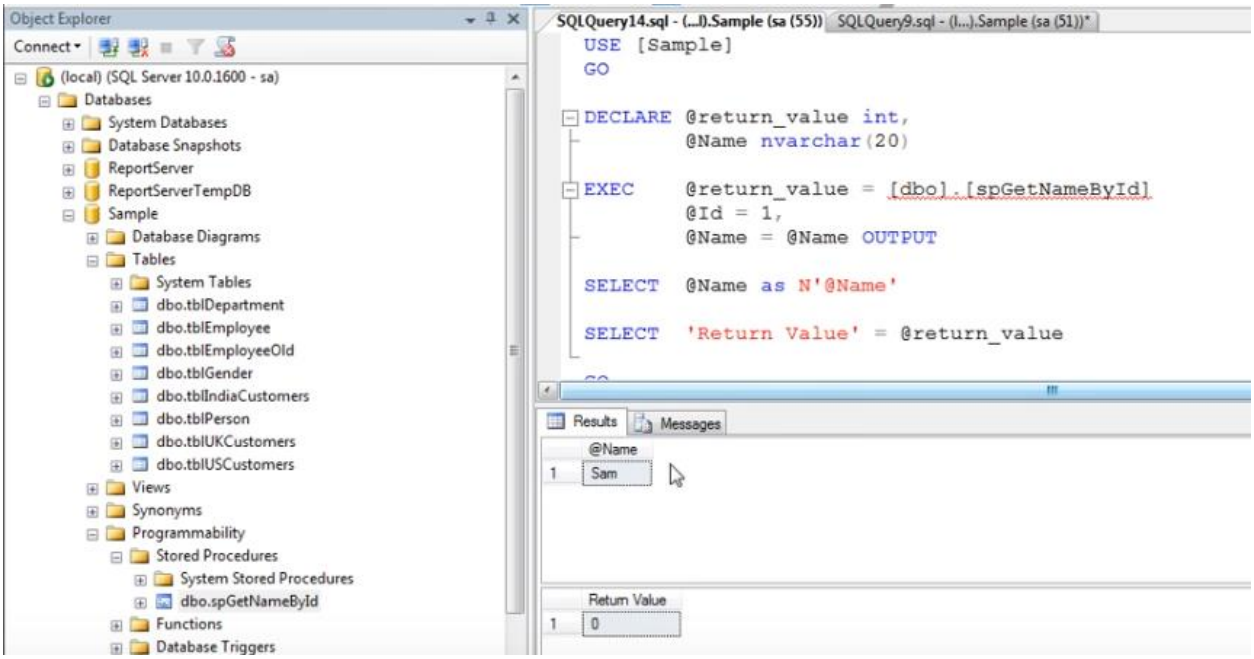
When we want to execute a stored procedure, in the SSMS we right click on stored procedure name then click Execute Stored Procedure.



Execute Procedure dialog opens up, we provide the input parameter value for Id as 1 then click OK.



The stored procedure gets executed and we get the Output parameter value @Name and a Return Value.



Difference between Output Parameters and Return Values

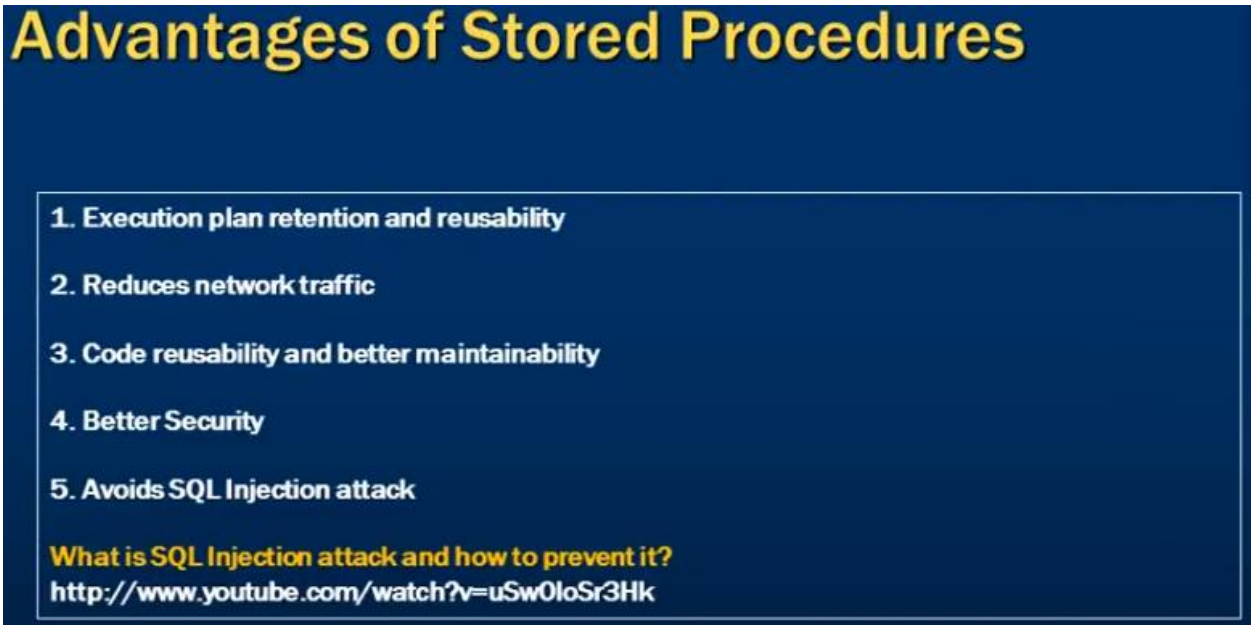
Return Staus Value	Output Parameters
Only Integer Datatype	Any Datatype
Only one value	More than value
Use to convey success or failure	Use to return values like name, count etc..

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

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

Advantages of stored procedures

- Advantages of using Stored Procedures over adhoc queries (inline SQL)



When we issue a query to SQL Server then 3 things happen:

1. First it checks the syntax of the query
2. Second, it compiles that query
3. Third, it generates an execution plan

What is an Execution Plan?

- For the query to retrieve the data what is the best possible way available. This depends upon Indexes that are available to help that query. Based on all these factors it is going to generate an execution plan and then it executes the query.
- Next time we issue the same query it has the execution plan available cached by sql server. It reuses that execution plan, the stored procedures are capable of caching the execution plan.
- However, even a small change in query can prevent the execution plan from being reused.

For better security specific persons can be granted access to specific stored procedures that access tables with sensitive information without granting access to the table itself. This enables fine grain control over what data a specific employee can see.

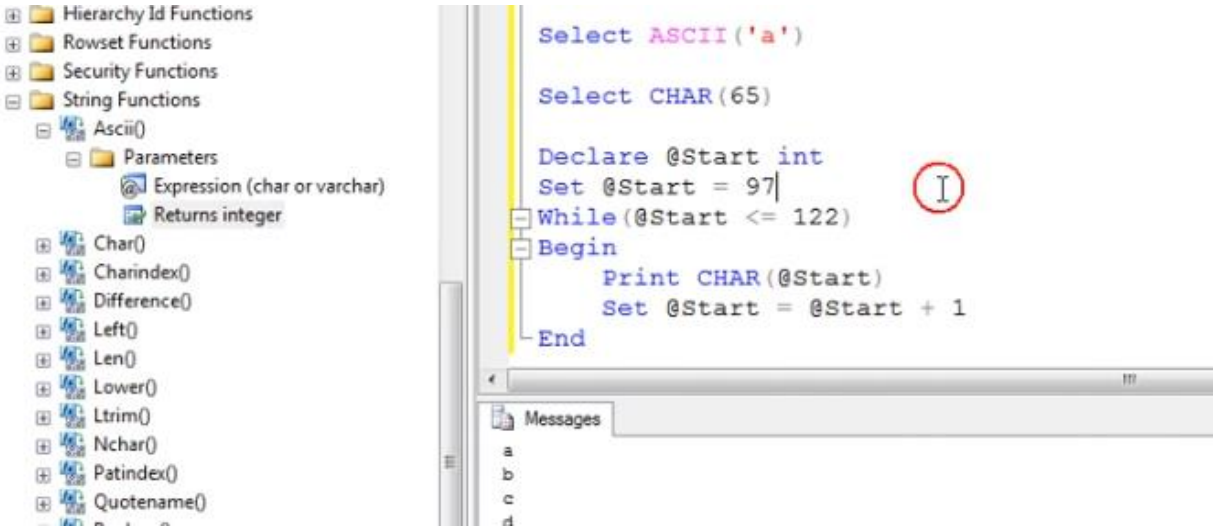
People can type malicious sql code in user interfaces and execute it. To prevent this stored procedures are helpful.

String Functions

String functions

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.

Function	Purpose
LEFT(Character_Expression, Integer_Expression)	Returns the specified number of characters from the left hand side of the given character expression.
RIGHT(Character_Expression, Integer_Expression)	Returns the specified number of characters from the right hand side of the given character expression.
CHARINDEX('Expression_To_Find', 'Expression_To_Search', 'Start_Location')	Returns the starting position of the specified expression in a character string
SUBSTRING('Expression', 'Start', 'Length')	Returns substring (part of the string), from the given expression



```
Select LTRIM(FirstName) as FirstName, MiddleName, LastName,
FirstName + ' ' + MiddleName + ' ' + LastName as FullName
 from tblEmployee
```

Results | Messages

FirstName	MiddleName	LastName	FullName
Sam	S	Sony	Sam S Sony
Ram	R	Barber	Ram R Barber
Sara		Sanosky	Sara Sanosky
Todd		Gartner	Todd Gartner
John	J	Grover	John J Grover
Sana	S	Lenin	Sana S Lenin
James	J	Bond	James J Bond
Rob	R	Hunter	Rob R Hunter
Steve	S	Wilson	Steve S Wilson

```
Select RTRIM('    Hello    ')

Select LTRIM(FirstName) as FirstName, MiddleName, LastName,
RTRIM(LTRIM(FirstName)) + ' ' + MiddleName + ' ' + LastName as FullName
 from tblEmployee
```

Results | Messages

FirstName	MiddleName	LastName	FullName
Sam	S	Sony	Sam S Sony
Ram	R	Barber	Ram R Barber
Sara		Sanosky	Sara Sanosky
Todd		Gartner	Todd Gartner
John	J	Grover	John J Grover
Sana	S	Lenin	Sana S Lenin
James	J	Bond	James J Bond
Rob	R	Hunter	Rob R Hunter
Steve	S	Wilson	Steve S Wilson

```
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))
```

Results | Messages

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

LEFT, RIGHT, CHARINDEX and SUBSTRING

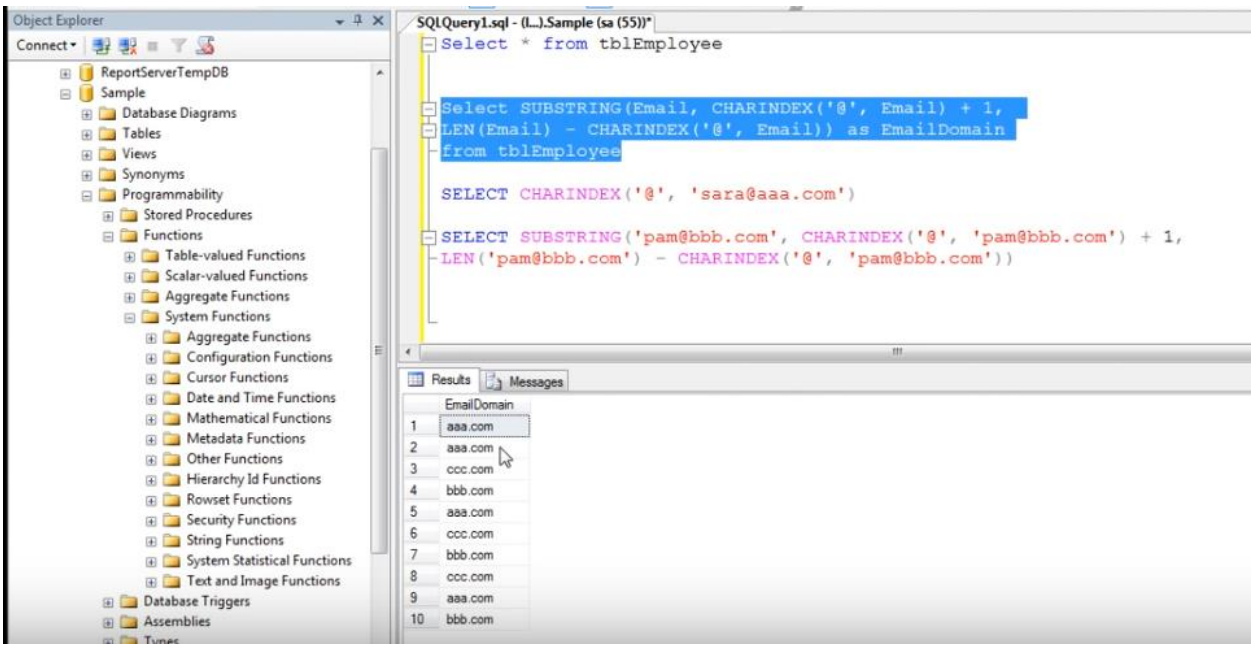
String functions

Function	Purpose
LEFT(Character_Expression, Integer_Expression)	Returns the specified number of characters from the left hand side of the given character expression.
RIGHT(Character_Expression, Integer_Expression)	Returns the specified number of characters from the right hand side of the given character expression.
CHARINDEX('Expression_To_Find', 'Expression_To_Search', 'Start_Location')	Returns the starting position of the specified expression in a character string
SUBSTRING('Expression', 'Start', 'Length')	Returns substring (part of the string), from the given expression

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

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

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




```
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

- Few more commonly used built-in string functions in SQL server
  - Replicate
  - Space
  - Patindex
  - Replace
  - Stuff

REPLICATE Function

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

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

→

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

Mask the email with 5 * (star) symbols

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

SPACE Function

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

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

→

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

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

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

STUFF Function

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.

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

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

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

REPLACE Function

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

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

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

All .COM strings are replaced with .NET

```
Select  Email, REPLACE(Email, '.com', '.net') as ConvertedEmail
from    tblEmployee
```

PATINDEX Function

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.

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

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

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

SQLQuery3.sql - LA...B9U5G9\Gauri (53))* | SQLQuery2.sql - not connected

```
Select CHARINDEX('@', 'ssf@h.com')

Select SUBSTRING('pam@bbb.com', CHARINDEX('@', 'pam@bbb.com') + 1, LEN('pam@bbb.com') - CHARINDEX('@', 'pam@bbb.com'))

Select REPLICATE('Gauri ', 3)

--REPLICATE FUNCTION
Select Name, City,
    SUBSTRING(Email, 1, 2) + REPLICATE('*', 5) +
    SUBSTRING(Email, CHARINDEX('@',Email) + 1, LEN(Email) - CHARINDEX('@', Email) + 1) as Email
from tbl_persontbl

--SPACE FUNCTION
Select Space(5)
Select Name + SPACE(5) + DepartmentName as Employee_Detail from Department_Name

--PATINDEX FUNCTION
Select Email, PATINDEX('%com', Email) as FirstOccurence
from tbl_persontbl
where PATINDEX('%com', Email) >0
```

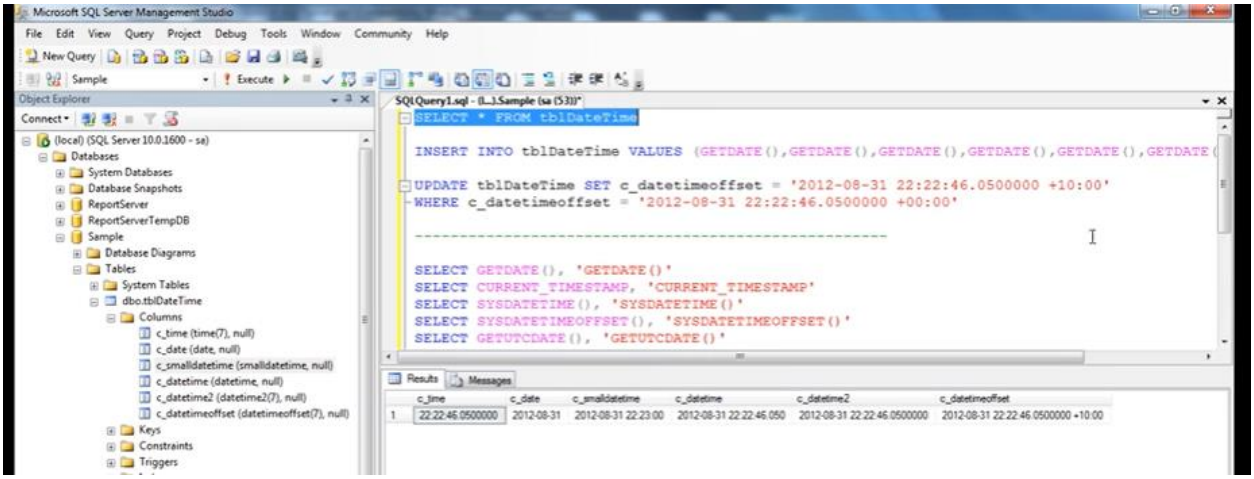
DateTime functions

DateTime

Data type	Format	Range	Accuracy	Storage size (bytes)
time	hh:mm:ss[.nnnnnnn]	00:00:00.0000000 through 23:59:59.9999999	100 nanoseconds	3 to 5
date	YYYY-MM-DD	0001-01-01 through 9999-12-31	1 day	3
smalldatetime	YYYY-MM-DD hh:mm:ss	1900-01-01 through 2079-06-06	1 minute	4
datetime	YYYY-MM-DD hh:mm:ss[.nnn]	1753-01-01 through 9999-12-31	0.00333 second	8
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999	100 nanoseconds	6 to 8
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 (in UTC)	100 nanoseconds	8 to 10

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.

Function	Date Time Format	Description
GETDATE()	2012-08-31 20:15:04.543	Commonly used
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



IsDate, Day, Month, Year and DateName DateTime functions

- IsDate, Day, Month, Year and DateName
- A practical example of using these functions

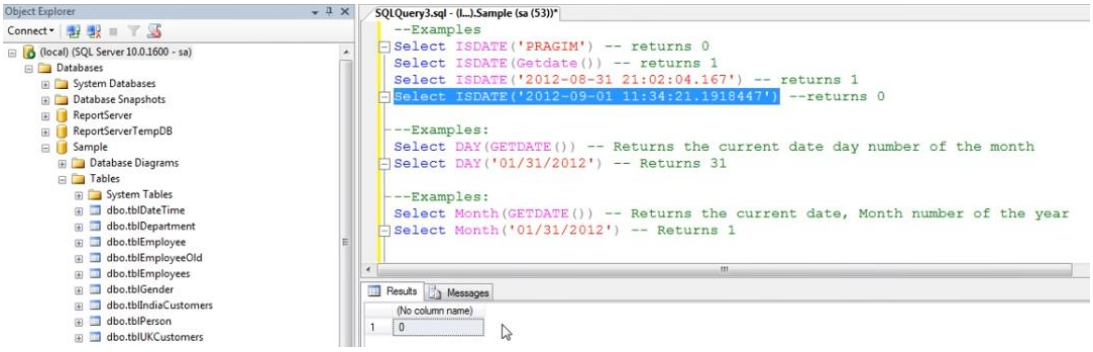
ISDATE

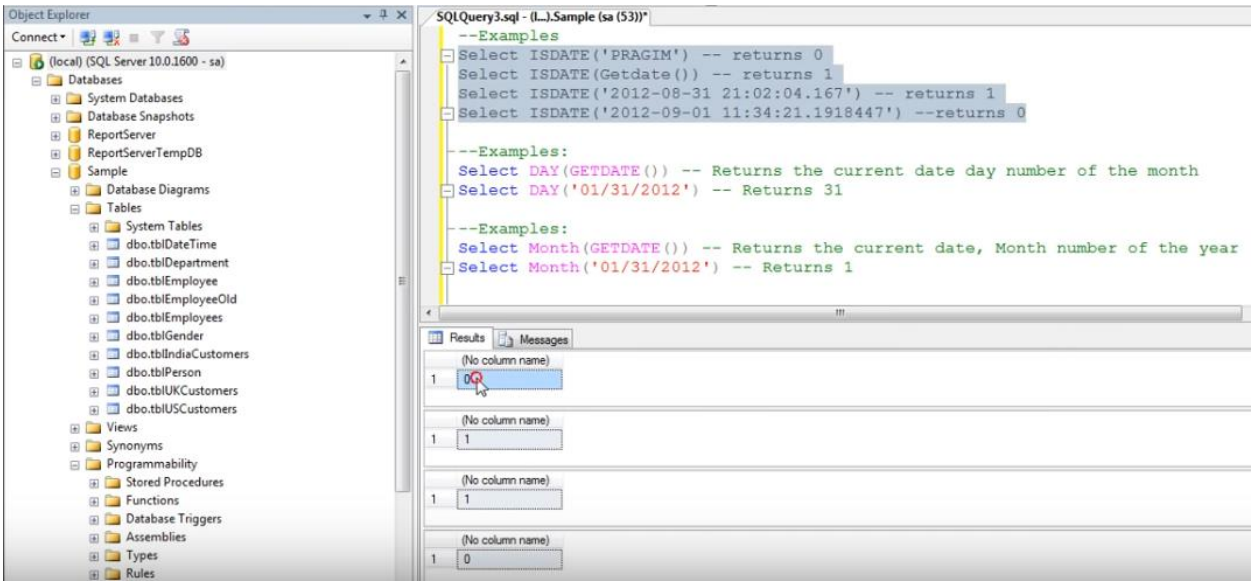
Function	Date Time Format	Description
GETDATE()	2012-08-31 20:15:04.543	Commonly used
CURRENT_TIMESTAMP	2012-08-31 20:15:04.543	ANSI SQL equivalent to GETDATE
SYSDATETIME()	2012-08-31 20:15:04.5380028	More fractional seconds precision
SYSDATETIMEOFFSET()	2012-08-31 20:15:04.5380028 + 01:00	More fractional 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

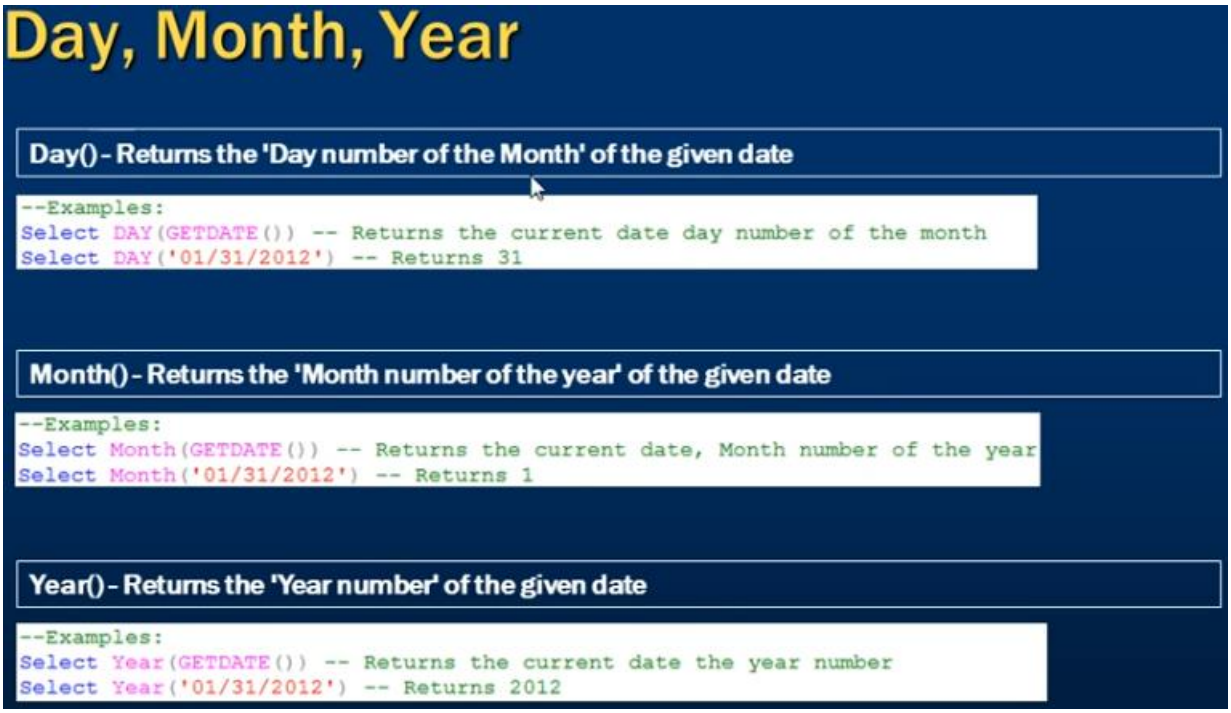
ISDATE() - Checks if the given value, is a valid date, time, or datetime. Returns 1 for success, 0 for failure.

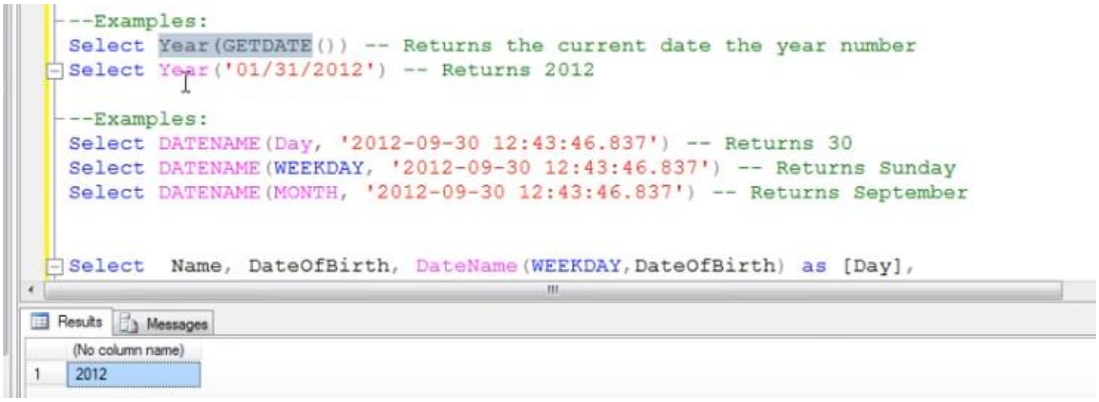
```
--Examples
Select ISDATE('PRAGIM') -- returns 0
Select ISDATE(Getdate()) -- returns 1
Select ISDATE('2012-08-31 21:02:04.167') -- returns 1
Select ISDATE('2012-09-01 11:34:21.1918447') --returns 0
```

Note: For datetime2 values, IsDate returns ZERO.




Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.tblDateTime
dbo.tblDepartment
dbo.tblEmployee
dbo.tblEmployeeOld
dbo.tblEmployees
dbo.tblGender
dbo.tblIndiaCustomers
dbo.tblPerson
dbo.tblUKCustomers
dbo.tblUSCustomers
Views
Synonyms
Programmability
Stored Procedures
Functions
Database Triggers
Assemblies
Types
Rules
SQLQuery3.sql - (l...).Sample (sa (53))*
--Examples
Select ISDATE('PRAGIM') -- returns 0
Select ISDATE(Getdate()) -- returns 1
Select ISDATE('2012-08-31 21:02:04.167') -- returns 1
Select ISDATE('2012-09-01 11:34:21.1918447') --returns 0
--Examples:
Select DAY(GETDATE()) -- Returns the current date day number of the month
Select DAY('01/31/2012') -- Returns 31
--Examples:
Select Month(GETDATE()) -- Returns the current date, Month number of the year
Select Month('01/31/2012') -- Returns 1
Results
Messages
(No column name)
1 0
(No column name)
1 1
(No column name)
1 1
(No column name)
1 0


Day, Month, Year
Day() - Returns the 'Day number of the Month' of the given date
--Examples:
Select DAY(GETDATE()) -- Returns the current date day number of the month
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 current date, Month number of the year
Select Month('01/31/2012') -- Returns 1
Year() - Returns the 'Year number' of the given date
--Examples:
Select Year(GETDATE()) -- Returns the current date the year number
Select Year('01/31/2012') -- Returns 2012


--Examples:
Select Year(GETDATE()) -- Returns the current date the year number
Select Year('01/31/2012') -- Returns 2012
--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
Select Name, DateOfBirth, DateName(WEEKDAY,DateOfBirth) as [Day],
Results
Messages
(No column name)
1 2012

DateName

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.

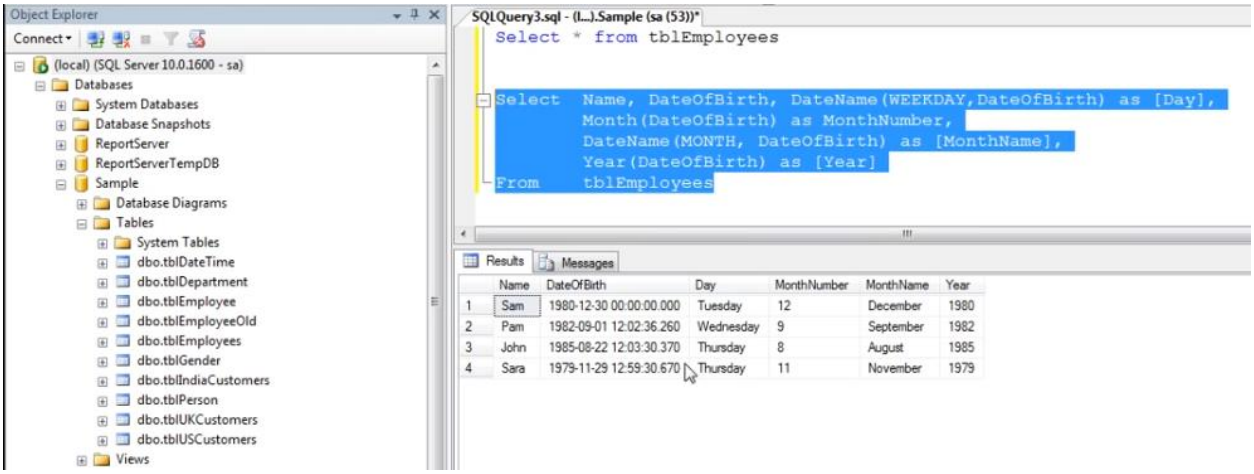
```
--Examples:
Select DATENAME(Day, '2012-09-30 12:43:46.837') -- Returns 30
Select DATENAME(WEEKDAY, '2012-09-30 12:43:46.837') -- Returns Sunday
Select DATENAME(MONTH, '2012-09-30 12:43:46.837') -- Returns September
```

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

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

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

```
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

DatePart, DateAdd and DateDiff

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.

```
--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
```

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

Calculating Age

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

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

```
DECLARE @DOB datetime, @tmpdate datetime, @years int, @months int, @days int
SET @DOB = '10/08/1982'

SELECT @tmpdate = @DOB

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

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

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

SELECT @years as Years, @months as Months, @days as [Days]
```

SQLQuery5.sql - (l...).Sample (sa (54))* SQLQuery4.sql - (l...).Sample (sa (55))*

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

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())
```

```
SELECT @tempdate = DATEADD(MONTH, @months, @tempdate)

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

    Declare @Age nvarchar(50)
    Set @Age = @years + ' Years ' + @months + ' Months ' +  @days + ' Days Old'

    return @Age
End
```

SQLQuery5.sql - (l...).Sample (sa (54))* SQLQuery4.sql - (l...).Sample (sa (55))*

```
--12/01/2005
--01/01/2006

--Select DATEDIFF(DAY, '11/30/2005','01/31/2006') -- returns 2

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

Select dbo.fnComputeAge('11/30/2005')

Create function fnComputeAge(@DOB DateTime)
returns nvarchar(50)
as
Begin
```

Results | Messages

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

Cast and Convert functions

These functions convert an expression of one data type to another.

- CAST and CONVERT functions
- When to choose one over the other

CAST & CONVERT

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 ] )
```

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

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

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

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

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

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

http://msdn.microsoft.com/en-us/library/ms187928.aspx

Difference CAST - CONVERT

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.

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

SQLQuery1.sql - (L...).Sample (sa (55))*

```
Select * from tblEmployees

Select Id, Name, DateOfBirth, CAST(DateofBirth as nvarchar(20)) as ConvertedDOB from tblEmployees
Select Id, Name, DateOfBirth, Convert(nvarchar, DateOfBirth) as ConvertedDOB from tblEmployees
```

Results | Messages

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

DatePart of DateTime

```
--To get just the date part, from DateTime
SELECT CONVERT(VARCHAR(10),GETDATE(),101) -- Returns 09/02/2012

--In SQL Server 2008, Date datatype is introduced, so you can also use
SELECT CAST(GETDATE() as DATE)    -- 2012-09-02
SELECT CONVERT(DATE, GETDATE())   -- 2012-09-02
```

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.

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

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

Practical Example

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

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

```
Select      CAST(RegisteredDate as DATE)as RegistrationDate, COUNT(Id) as TotalRegistrations
from        tblRegistrations
Group BY    CAST(RegisteredDate as DATE)
```

```
Select      RegisteredDate as RegistrationDate, COUNT(Id) as TotalRegistrations
from        tblRegistrations
Group BY    RegisteredDate
```

RegistrationDate	TotalRegistrations
2012-08-24 11:04:30.230	1
2012-08-24 15:04:30.730	1
2012-08-24 15:05:30.330	1
2012-08-25 14:04:29.780	1
2012-08-25 15:04:29.780	1
2012-08-26 15:05:30.330	1

Mathematical functions

- Mathematical functions
- Abs
- Ceiling
- Floor
- Power
- Rand
- Square
- Sqrt
- Round

Mathematical Functions

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

```
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.

```
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.

```
Select POWER(2,3) -- Returns 8
```

SQUARE (Number)

Returns the square of the given number.

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

SQRT (Number)

Returns the square root of the given number

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

SQLQuery3.sql - (l...).Sample (sa (51))* | SQLQuery2.sql - not connected* | SQLQuery1.sql - not connected*

```
Select ABS(-101.5) -- returns 101.5, without the - sign.

Select CEILING(15.2) -- Returns 16
Select CEILING(-15.2) -- Returns -15

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

Select POWER(2,3) -- Returns 8

Select SQUARE(9) -- Returns 81

Select SQRT(81) -- Returns 9

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

Select FLOOR(RAND() * 100)

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

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

RAND() function

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.

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

Generate a random number between 1 and 100

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

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
```

SQLQuery3.sql - (L...).Sample (sa (51))* | SQLQuery2.sql - not connected* | SQLQuery1.sql - not connected*

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

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

Select FLOOR(RAND() * 100)

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

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

Messages

```
73
68
22
11
99
54
90
73
72
39
```

ROUND() function

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. 0 indicates rounding, non zero indicates truncation. Default, if not specified is 0.

```
-- 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
```

User defined functions

- User Defined Functions
- Types of User Defined Functions
- Creating a Scalar User Defined Function
- Calling a Scalar User Defined Function
- Places where we can use Scalar User Defined Function
- Altering and Dropping User Defined Function

SCALAR UDF

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

In SQL Server there are 3 types of User Defined functions

1. Scalar functions
2. Inline table-valued functions
3. Multi-statement table-valued functions

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

```
--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
```

SCALAR UDF

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

When calling a scalar user-defined function, you must supply a two-part name, OwnerName.FunctionName. dbo stands for database owner.

```
Select dbo.Age('10/08/1982')
```

You can also invoke it using the complete 3 part name, DatabaseName.OwnerName.FunctionName

```
Select SampleDB.dbo.Age( dbo.Age('10/08/1982')
```

SCALAR UDF

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

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 Select Where clauses
Select Name, DateOfBirth, dbo.Age(DateOfBirth) as Age from tblEmployees
Where dbo.Age(DateOfBirth) > 30
```

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

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.

SQLQuery1.sql - (L...).Sample (sa (52))*

```
CREATE FUNCTION CalculateAge(@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

Select dbo.CalculateAge('10/08/1982')
```

SAME FUNCTION AS STORED PROC:

```
CREATE PROC spCalculateAge
@DOB Date
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
SELECT @Age
END
```

Inline table valued functions

- Creating Inline Table Valued Function
- How to call an Inline Table Valued Function
- Where do we use inline table valued functions

Inline Table Valued Functions

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

SCALAR FUNCTION – RETURNS A SCALAR VALUE
INLINE TABLE VALUED FUNCTION – RETURNS A 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

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

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

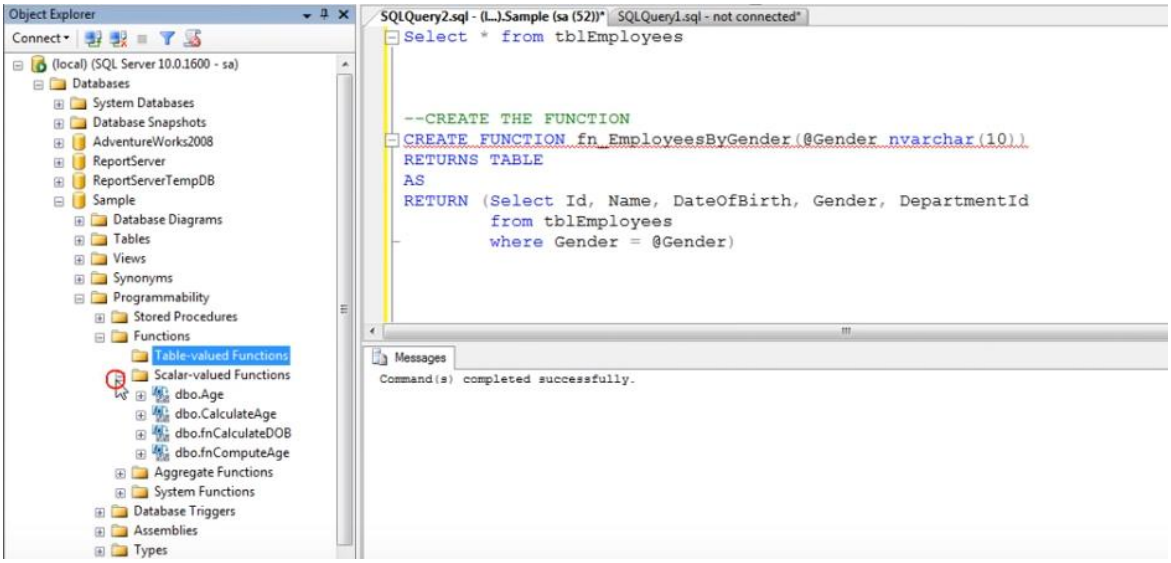
1. We specify TABLE as the return type, instead of any scalar data type
2. The function body is not enclosed between BEGIN and END block.
3. The structure of the table that gets returned, is determined by the SELECT statement with in the function.

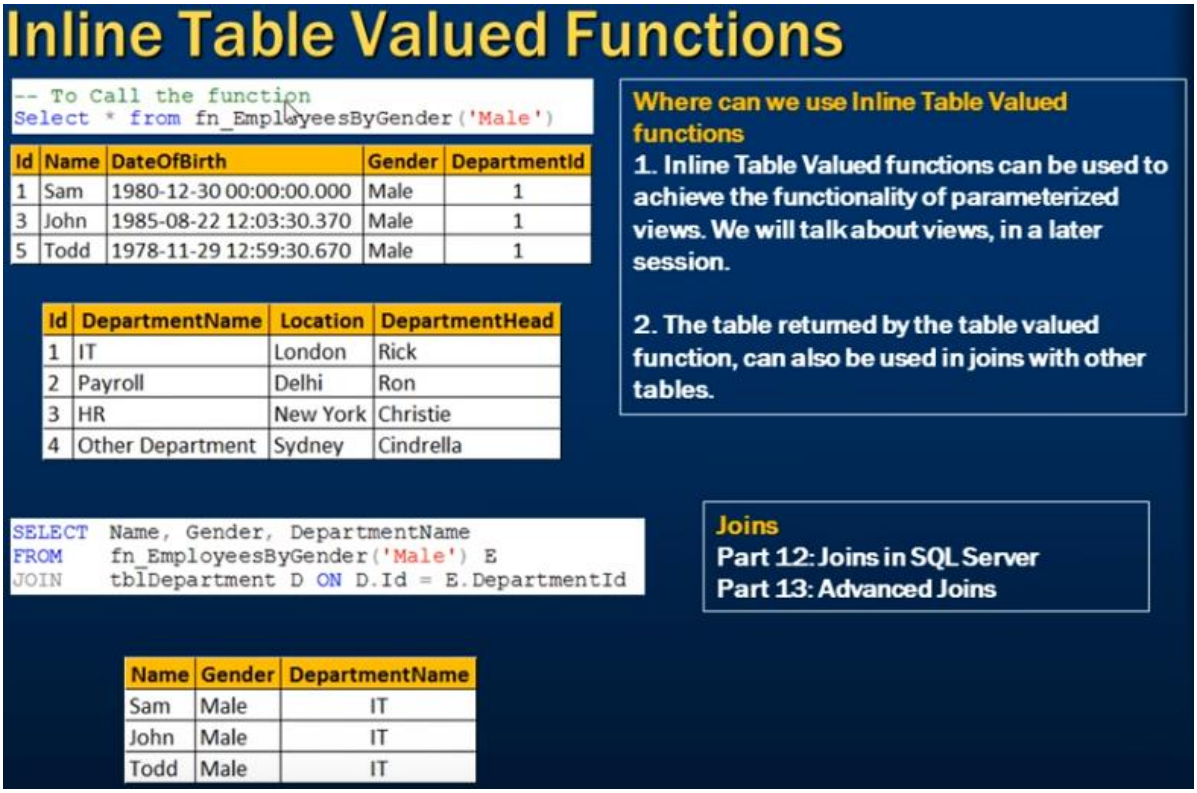
SQLQuery2.sql - (local).Sample (sa (52)) | SQLQuery1.sql - not connected*

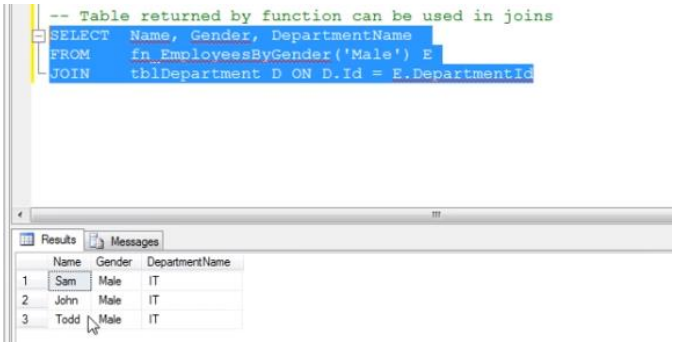
```
Select * from tblEmployees


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

Messages

```
Command(s) completed successfully.
```

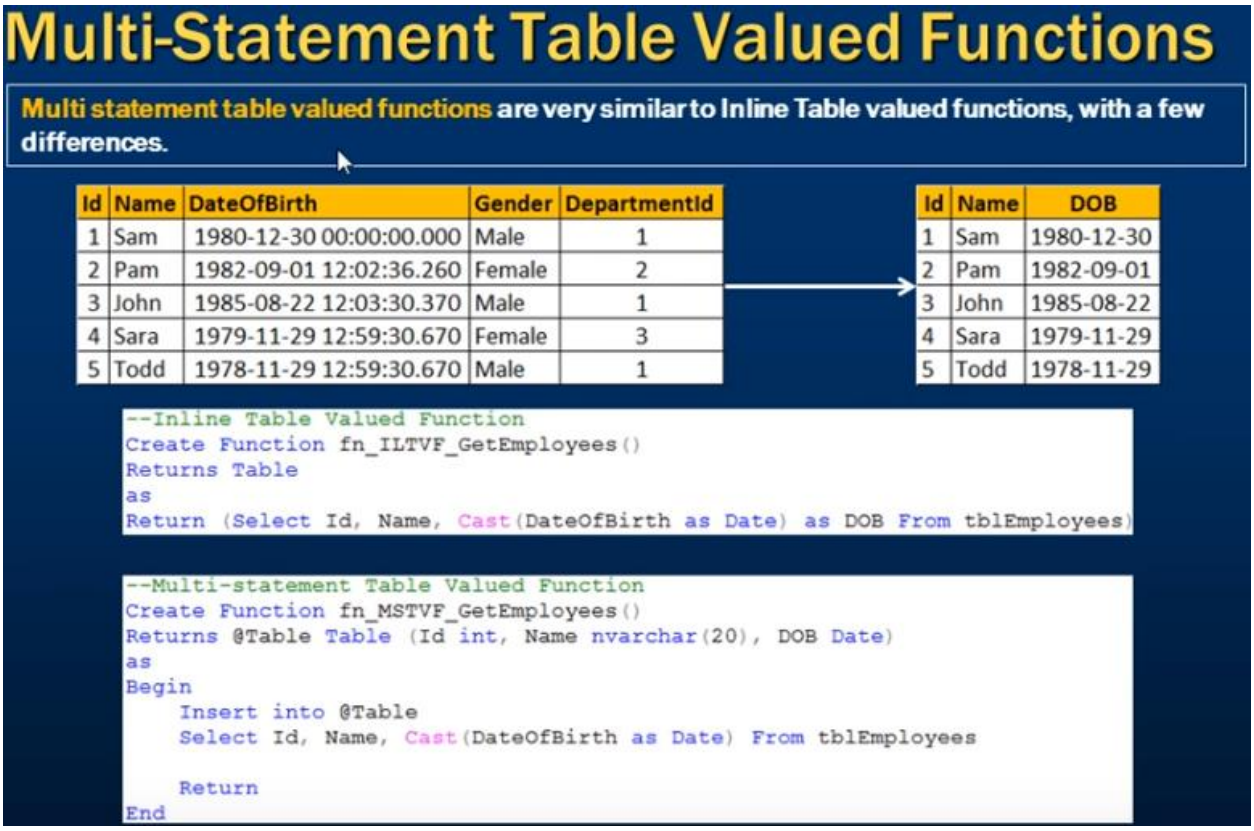

Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
AdventureWorks2008
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
Views
Synonyms
Programmability
Stored Procedures
Functions
Table-valued Functions
Scalar-valued Functions
dbo.Age
dbo.CalculateAge
dbo.fnCalculateDOB
dbo.fnComputeAge
Aggregate Functions
System Functions
Database Triggers
Assemblies
Types
SQLQuery2.sql - (L...).Sample (sa (52))*
SQLQuery1.sql - not connected*
Select * from tblEmployees
--CREATE THE FUNCTION
CREATE FUNCTION fn_EmployeesByGender(@Gender nvarchar(10))
RETURNS TABLE
AS
RETURN (Select Id, Name, DateOfBirth, Gender, DepartmentId
from tblEmployees
where Gender = @Gender)
Messages
Command(s) completed successfully.


Inline Table Valued Functions
-- To Call the function
Select * from fn_EmployeesByGender('Male')
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
Id DepartmentName Location DepartmentHead
1 IT London Rick
2 Payroll Delhi Ron
3 HR New York Christie
4 Other Department Sydney Cindrella
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.
SELECT Name, Gender, DepartmentName
FROM fn_EmployeesByGender('Male') E
JOIN tblDepartment D ON D.Id = E.DepartmentId
Joins
Part 12: Joins in SQL Server
Part 13: Advanced Joins
Name Gender DepartmentName
Sam Male IT
John Male IT
Todd Male IT

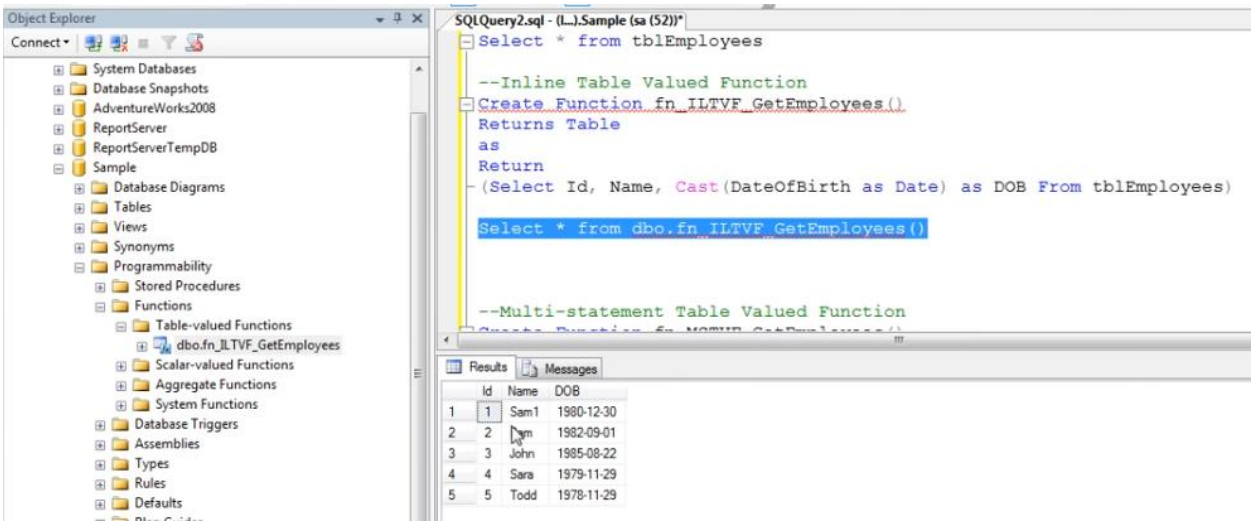

-- Table returned by function can be used in joins
SELECT Name, Gender, DepartmentName
FROM fn_EmployeesByGender('Male') E
JOIN tblDepartment D ON D.Id = E.DepartmentId
Results
Messages
Name Gender DepartmentName
1 Sam Male IT
2 John Male IT
3 Todd Male IT

Multi statement table valued functions

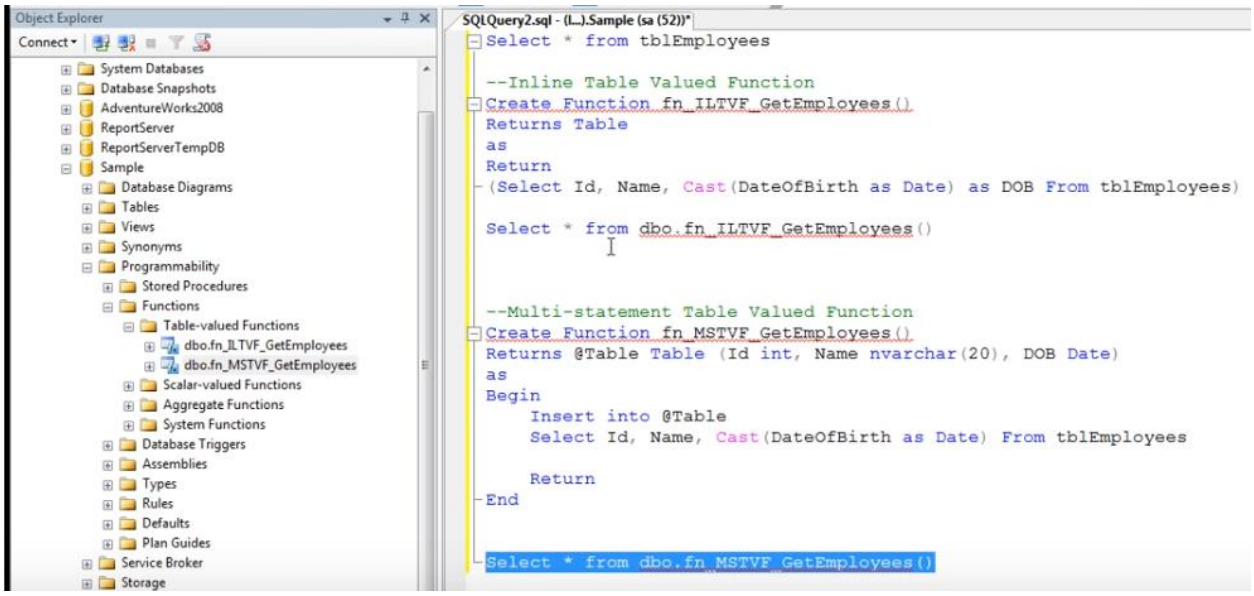
A **multi-statement table-valued function** or MSTVF is a **table-valued function** that returns the result of multiple **statements**. The **multi-statement-table-valued function** is very useful because you can execute multiple queries within the **function** and aggregate results into the returned **table**.

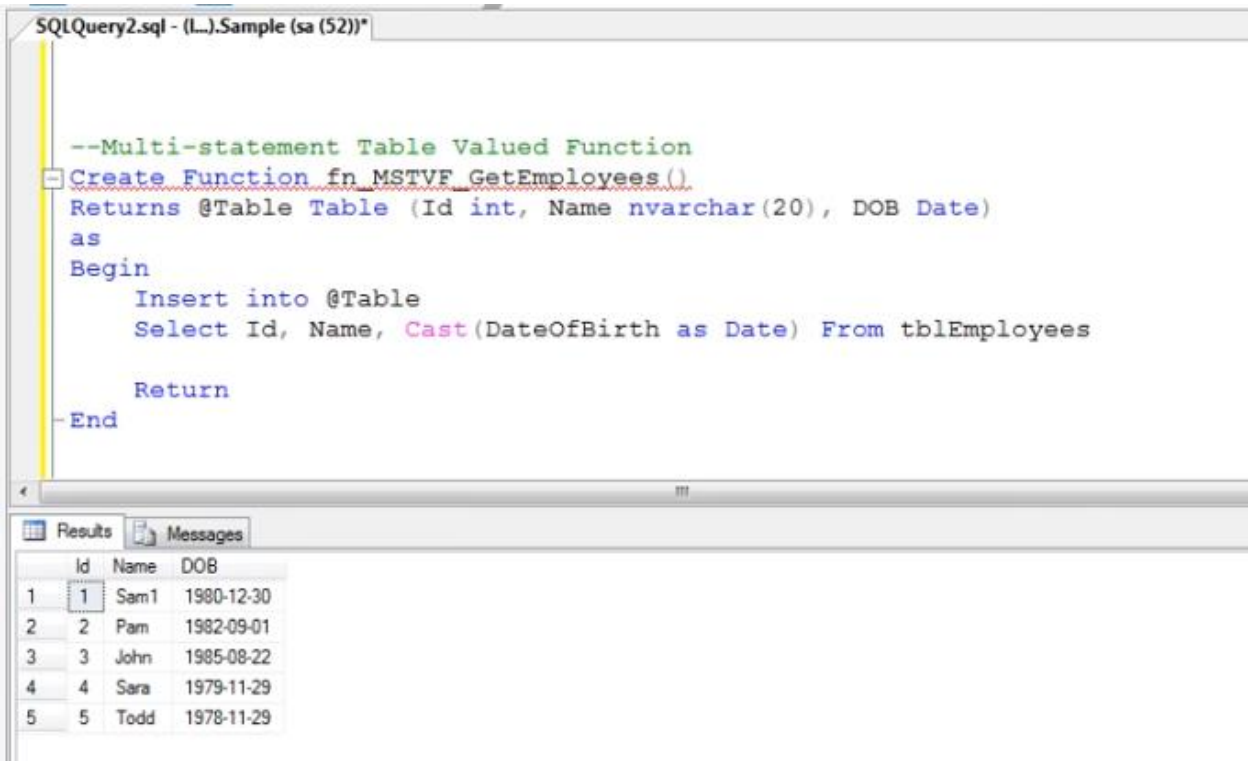


In the example above we are extracting Id, Name, DateOfBirth from tblEmployees and inserting the same into a Table and then returning that Table. First an Example below: **Inline Table Valued Function**



Similarly, we can use: **Multi Statement Table Valued** Function as shown below:


Object Explorer
Connect
System Databases
Database Snapshots
AdventureWorks2008
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
Views
Synonyms
Programmability
Stored Procedures
Functions
Table-valued Functions
dbo.fn_ILTVF_GetEmployees
dbo.fn_MSTVF_GetEmployees
Scalar-valued Functions
Aggregate Functions
System Functions
Database Triggers
Assemblies
Types
Rules
Defaults
Plan Guides
Service Broker
Storage
SQLQuery2.sql - (l...).Sample (sa (52))*
Select * from tblEmployees
--Inline Table Valued Function
Create Function fn_ILTVF_GetEmployees()
Returns Table
as
Return
(Select Id, Name, Cast(DateOfBirth as Date) as DOB From tblEmployees)
Select * from dbo.fn_ILTVF_GetEmployees()
--Multi-statement Table Valued Function
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
Select * from dbo.fn_MSTVF_GetEmployees()


SQLQuery2.sql - (l...).Sample (sa (52))*
--Multi-statement Table Valued Function
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
Results
Messages
Id Name DOB
1 1 Sam1 1980-12-30
2 2 Pam 1982-09-01
3 3 John 1985-08-22
4 4 Sara 1979-11-29
5 5 Todd 1978-11-29

Differences

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.

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.

However, Updates can not be performed on multi-statement table valued functions.

SQLQuery2.sql - (l...).Sample (sa (52))*

```
--Multi-statement Table Valued Function
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


Select * from dbo.fn_MSTVF_GetEmployees()
Update fn_MSTVF_GetEmployees() set Name = 'Sam 1' Where Id = 1
```

Messages

```
Msg 270, Level 16, State 1, Line 1
Object 'fn_MSTVF_GetEmployees' cannot be modified.
```

Important concepts related to functions

- Deterministic and Nondeterministic Functions
- Encrypting functions
- Schema binding functions

Deterministic and Nondeterministic

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: Square(), Power(), Sum(), AVG() 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.

WITH ENCRYPTION and SCHEMABINDING

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.

Use WITH ENCRYPTION

Creating a function WITH SCHEMABINDING option:
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.

Use WITH SCHEMABINDING

Using system stored procedure sp_heltptext we cannot display Text of encrypted function.

```
select * from tblEmployees

Select dbo.fn_GetNameById(1)

sp_helptext fn_GetNameById

Alter function fn_GetNameById(@Id int)
Returns nvarchar(30)
WITH ENCRYPTION
as
Begin
 return (Select Name from tblEmployees Where Id = @Id)
End
```

```
Messages
The text for object 'fn_GetNameById' is encrypted.
```

Due to Schemabinding can not drop any table or make any changes to a table that is referenced in a function with Schemabinding:

```
select * from tblEmployees

drop table tblemployees

Select dbo.fn_GetNameById(1)

sp_helptext fn_GetNameById

Alter function fn_GetNameById(@Id int)
Returns nvarchar(30)
WITH SCHEMABINDING
as
Begin
 return (Select Name from dbo.tblEmployees Where Id = @Id)
```

```
Messages
Msg 3729, Level 16, State 1, Line 1
Cannot DROP TABLE 'tblemployees' because it is being referenced by object 'fn_GetNameById'.
```

Temporary tables

- What are temporary tables
- Types of temporary tables – Local and Global
- Difference between local and global temp tables

Temporary Tables

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

1. Local Temporary Tables
2. Global Temporary tables.

```
--Create Local Temporary Table
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 data from the temporary table:
Select * from #PersonDetails
```

Local Temporary Tables

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.

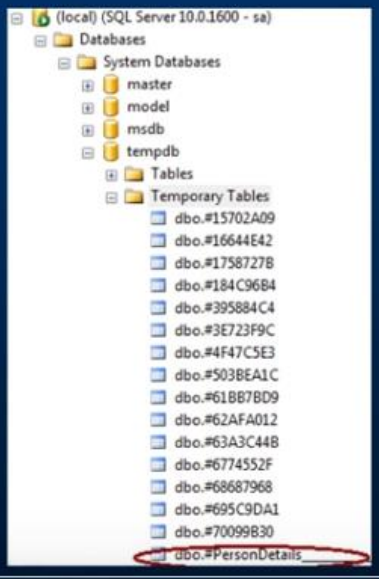
```
Select  name from tempdb..sysobjects
where   name like '#PersonDetails%'
```

A local temporary table is available, **only for the connection that has created the table**

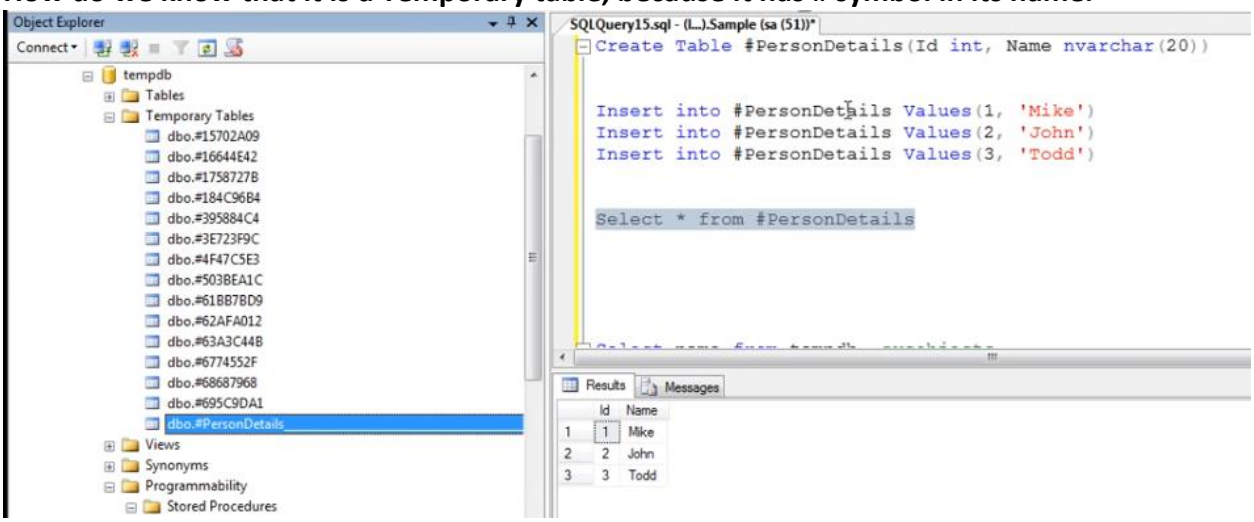
A local temporary table **is automatically dropped**, when the connection that has created the it, is closed.

If the user wants to explicitly drop the temporary table, he can do so using **DROP TABLE #PersonDetails**

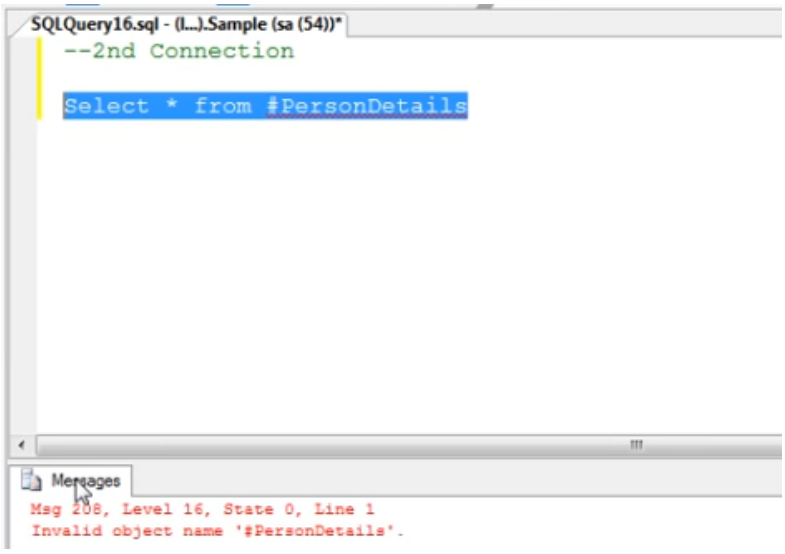
You can also check the existence of temporary tables using object explorer.



How do we know that it is a Temporary table, because it has # symbol in its name.



Temporary Tables are only available in the connection that they are created in. The TABLE AUTOMATICALLY gets DROPPED when the connection that created it is closed. 2 different windows have two different connections.



Local Temporary Tables

If the temporary table, is created inside the stored procedure, it get's dropped automatically upon the completion of 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.

GLOBAL Temporary tables can be selected or accessed from any connection unlike the LOCAL temporary tables

Global Temporary Tables

To create a Global Temporary Table, prefix the name of the table with 2 pound (##) 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

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

- What are indexes
- Why do we use indexes
- Advantages of indexes

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries. In a way we can compare them with indexes in a book, which help us in finding a chapter or topic.

An **index** contains keys built from one or more columns in the table or view. These keys are stored in a structure (B-tree) that enables **SQL** Server to find the row or rows associated with the key values quickly and efficiently.

TO improve performance in querying data from the table based on a where clause, we create index on the respective columns in the table that we will run the query on otherwise the query engine will have to perform an entire table scan where it will have to scan through each and every row which can be very slow performance wise.

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, and I ask you to locate a specific chapter in the 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

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

```
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.

Creating an Index

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

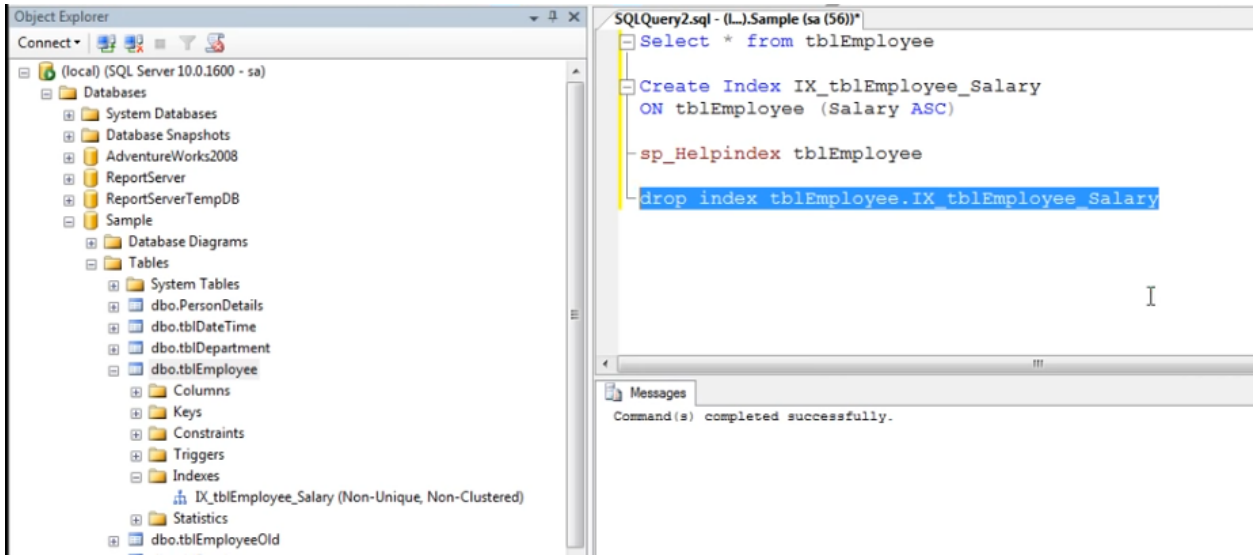
The index stores salary of each employee, in the ascending order as shown below. The actual index may look slightly different.

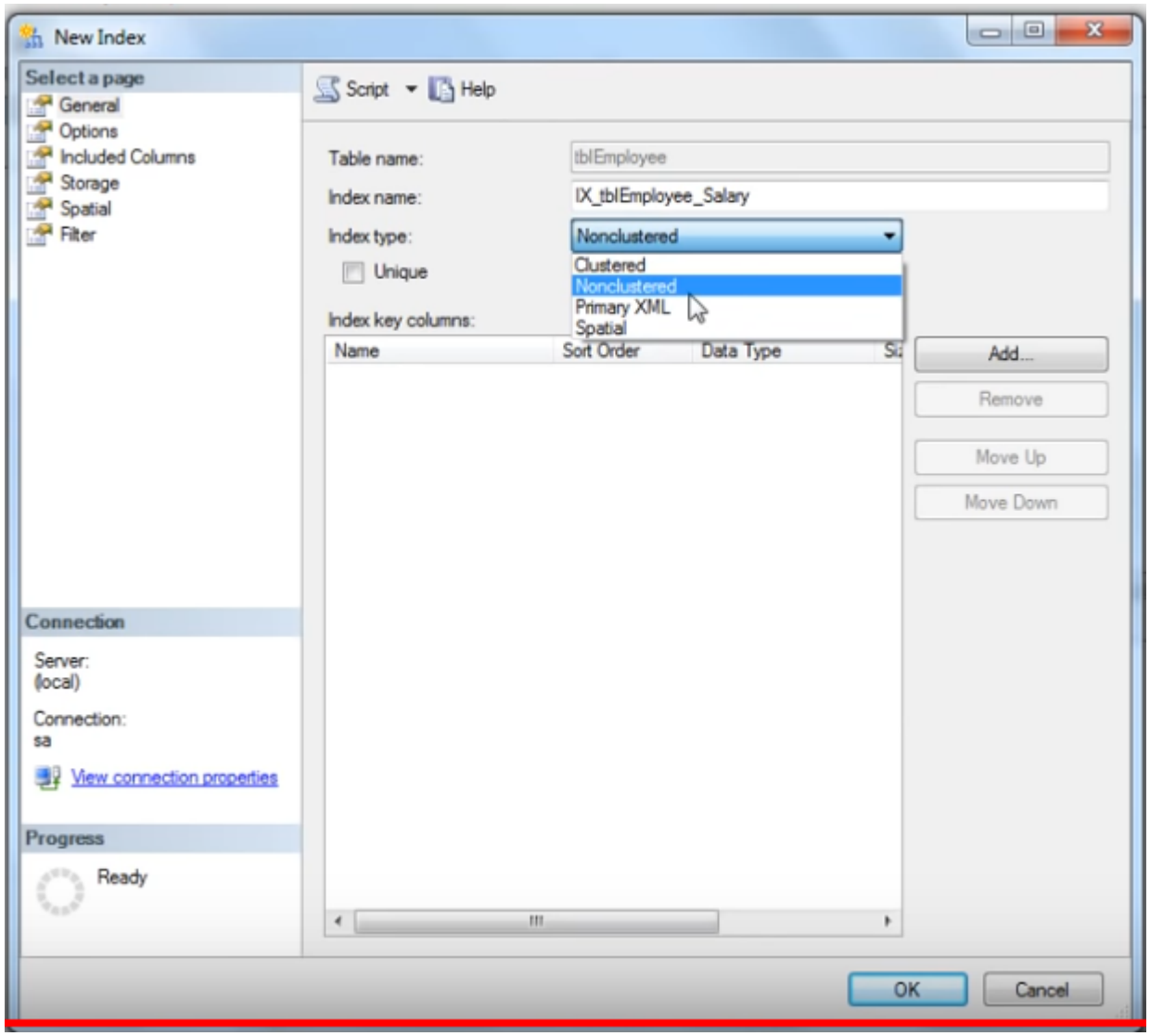
Id	Name	Salary	Gender
1	Sam	2500	Male
2	Pam	6500	Female
3	John	4500	Male
4	Sara	5500	Female
5	Todd	3100	Male

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.

Next Video – Different Types of indexes, Disadvantage etc...


Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
Database Snapshots
AdventureWorks2008
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
System Tables
dbo.PersonDetails
dbo.tblDateTime
dbo.tblDepartment
dbo.tblEmployee
Columns
Keys
Constraints
Triggers
Indexes
IX_tblEmployee_Salary (Non-Unique, Non-Clustered)
Statistics
dbo.tblEmployeeOld
SQLQuery2.sql - (l...).Sample (sa (56))*
Select * from tblEmployee
Create Index IX_tblEmployee_Salary
ON tblEmployee (Salary ASC)
sp_Helpindex tblEmployee
drop index tblEmployee.IX_tblEmployee_Salary
Messages
Command(s) completed successfully.


New Index
Select a page
General
Options
Included Columns
Storage
Spatial
Filter
Script
Help
Table name:
tblEmployee
Index name:
IX_tblEmployee_Salary
Index type:
Nonclustered
Clustered
Nonclustered
Primary XML
Spatial
Unique
Index key columns:
Name
Sort Order
Data Type
Add...
Remove
Move Up
Move Down
Connection
Server:
(local)
Connection:
sa
View connection properties
Progress
Ready
OK
Cancel

Index Types

Index Types
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 session: Clustered and Nonclustered

Clustered and nonclustered 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 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.

To confirm: `Execute sp_helpindex tblEmployee`

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')
```

```
Select * from tblEmployee
```

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

How to find if a Clustered Index exists? Use System stored Procedure as below:

SQLQuery6.sql - (l...).Sample (sa (54))* | SQLQuery5.sql - (l...).Sample (sa (52))*

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

Execute sp_helpindex tblEmployee

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')
```

Results | Messages

	index_name	index_description	index_keys
1	PK__tblEmplo__3214EC071AD3FDA4	clustered, unique, primary key located on PRIMARY	Id

The above index on id column was not created by us explicitly, it got created automatically as a primary key constraint creates a clustered index if the table does not have any other clustered index.

Clustered Index

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.

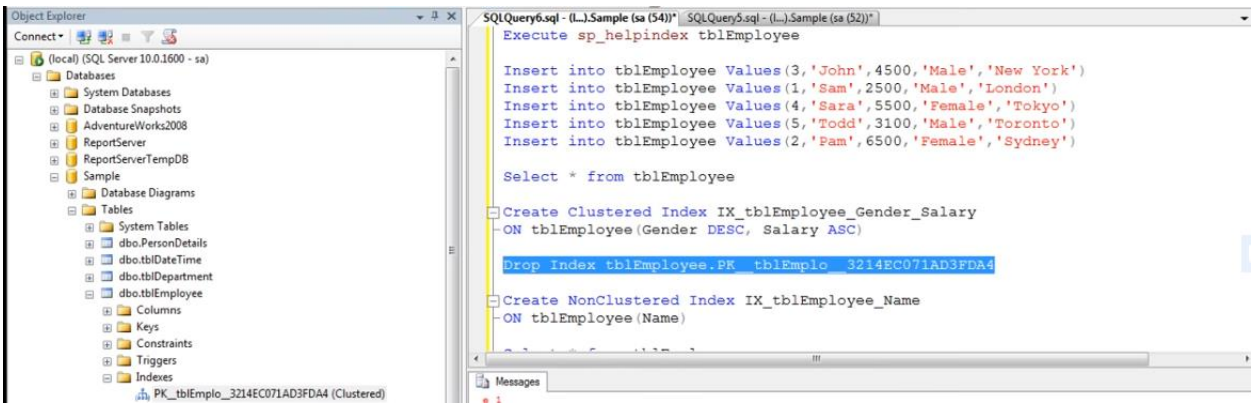
Create a composite clustered Index on the Gender and Salary columns

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

```
Select * from tblEmployee
```

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

Dropping an Index



Note: It is not possible to have more than one Clustered Index in a table but it is possible to have more than one column in an index.

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

In this index there will be two columns as index keys, gender and salary. First gender will be arranged in descending order and then within gender the salary will be arranged in ascending order.

NonClustered Index

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

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	Femal	Sydney

Name	Row Address
John	Row Address
Pam	Row Address
Sam	Row Address
Sara	Row Address
Todd	Row Address

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.

Difference

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 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 separately from the table, additional storage space is required.

Unique and Non Unique Indexes

A unique index ensures that the values in the index **key** columns are unique. A unique constraint also guarantees that no duplicate values can be inserted into the column(s) on which the constraint is created. When a unique constraint is created a corresponding unique index is automatically created on the column(s).

- What is a Unique Index
- Difference between unique index and unique constraint

Unique Index

Unique index is used to enforce uniqueness of key values in the index.

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

Note: By default, PRIMARY KEY constraint, creates a unique clustered index.

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

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

Difference between Unique Constraint and Unique Index

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.

A unique clustered index gets created automatically when a primary key is created.

SQLQuery3.sql - (l...).Sample (sa (55))* SQLQuery1.sql - (l...).Sample (sa (52))*

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

sp_helpindex tblEmployee

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

Results | Messages

	index_name	index_description	index_keys
1	PK__tblEmplo__3214EC0757DD0BE4	clustered, unique, primary key located on PRIMARY	Id

Adding Unique CONSTRAINT:

CREATE UNIQUE CLUSTERED INDEX ON BOTH COLUMNS FIRSTNAME AND LASTNAME TO ENSURE THAT BOTH WE ONLY GET UNIQUE FULL NAMES.

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

sp_helptext UQ_tblEmployee_City

ALTER TABLE tblEmployee
 ADD CONSTRAINT UQ_tblEmployee_City
UNIQUE (City)

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

UNIQUE CONSTRAINT BY DEFAULT CREATES A NON-CLUSTERED DEFAULT INDEX. CAN CHECK BOTH KEYS AND INDEX FOLDERS IN OBJECT EXPLORER TO CHECK WHAT TYPE OF KEY AND INDEX IS GENERATED.

Unique Index

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.

Useful points to remember:

1. By default, a PRIMARY KEY constraint, creates a unique clustered index, where as a UNIQUE constraint creates a unique Non-Clustered 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.

Advantages and disadvantages of indexes

Advantages of Indexes

In Part 35, we have learnt that, Indexes are used by queries to find data quickly.

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

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

```
--SELECT statement with a WHERE clause
Select * from tblEmployee where Salary > 4000 and Salary < 8000

--DELETE and UPDATE statement
Delete from tblEmployee where Salary = 2500
Update tblEmployee Set Salary = 9000 where Salary = 7500

--ORDER BY ASCENDING
Select * from tblEmployee order by Salary

--ORDER BY DESCENDING
Select * from tblEmployee order by Salary Desc

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

Disadvantages 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.

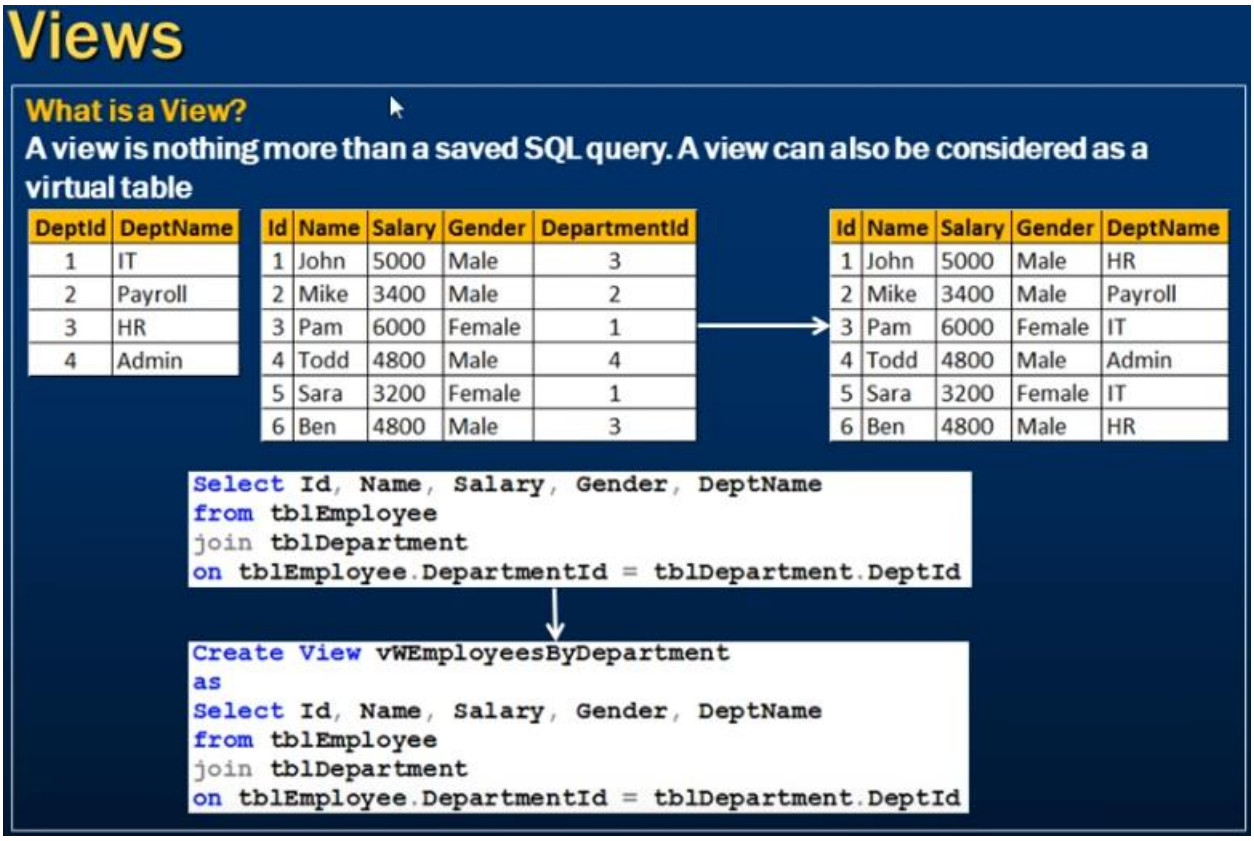
Insert Update and Delete statements can become slow: When DML (Data Manipulation Language) statements (INSERT, UPDATE, DELETE) modifies data in a table, the data in all the indexes also needs to be updated. Indexes can help, to search and locate the rows, that we want to delete, but too many indexes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

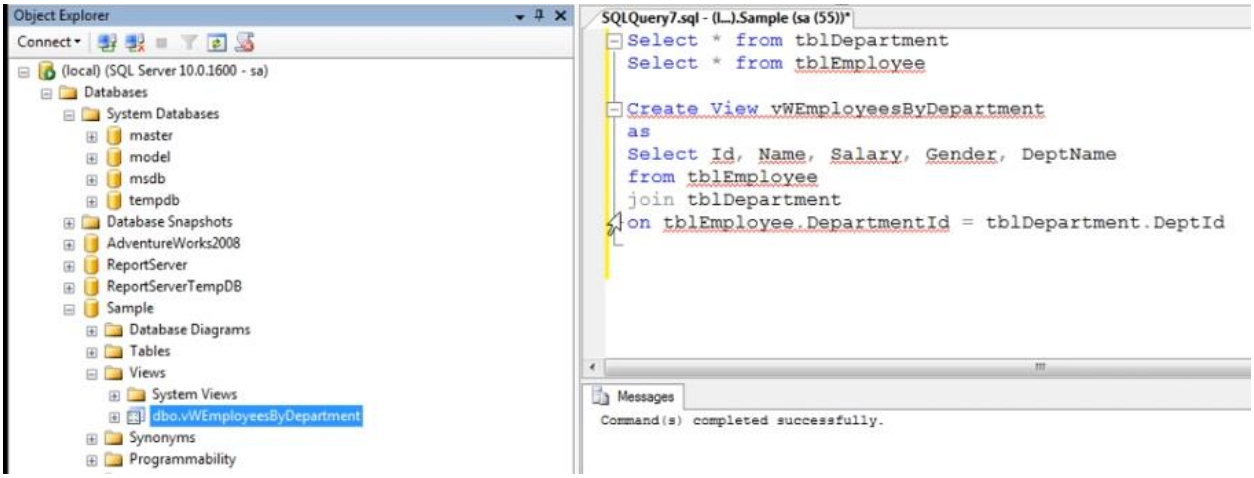
What is a covering query - If all the columns that you have requested in the SELECT clause of query, are present in the index, then there is no need to lookup in the table again. The requested columns data can simply be returned from the index.

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 non-clustered indexes can be composite indexes. To a certain extent, a composite index, can cover a query.

Views

In a database, a view is the result set of a stored query on the data, which the database users can query just as they would in a persistent database collection object. This pre-established query command is kept in the database dictionary.





The view created above can be treated just like a table.

```
Select * from vWEmployeesByDepartment
```

Results | Messages

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

Advantages of views

Views can be used to reduce the complexity of the database schema

Views can be used as a mechanism to implement row and column level security.

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

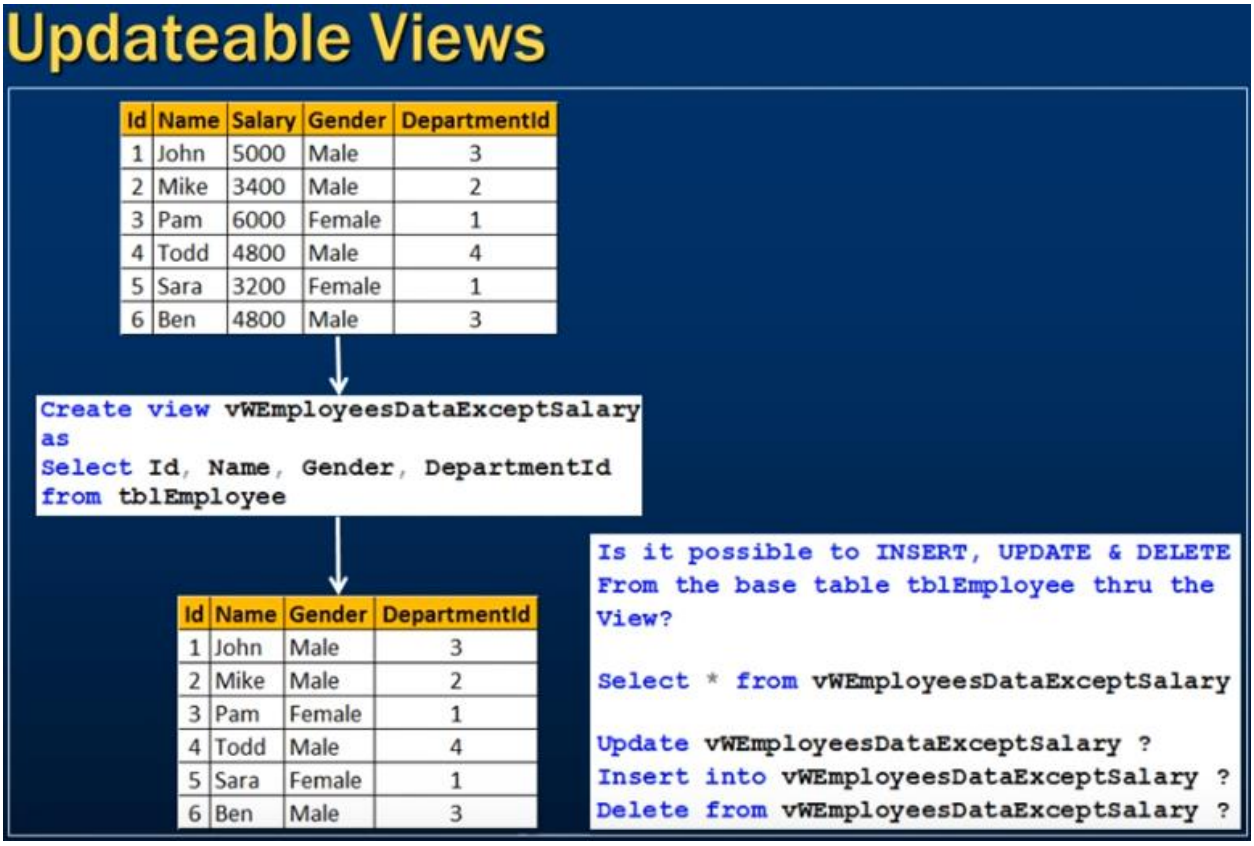
To modify a view - ALTER VIEW statement
To Drop a view - DROP VIEW vWName

```
Create View vwEmployeeByITDept
as
Select h.Id, h.Location, h.Department_Head, d.Name, d.Gender, d.DepartmentName
from Department_Head h
join Department_Name d
on h.Id = d.Id
where d.DepartmentName = 'IT'

select * from vwEmployeeByITDept

Create View vwEmployeeByCountofDept
as
Select d.DepartmentName, Count(d.Id) as Total_Employees
from Department_Head h
join Department_Name d
on h.Id = d.Id
group by DepartmentName

select * from vwEmployeeByCountofDept
```

Updatable views

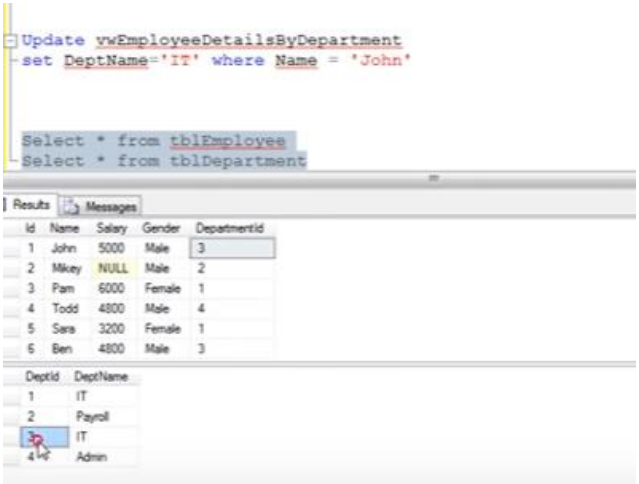
Some views are updatable and references to them can be used to specify tables to be updated in data change statements. That is, you can use them in statements such as UPDATE, DELETE, or INSERT to update the contents of the underlying table.

Thus it is possible to update the base table using views. In SQL Server vies are updatable.

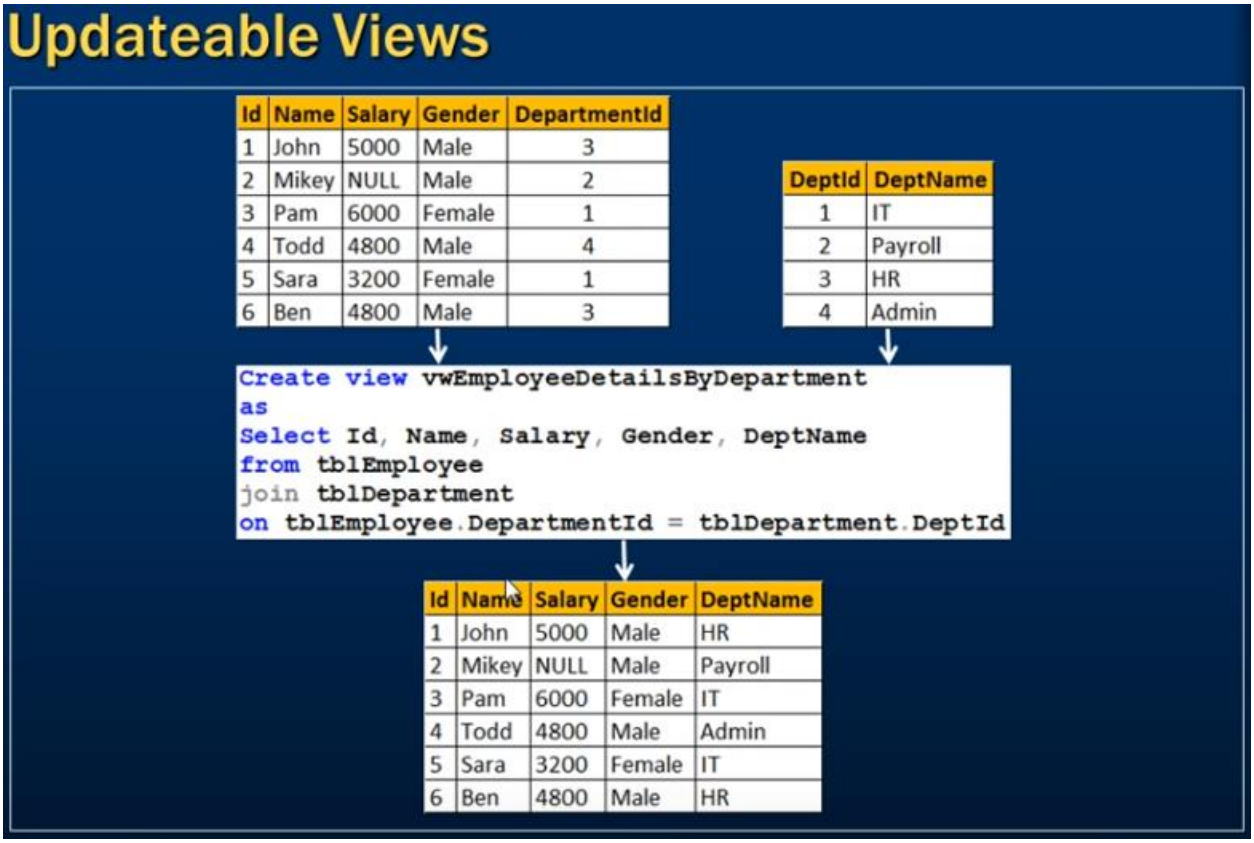


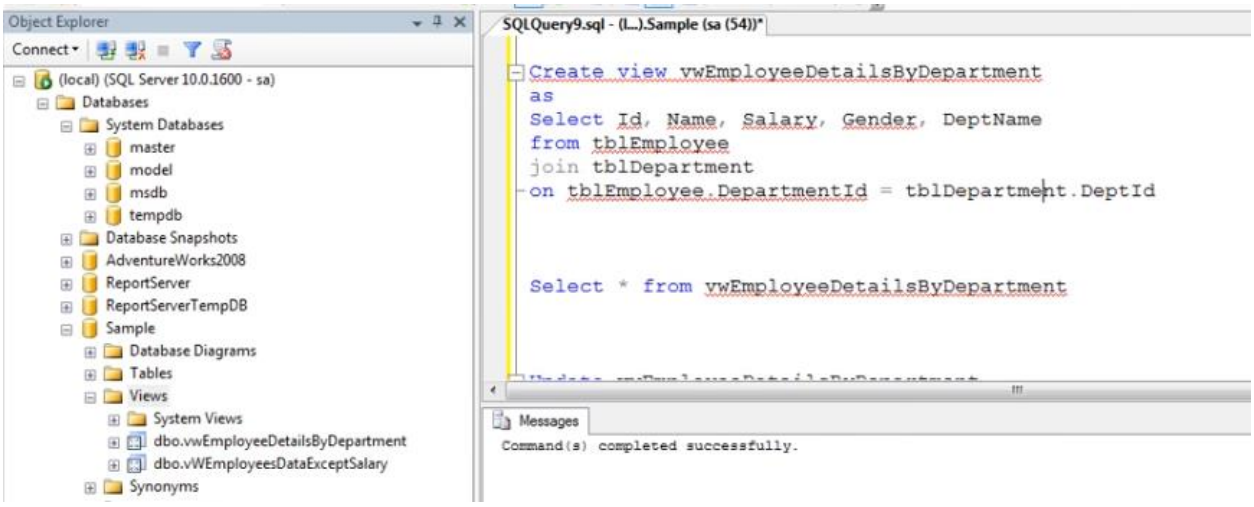
CAN ALSO UPDATE A VIEW

```
UPDATE vwEmployeeByITDept
Set Name = 'Priya' where Name = 'Simon'
```

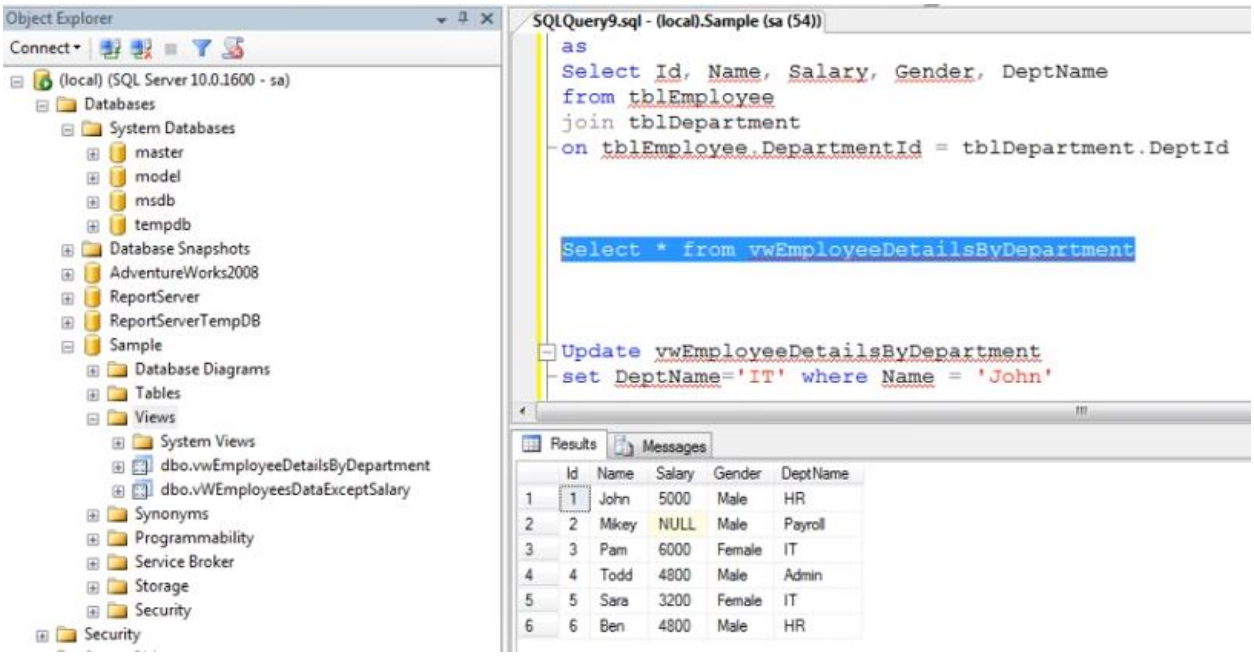


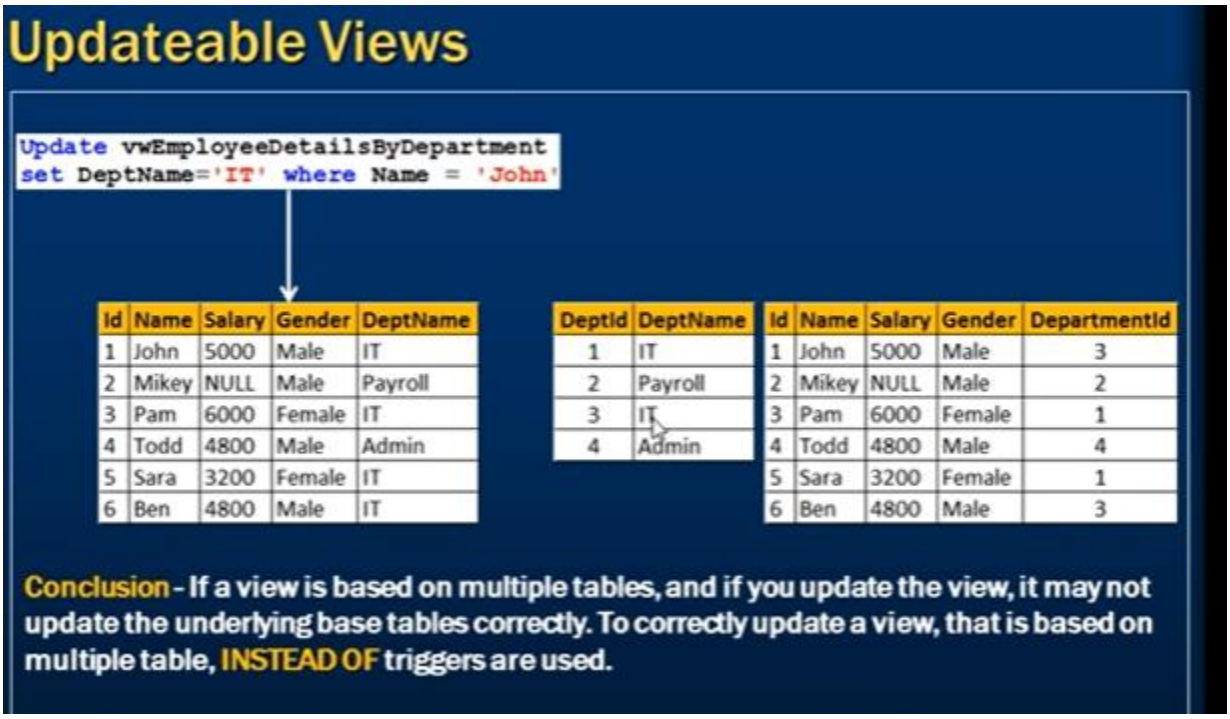
If you have a view that is based on multiple tables when you update that view then what happens?





Let us select data from that view below and we get data as expected.


Object Explorer
Connect
(local) (SQL Server 10.0.1600 - sa)
Databases
System Databases
master
model
msdb
tempdb
Database Snapshots
AdventureWorks2008
ReportServer
ReportServerTempDB
Sample
Database Diagrams
Tables
Views
System Views
dbo.vwEmployeeDetailsByDepartment
dbo.vWEmployeesDataExceptSalary
Synonyms
Programmability
Service Broker
Storage
Security
Security
SQLQuery9.sql - (local).Sample (sa (54))
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
Select * from vwEmployeeDetailsByDepartment
Update vwEmployeeDetailsByDepartment
set DeptName='IT' where Name = 'John'
Results
Messages
Id Name Salary Gender DeptName
1 1 John 5000 Male HR
2 2 Mikey NULL Male Payroll
3 3 Pam 6000 Female IT
4 4 Todd 4800 Male Admin
5 5 Sara 3200 Female IT
6 6 Ben 4800 Male HR


Updateable Views
Update vwEmployeeDetailsByDepartment
set DeptName='IT' where Name = 'John'
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
DeptId DeptName
1 IT
2 Payroll
3 IT
4 Admin
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
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.

Indexed views

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.

Indexed Views - Example

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

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

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

```
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
```

Guidelines for creating Indexed Views

```
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
```

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

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.
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.

```
Create Unique Clustered Index UIX_vWTotalSalesByProduct_Name
on vWTotalSalesByProduct(Name)
```

For the complete list, please visit MSDN
http://msdn.microsoft.com/en-us/library/ms191432(v=sql.105).aspx

When a view is created with Schemabinding option, then it does not user to make changes in the base table or delete any data from base table that can affect the definition of the view.

COUNT_BIG(*) function is used when the view has index in it.

It is important to index the view to improve performance in select query of the view statement where it has to scan through entire rows when trying to group by the column values.

Indexes are helpful when the data in the tables are not changed frequently such as in large data warehouses. However, in cases where data is frequently changed, creating indexes can cause higher cost in the performance due to the need for increased maintenance in updating the indexed on every change of data.

```
Create Unique Clustered Index UIX_vWTotalSalesByProduct_Name
on vWTotalSalesByProduct(Name)
```

IN SQL SERVER IT IS NOT POSSIBLE TO PASS PARAMETERS TO VIEW:

INSTEAD USE TABLE-VALUED FUNCTIONS.

View limitations

View Limitations

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

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.

4. Views cannot be based on temporary tables.

View Limitations

```
Create View vWEmployeeDetails
as
Select Id, Name, Gender, DepartmentId
from   tblEmployee
```

```
Create View vWEmployeeDetails
@Gender nvarchar(20)
as
Select Id, Name, Gender, DepartmentId
from   tblEmployee
Where  Gender = @Gender
```

```
--You can filter Gender in the WHERE clause
Select * from vWEmployeeDetails where Gender = 'Male'
```

```
-- Inline Table valued function 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)


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

Cannot create views on Temp 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')


Create View vwOnTempTable
as
Select Id, Name, Gender
from ##TestTempTable
```

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

DML triggers

A **trigger** is a special type of stored procedure that automatically runs when an event occurs in the database server. DML **triggers** run when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view.

Triggers

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

DML triggers are fired automatically in response to DML events (INSERT, UPDATE & DELETE)

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

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

INSTEAD of triggers, fires instead of the triggering action. The INSERT, UPDATE, and DELETE statements, causes an INSTEAD OF trigger to fire INSTEAD OF the respective statement execution.

After Triggers

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

Id	AuditData
1	New employee with Id = 8 is added at Sep 17 2012 8:04PM
2	An existing employee with Id = 8 is deleted at Sep 17 2012 8:29PM
3	An existing employee with Id = 1 is deleted at Sep 17 2012 8:31PM
4	An existing employee with Id = 3 is deleted at Sep 17 2012 8:31PM
5	An existing employee with Id = 4 is deleted at Sep 17 2012 8:31PM

```
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
```

```
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
```

```
Insert into tblEmployee values
(8,'Ben', 4800, 'Male', 3)
```

```
Delete from tblEmployee where Id = 3
```

SQLQuery18.sql - (L...).Sample (sa (53))* | SQLQuery15.sql - not connected* | SQLQuery14.sql - not connected*

```
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 ' + cas
END
```

Object Explorer

- System Databases
- Database Snapshots
- AdventureWorks2008
- ReportServer
- ReportServerTempDB
- Sample
  - Database Diagrams
  - Tables
    - System Tables
    - dbo.PersonDetails
    - dbo.tblDateTime
    - dbo.tblDepartment
    - dbo.tblEmployee
      - Columns
      - Keys
      - Constraints
      - Triggers
        - tr_tblEMployee_ForInsert
      - Indexes
      - Statistics
    - dbo.tblEmployeeAudit
    - dbo.tblEmployeeOld
    - dbo.tblEmployees
    - dbo.tblGender
    - dbo.tblIndiaCustomers
    - dbo.tblPerson
    - dbo.tblProduct
    - dbo.tblProductSales
    - dbo.tblRegistrations

SQLQuery18.sql - (L...).Sample (sa (53))* | SQLQuery15.sql - not connected* | SQLQuery14.sql - not connected*

```
Select * from inserted

ALTER 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 ' + cas
END

Insert into tblEmployee values (9,'Jimmy', 1800, 'Female', 3)

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
END
```

Subscribe

SQLQuery8.sql - LA...B9U5G9\Gauri (61))* | SQLQuery5.sql - not connected

```
Create Trigger tr_tblEMployee_ForInsert
ON tbl_persontbl
FOR INSERT
AS
BEGIN
    select * from inserted

END

select * from tbl_persontbl

Insert into tbl_persontbl Values(13, 'Priya', 'p@ram.com', 2, 32, 'Toronto', 55000, 3)
```

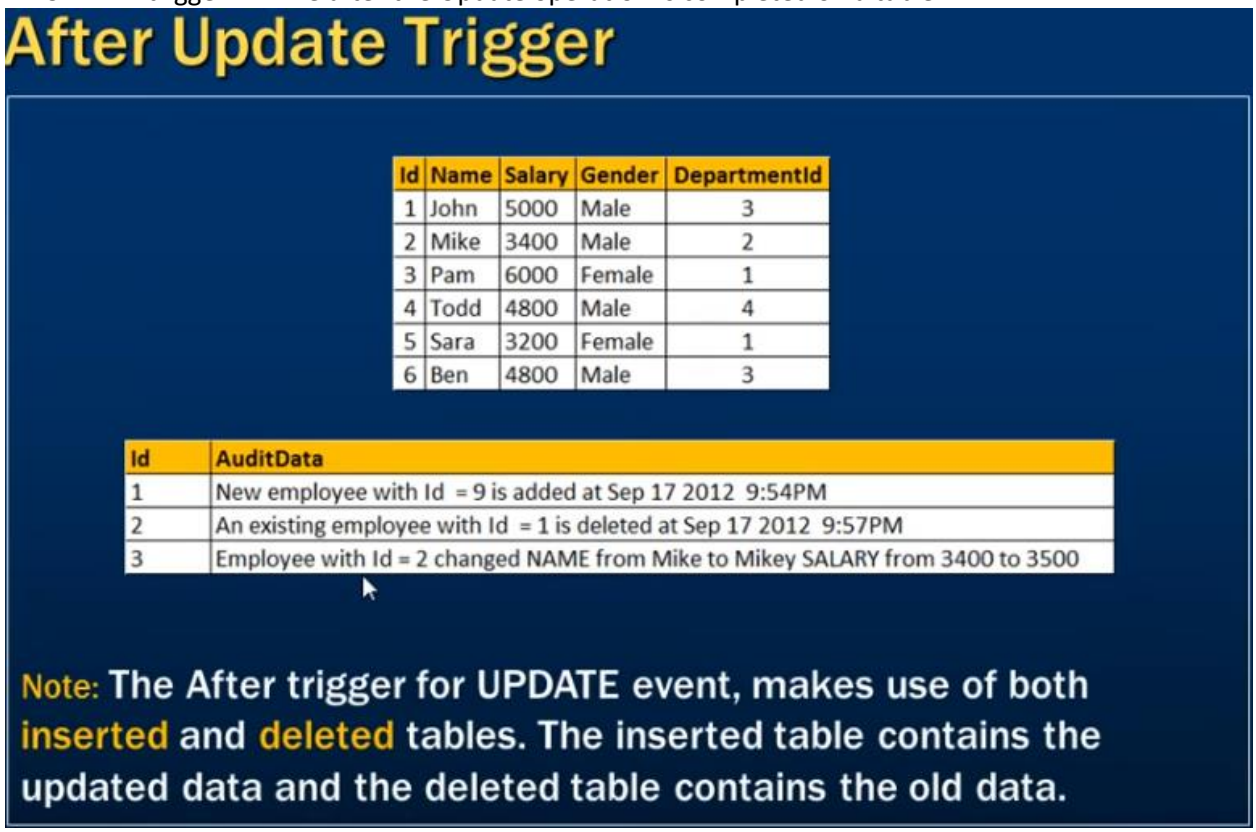
SQLQuery8.sql - LA...B9U5G9\Gauri (61))* | SQLQuery5.sql - not connected

```
Alter Trigger tr_tblEMployee_ForInsert
ON tbl_persontbl
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

Insert into tbl_persontbl Values(14, 'John', 'p@ram.com', 2, 32, 'Toronto', 55000, 3)

Create Table tblEmployeeAudit (
        Id int identity NOT NULL Primary key,
        Audit_Data nvarchar(max))
```

```
select * from tblEmployeeAudit
```

After update trigger

SQL AFTER UPDATE trigger will fire after the Update operation is completed on a table.



In the code below the trigger gets fired after Update execution completes.

SQLQuery1.sql - (l...).Sample (sa (52))*

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


Update tblEmployee set Name = 'James', Salary = 2000, Gender = 'Male'
where Id = 8
```

Results | Messages

	Id	Name	Salary	Gender	DepartmentId
1	8	Jane	1800	Female	3

	Id	Name	Salary	Gender	DepartmentId
1	8	James	2000	Male	3

```
Alter trigger tr_tblEmployee_ForUpdate
on tblEmployee
for Update
as
Begin
      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

      Declare @AuditString nvarchar(1000)

      Select *
      into #TempTable
      from inserted

      While(Exists(Select Id from #TempTable))
      Begin
            Set @AuditString = ''

            Select Top 1 @Id = Id, @NewName = Name,
            @NewGender = Gender, @NewSalary = Salary,
            @NewDeptId = DepartmentId
            from #TempTable
```

```
            Select @OldName = Name, @OldGender = Gender,
            @OldSalary = Salary, @OldDeptId = DepartmentId
            from deleted where Id = @Id

            Set @AuditString = 'Employee with Id = ' + Cast(@Id as nvarchar(4)) + ' chan
            if(@OldName <> @NewName)
                  Set @AuditString = @AuditString + ' NAME from ' + @OldName + ' to ' +

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

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

            if(@OldDeptId <> @NewDeptId)
                  Set @AuditString = @AuditString + ' DepartmentId from ' + Cast(@OldDep

            insert into tblEmployeeAudit values(@AuditString)

            Delete from #TempTable where Id = @Id
      End
End
```

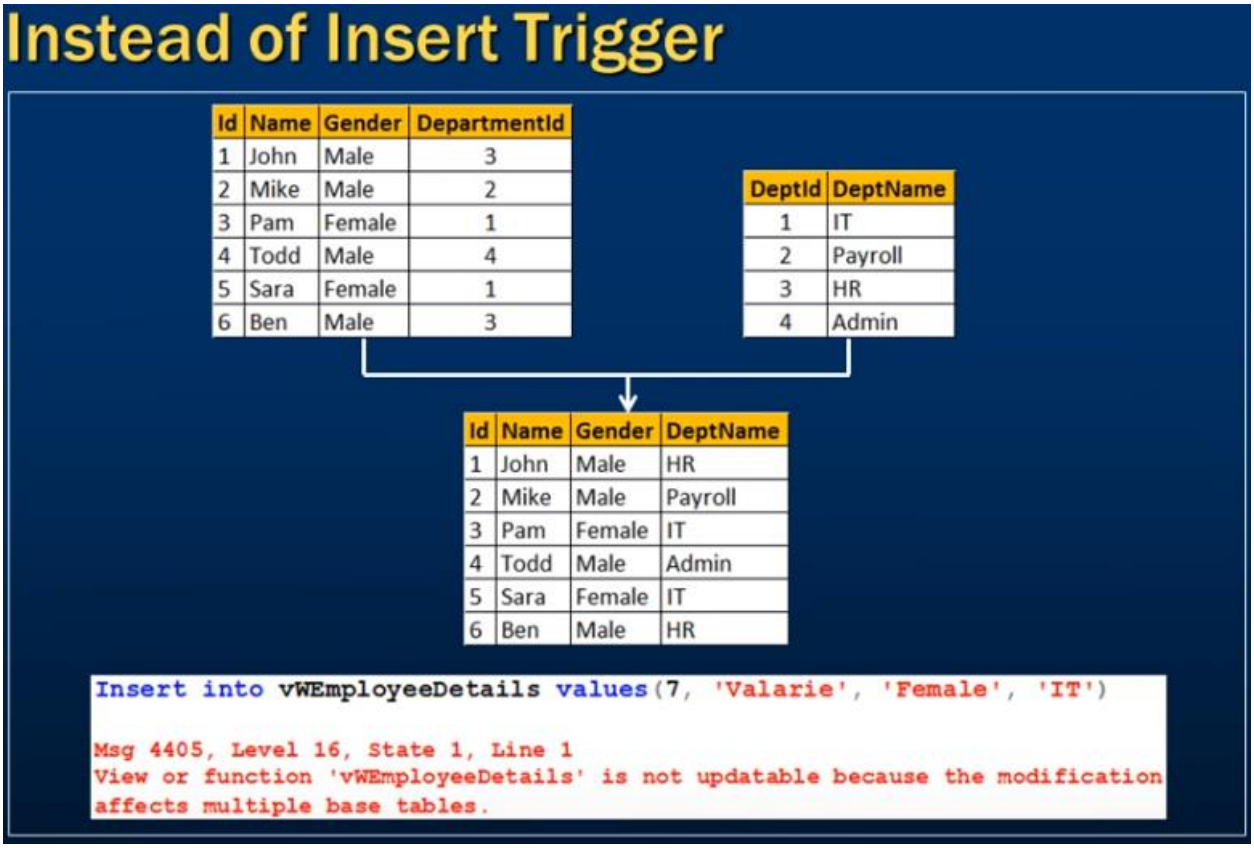
```
Update tblEmployee set Name = 'Tamed', Salary = 2000, Gender = 'Male'
where Id IN(1,2,3,4) 8
```

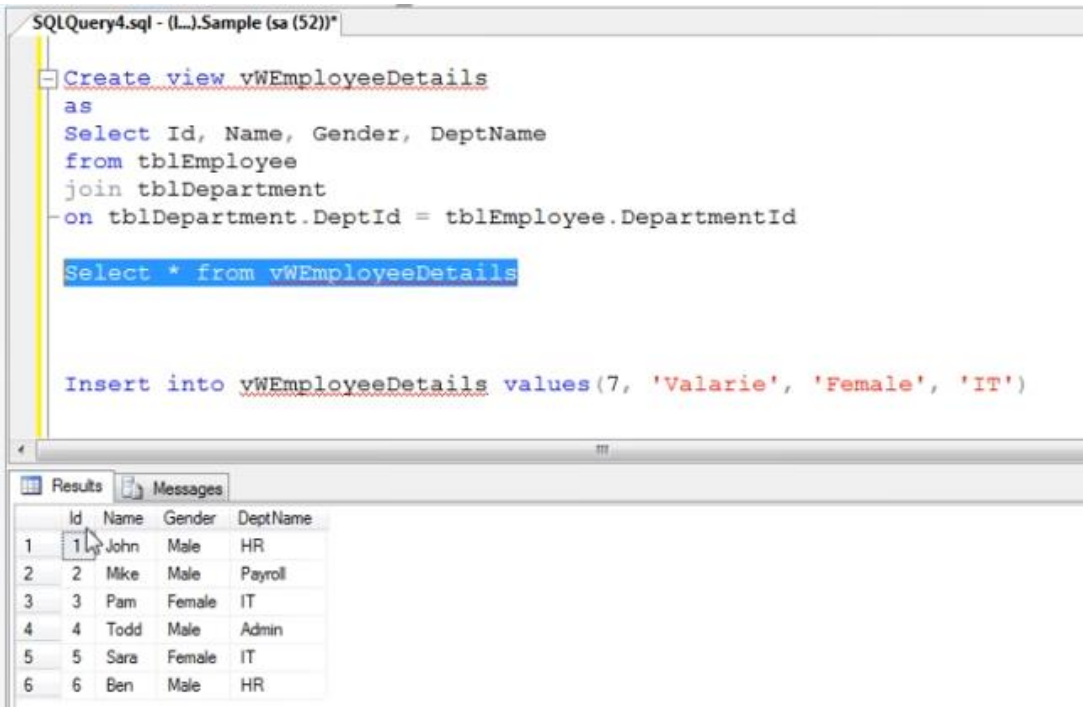
```
Update tblEmployee set Name = 'Tods', Salary = 2000, Gender = 'Female'
where Id = 4
```

4	4	Employee with Id = 4 changed NAME from Todd to Tods GENDER from Male to Female SALARY from 4800 to 2000

Instead of insert trigger

The SQL Server **Instead Of INSERT Trigger** will fire before the Insert operation on a table starts. These are usually used to update views correctly that are based on multiple tables.





```sql
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
```

SQLQuery4.sql - (L...).Sample (sa (52))*

```sql
Select * from tblEmployee
Select * from tblDepartment
```

Results | Messages

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

	DeptId	DeptName
1	1	IT
2	2	Payroll
3	3	HR
4	4	Admin

```sql
Insert into vWEmployeeDetails values(7, 'Valarie', 'Female', 'IT')
Select * from tblEmployee
Select * from tblDepartment
```

Results | Messages

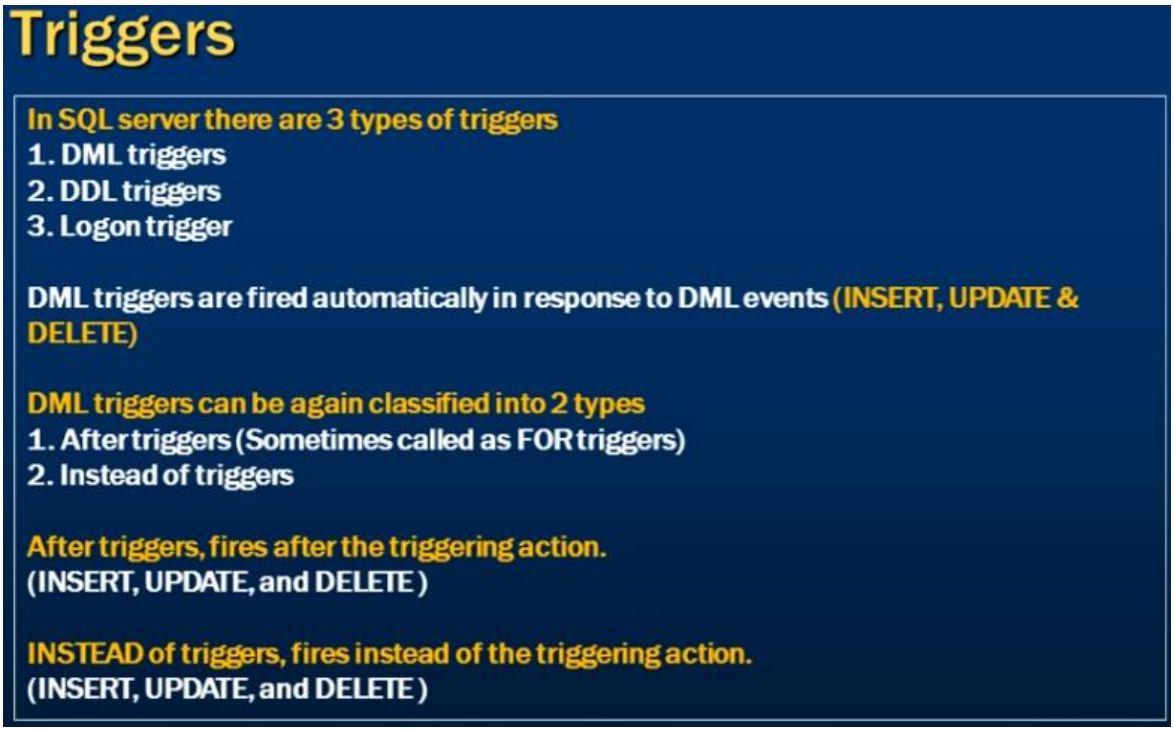
	Id	Name	Gender	DepartmentId
1	1	John	Male	3
2	2	Mike	Male	2
3	3	Pam	Female	1
4	4	Todd	Male	4
5	5	Sara	Female	1
6	6	Ben	Male	3
7	7	Valarie	Female	1

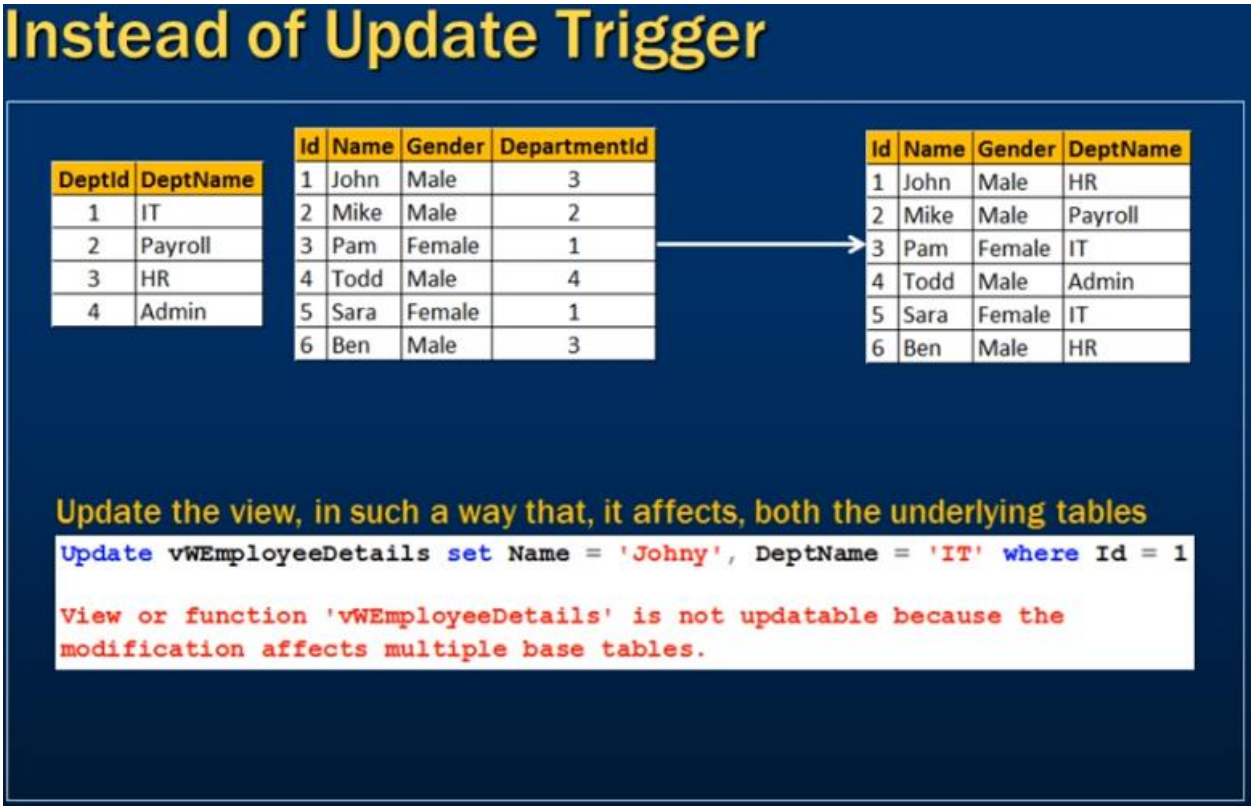
	DeptId	DeptName
1	1	IT
2	2	Payroll
3	3	HR

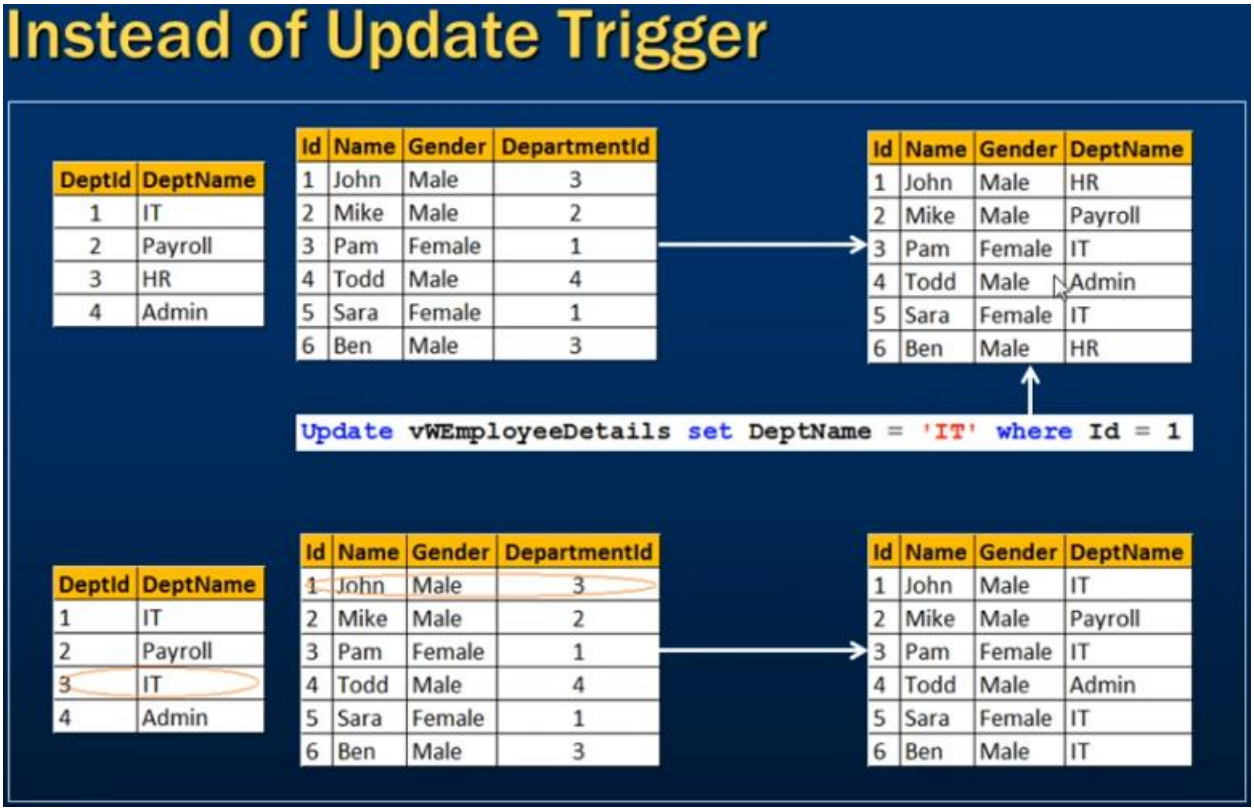
Row 7 got inserted into base table as result of instead of trigger.

Instead of update triggers

These **triggers** are executed **instead** of any of the Insert, **Update** or Delete operations. For **example** consider an **Instead** of **Trigger** for **Update** operation, whenever an **Update** is performed the **Trigger** will be executed first and if the **Trigger updates** record then only the record will be **updated**.







When updating the view that affects multiple tables, we will need to create an instead of trigger because it will not allow us to do so otherwise.

SQLQuery7.sql - (L...).Sample (sa (56))* | SQLQuery5.sql - not connected*

```
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
```

```
Select * from  vWEmployeeDetails


-- Update affecting multiple base tables
Update vWEmployeeDetails set Name = 'Johny', DeptName = 'IT' where Id = 1


-- Update affecting just one base tables
Update vWEmployeeDetails set DeptName = 'IT' where Id = 1


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

Results | Messages

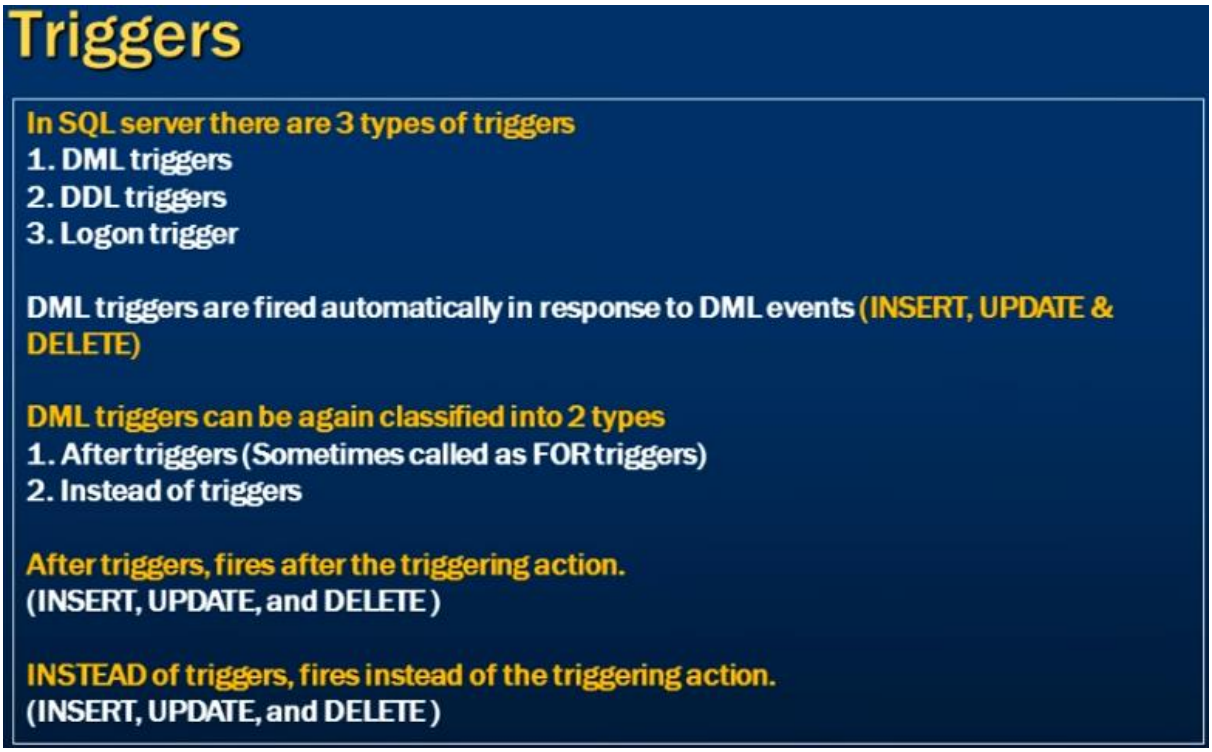
	Id	Name	Gender	DeptName
1	1	John	Male	HR
2	2	Mike	Male	Payroll
3	3	Pam	Female	IT
4	4	Todd	Male	Admin
5	5	Sara	Female	IT
6	6	Ben	Male	HR

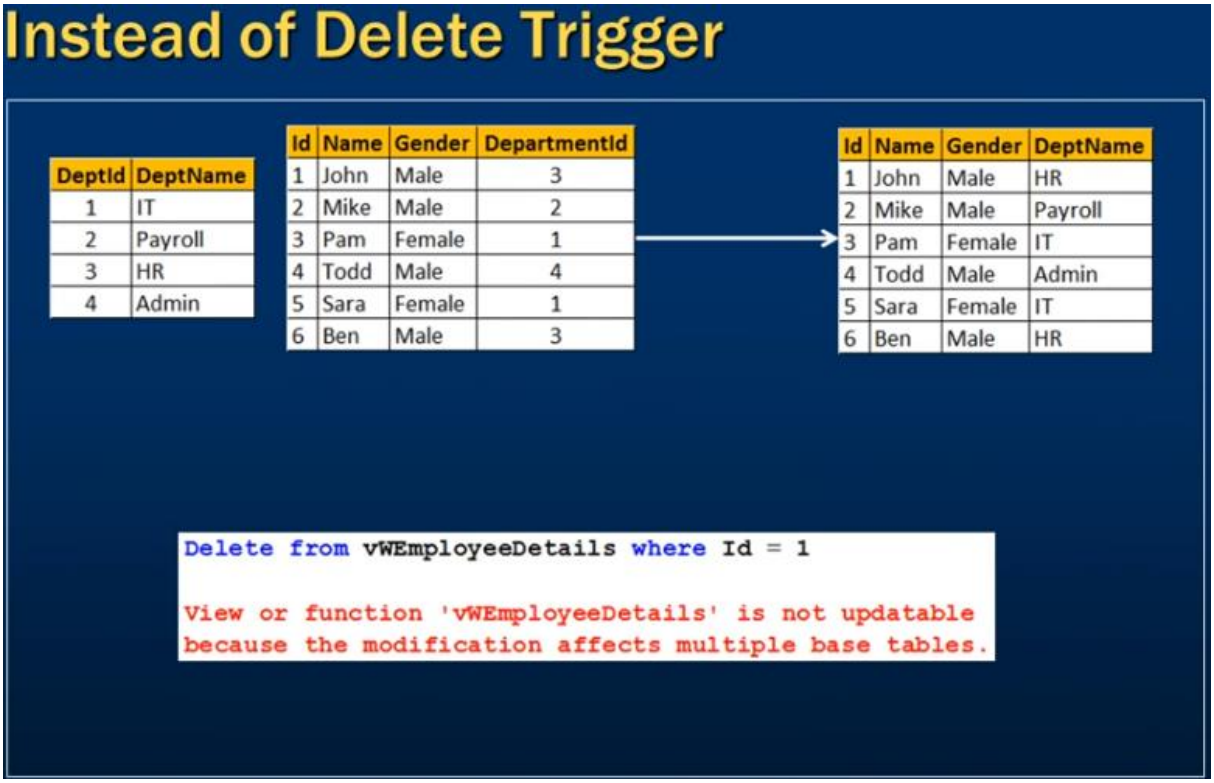
```
Update vWEmployeeDetails set DeptName = 'IT' where Id = 1

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

```
(1 row(s) affected)
```

Instead of delete triggers

Instead Of **triggers** are executed **instead** of any of the Insert, Update or **Delete** operations. For **example** consider an **Instead** of **Trigger** for **Delete** operation, whenever a **Delete** is performed the **Trigger** will be executed first and if the **Trigger** deletes record then only the record will be deleted.





SQLQuery8.sql - (L...).Sample (sa (53))*

```
Select * from tblEmployee
Select * from tblDepartment

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

Select * from  vWEmployeeDetails


Delete from vWEmployeeDetails where Id = 1
```

```
Select * from  vWEmployeeDetails
```

Results | Messages

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

```
Delete from vWEmployeeDetails where Id = 1
```

Messages

```
Msg 4405, Level 16, State 1, Line 1
View or function 'vWEmployeeDetails' is not updatable because the modification affects multiple base tables.
```

Let's create a Trigger on this view.

```
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
```

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
```

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)

Note: 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.

```
Select * from vWEmployeeDetails
```

Results | Messages

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

```
Delete from vWEmployeeDetails where Id IN (1,2)
```

```
(2 row(s) affected)

(2 row(s) affected)
```

```
Select * from vWEmployeeDetails
```

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

Subquery works in the same way as the main query. Therefore, it is commented out.

Now we will uncomment the subquery and comment the upper half just to prove that subquery works in the same way. Also change Create to Alter as query already exists.

```
Alter 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
```

```
Messages
Command(s) completed successfully.
```

Now let us try to delete records 3 and 4 and see if this works.

```
Delete from vWEmployeeDetails where Id IN (3,4)

(2 row(s) affected)

(2 row(s) affected)

Select * from  vWEmployeeDetails
```

Results | Messages

	Id	Name	Gender	DeptName
1	5	Sara	Female	IT
2	6	Ben	Male	HR

Id 3 and 4 have been deleted.

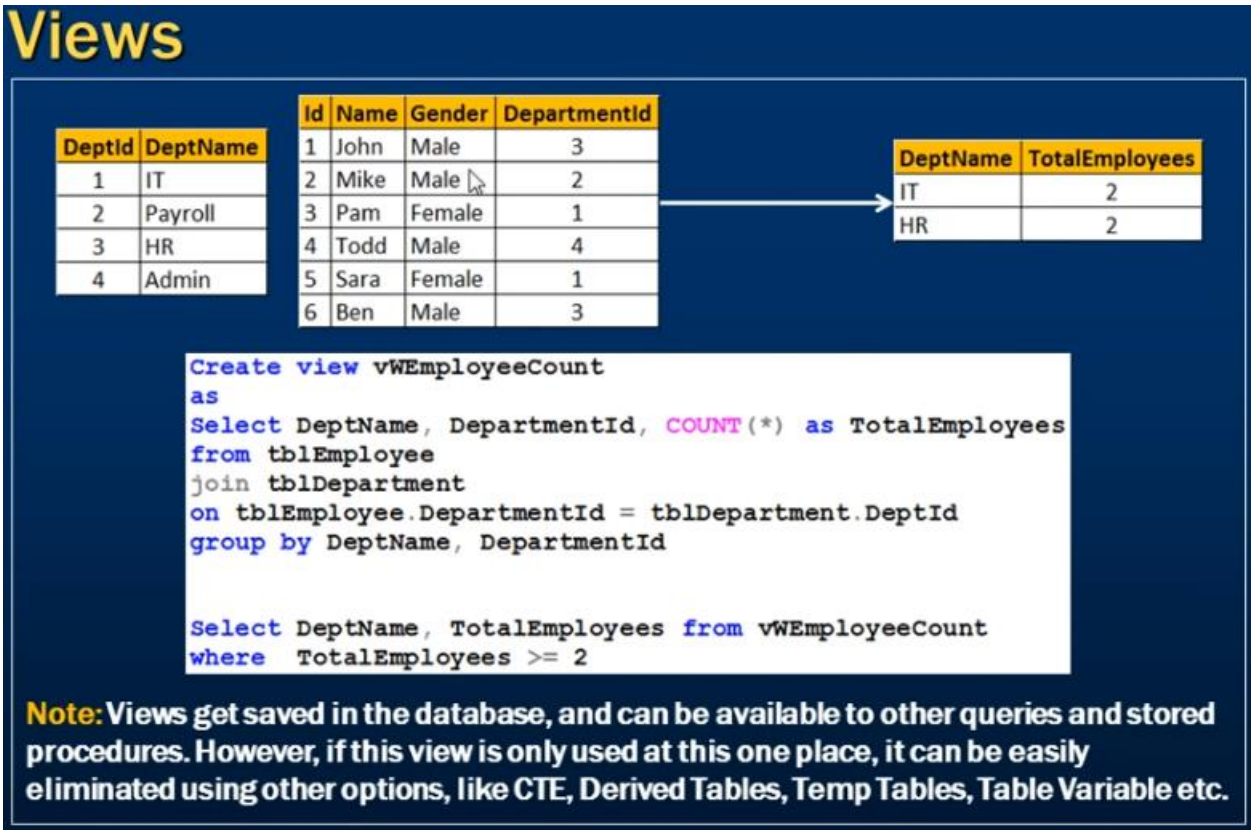
Derived tables and common table expressions

A **derived table** is a virtual **table** that is created within the scope of a query. The **table** is created with a SELECT statement of its own and is given an alias using the AS clause. It's contents can then be used within the query but become unavailable once the query has completed.

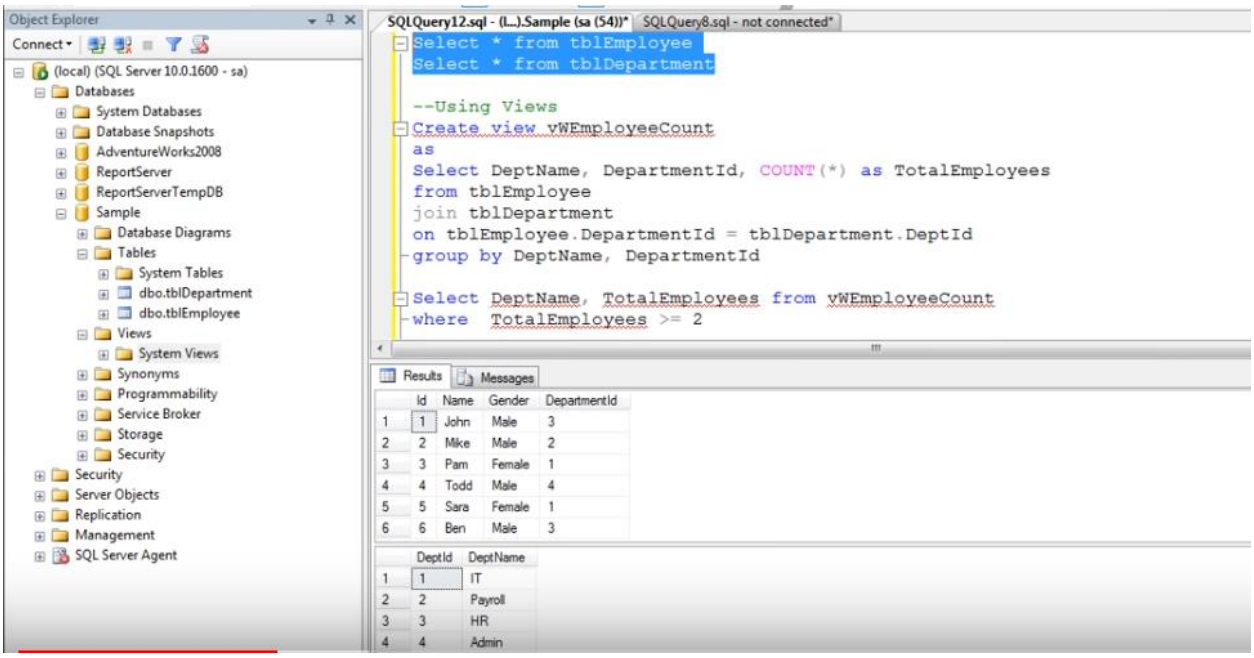
A **CTE** (**Common Table Expression**) is a temporary result set that you can reference within another SELECT, INSERT, UPDATE, or DELETE statement. They were introduced in **SQL** Server version 2005. They are **SQL**-compliant and part of the ANSI **SQL** 99 specification. A **CTE** always returns a result set.

- Derived tables and common table expressions
- Compare with – Views, Table Variable, Local and Global Temp Tables

Let's say we have two tables, we want to join these 2 tables and get result in the form of the 3rd table shown that lists department names and total employees in those departments but only those departments in which there are at least 2 employees. There are several ways to do it but we will do it below using Views.



Let us first list the data from both the tables.



Next we create the view

```
--Using Views
Create view vWEmployeeCount
as
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId
```

Execute the View

```
Messages
Command(s) completed successfully.

Select * from vWEmployeeCount
```

Results | Messages

	DeptName	DepartmentId	TotalEmployees
1	IT	1	2
2	Payroll	2	1
3	HR	3	2
4	Admin	4	1

But in our output we are expecting only Department Name and Total Employees.

```
Select DeptName, TotalEmployees from vWEmployeeCount
where TotalEmployees >= 2
```

Results | Messages

	DeptName	TotalEmployees
1	IT	2
2	HR	2

How can we achieve the same thing using Temporary Tables.

Temp Tables

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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

DeptName	TotalEmployees
IT	2
HR	2

```
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
```

Notes: 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 Temp Tables
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
into #TempEmployeeCount
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId
```

```
(4 row(s) affected)
```

We have the temporary table created. Let us view that:

```
Select * from #TempEmployeeCount
```

Results | Messages

	DeptName	DepartmentId	TotalEmployees
1	IT	1	2
2	Payroll	2	1
3	HR	3	2
4	Admin	4	1

So, if we want the same result as above then we can run query:

```
Select DeptName, TotalEmployees
From #TempEmployeeCount
where  TotalEmployees >= 2
```

Results | Messages

	DeptName	TotalEmployees
1	IT	2
2	HR	2

We can do the same thing using Table Variable. However, when we created temporary table we did not define the structure and it was inferred based on the select statement used. But when using table variable we have to define structure explicitly.

Table Variable

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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

DeptName	TotalEmployees
IT	2
HR	2

```
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.

The advantage of using Table Variable is that we don't have to drop it like Temporary Tables.

Also there are several other differences between temporary tables and table variables.

We can also achieve same result using Derived Tables.

Derived Tables

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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

DeptName	TotalEmployees
IT	2
HR	2

```
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
```

EmployeeCount is the derived table

Note: Derived tables are available only in the context of the current query.

```
--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
```

Results | Messages

	DeptName	TotalEmployees
1	IT	2
2	HR	2

CTE

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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

DeptName	TotalEmployees
IT	2
HR	2

```
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.

CTE

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.

```
--Syntax
WITH cte_name (Column1, Column2, ..)
AS
( CTE_query )
```

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

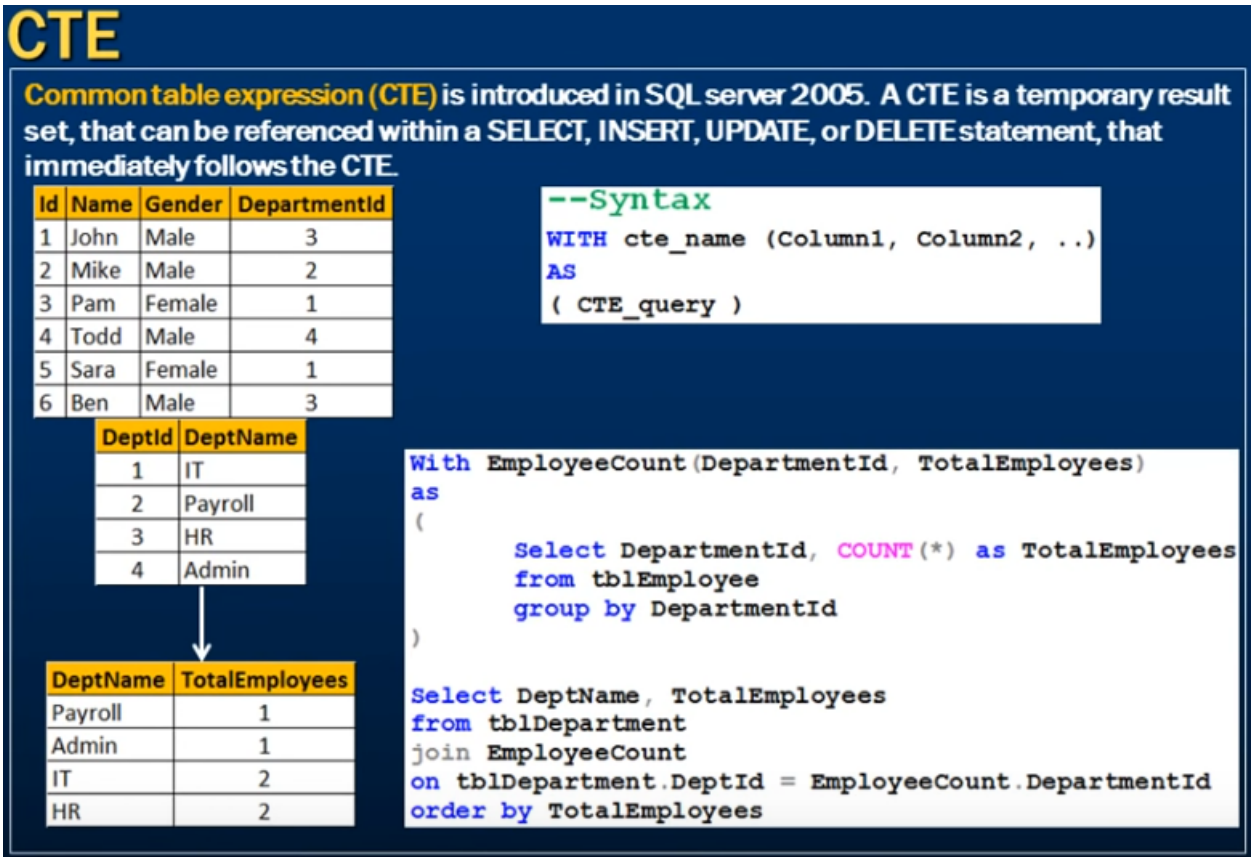
DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

DeptName	TotalEmployees
Payroll	1
Admin	1
IT	2
HR	2

```
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
```

CTE

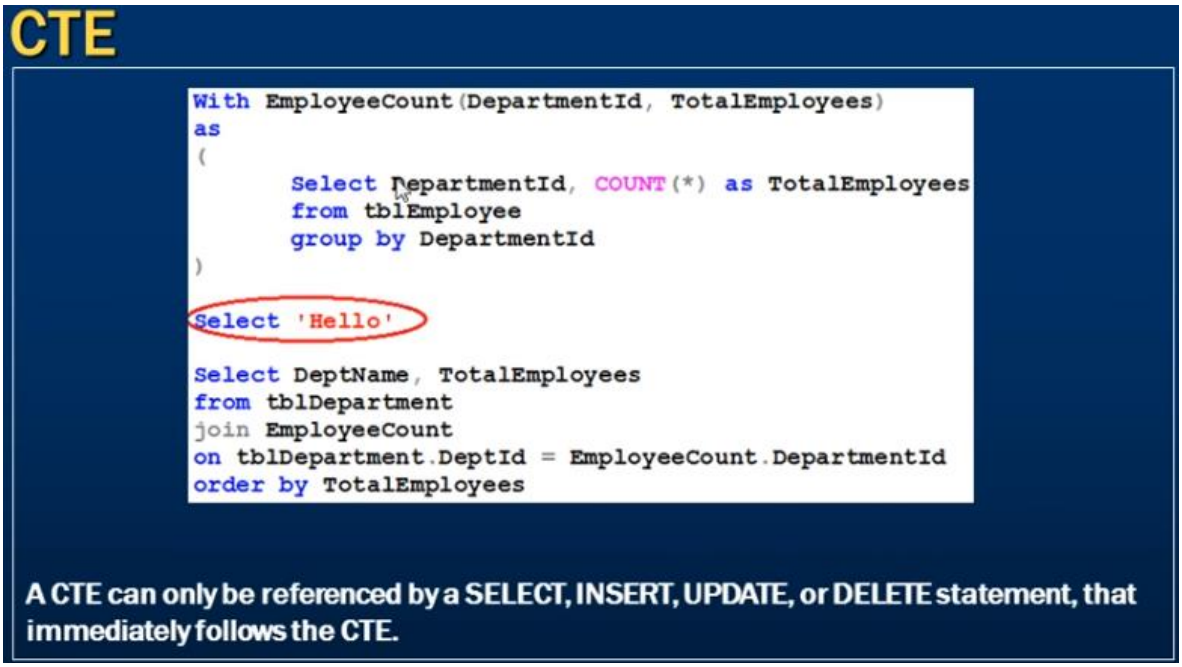
A **CTE** (**Common Table Expression**) is a temporary result set that you can reference within another SELECT, INSERT, UPDATE, or DELETE statement. They were introduced in **SQL** Server version 2005. They are **SQL**-compliant and part of the ANSI **SQL** 99 specification. A **CTE** always returns a result set.



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

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

DeptName	TotalEmployees
Payroll	1
Admin	1
IT	2
HR	2



CTE

```
--Creating 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
        where DeptName IN ('HR','Admin')
        group by DeptName
)
Select * from EmployeesCountBy_HR_Admin_Dept
UNION
Select * from EmployeesCountBy_Payroll_IT_Dept
```

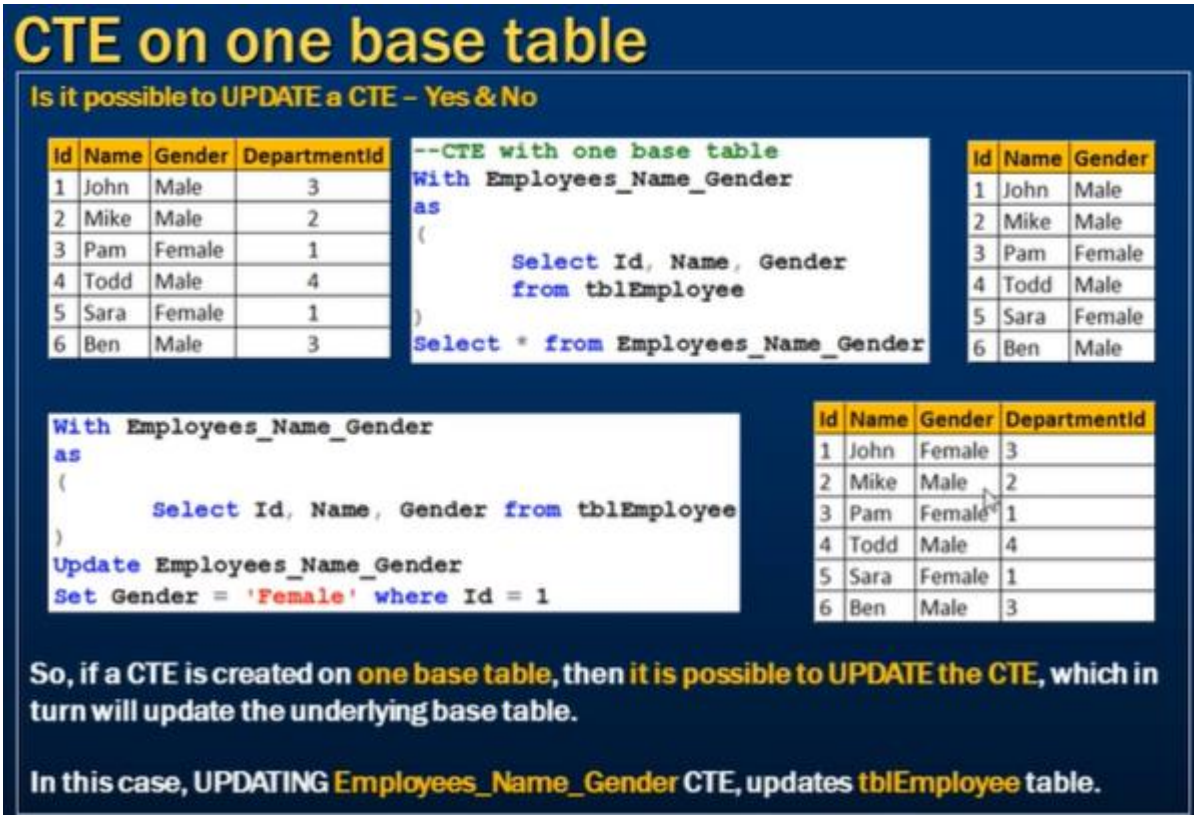
```
With EmployeesCountBy_Payroll_IT_Dept(DepartmentName, Total)
as
(
    Select DeptName, COUNT(Id) as TotalEmployees
    from tblEmployee
    join tblDepartment
    on tblEmployee.DepartmentId = tblDepartment.DeptId
    where DeptName IN ('Payroll','IT')
    group by DeptName
),
EmployeesCountBy_HR_Admin_Dept(DepartmentName, Total)
as
(
    Select DeptName, COUNT(Id) as TotalEmployees
    from tblEmployee
    join tblDepartment
    on tblEmployee.DepartmentId = tblDepartment.DeptId
    where DeptName IN ('HR','Admin')
    group by DeptName
)
Select * from EmployeesCountBy_HR_Admin_Dept
UNION
Select * from EmployeesCountBy_Payroll_IT_Dept
```

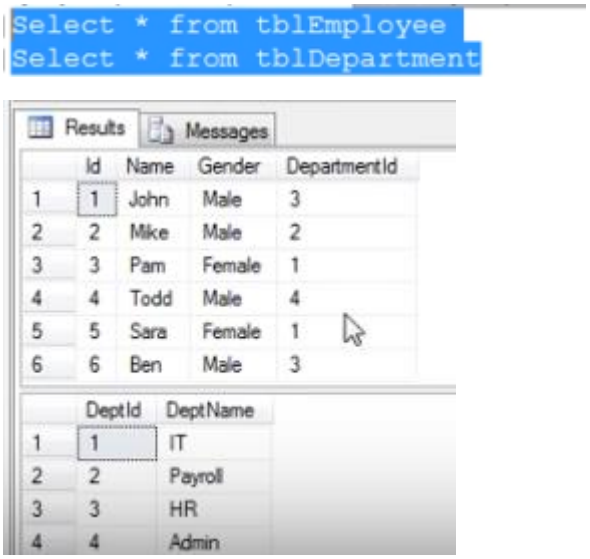
Results | Messages

	DepartmentName	Total
1	Admin	1
2	HR	2
3	IT	2
4	Payroll	1

Updatable common table expressions

A common table expression is a temporary result set. It can be considered as a derived table. Is it possible to update a CTE? The answer is yes and no. Under circumstances yes, under other circumstances no.





```
--CTE with one base table
With Employees_Name_Gender
as
(
    Select Id, Name, Gender
    from tblEmployee
)
Select * from Employees_Name_Gender
```

Results | Messages

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

```
--CTE with one base table
With Employees_Name_Gender
as
(
    Select Id, Name, Gender
    from tblEmployee
)
Select * from Employees_Name_Gender
```

Results | Messages

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

Now, we shall update the CTE.

```
With Employees_Name_Gender
as
(
    Select Id, Name, Gender from tblEmployee
)
Update Employees_Name_Gender Set Gender = 'Female' where Id = 1
```

We can see that tblEmployee got updated.

```
Select * from tblEmployee
```

Results | Messages

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

Thus if CTE is based on 1 one base table then it is possible to update the CTE, which in turn updates the base table.

CTE on 2 base tables

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

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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 on 2 base tables, Update affecting only one base table
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
```

Note: If a CTE is based on more than one table, and if the UPDATE affects only one base table, then the UPDATE is allowed.

CTE on 2 base tables

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

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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 on 2 base tables, Update affecting more than one base table
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
```

Note: 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.

CTE on 2 base tables

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

DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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

```
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
```

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

DeptId	DeptName
1	IT
2	Payroll
3	IT
4	Admin

Note: 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)

Updatable CTE

1. If a CTE is based on a single base table, then the UPDATE succeeds and works as expected.

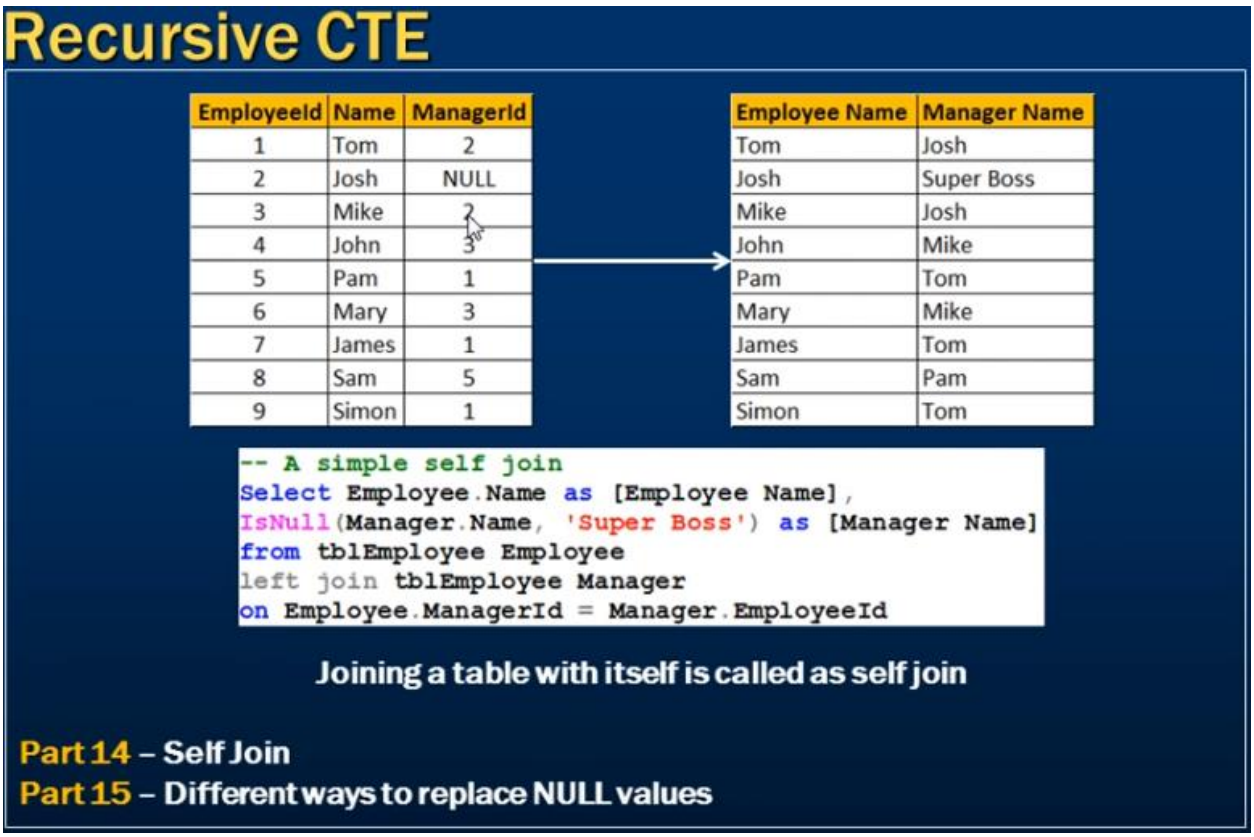
2. If a CTE is based on more than one base table, and if the UPDATE affects multiple base tables, the update is not allowed and the statement terminates with an error.

3. If 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

A **recursive CTE** is one that references itself within that **CTE**. The **recursive CTE** is useful when working with hierarchical data because the **CTE** continues to execute until the query returns the entire hierarchy. A typical example of hierarchical data is a table that includes a list of employees.

Below we have an Employees table, this has employee ID, Name, and Manager ID. Since a manager is also an employee hence manager is also listed as an employee in the same table. If someone asks who is the manager of Mike then we look up in the employees table and find that ManagerId is 2 and 2 is the EmployeeId of Josh, thus Josh is the manager of Mike. So we are looking up the ManagerId column and then looking up another column in the same table. So we are referencing the same table. This is called a **Self Referencing Table**. So if we have to create a table like on right to list Employee Name and Manager Name then we have to join this employees table with itself and we call it a Self Join.



We can see that Josh does not have a manager and ManagerId for Josh is NULL, hence for Josh we are displaying 'Super Boss' with the help of IsNull function.

What if we want to display the hierarchy with managers along with Employee Name and Manager Name i.e. the Level of the employee in the organization as displayed in the chart below.

If we take a closer look, Josh is the top boss as he does not have a manger and is at the top of the hierarchy, there are two who report direct to Josh, Tom and Mike.

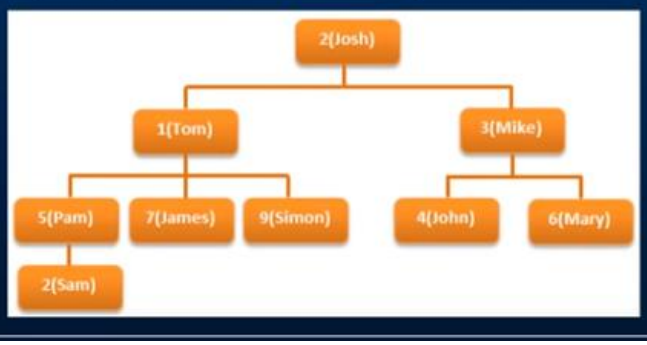
Let us see how to solve this using Recursive CTE.

Recursive CTE

Along with Employee and their Manager name, we also want to display their level in the organization

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

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



Recursive 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
```

Note: Why have we wrapped [Level] in square brackets? This is because Level is a key word in SQL server, and if we want to use a keyword as a column name then we have to wrap it in square brackets.

```sql
--Recursive 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
```

Results | Messages

	Employee	Manager	Level
1	Josh	Super Boss	1
2	Tom	Josh	2
3	Mike	Josh	2
4	John	Mike	3
5	Mary	Mike	3
6	Pam	Tom	3
7	James	Tom	3
8	Simon	Tom	3
9	Sam	Pam	4

Database normalization

What is Normalization

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

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

Problems of Data Redundancy
1. Disk Space Wastage
2. Data Inconsistency
3. DML queries can become slow

Normalized Table Design

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

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.

First Normal Form (1NF)

A table is said to be in 1NF, if
1. The data in each column **should be atomic.** No **multiple values,** separated by comma.
2. The table does not contain any **repeating column groups**
3. Identify each record uniquely using **primary key.**

Non Atomic Employee Column

DeptName	Employee
IT	Sam, Mike, Shan
HR	Pam

Problems of Non Atomic Columns

It is not possible to SELECT, INSERT, UPDATE and DELETE just one employee

No Repeating Column Groups

DeptName	Employee1	Employee2	Employee3
IT	Sam	Mike	Shan
HR	Pam		

Problems of Repeating Column Groups

More than 3 employees - Table structure change required
Less than 3 employees - Wasted disk space

First Normal Form
1 NF

Primary Key

DeptId	DeptName
1	IT
2	HR

Foreign Key

DeptId	Employee
1	Sam
1	Mike
1	Shan
2	Pam

First Normal Form
1 NF

Second normal form and third normal form

Second Normal Form (2NF)

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.

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

Problems of Data Redundancy

1. Disk Space Wastage
2. Data Inconsistency
3. DML queries can become slow

Table Design in Second Normal Form

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

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

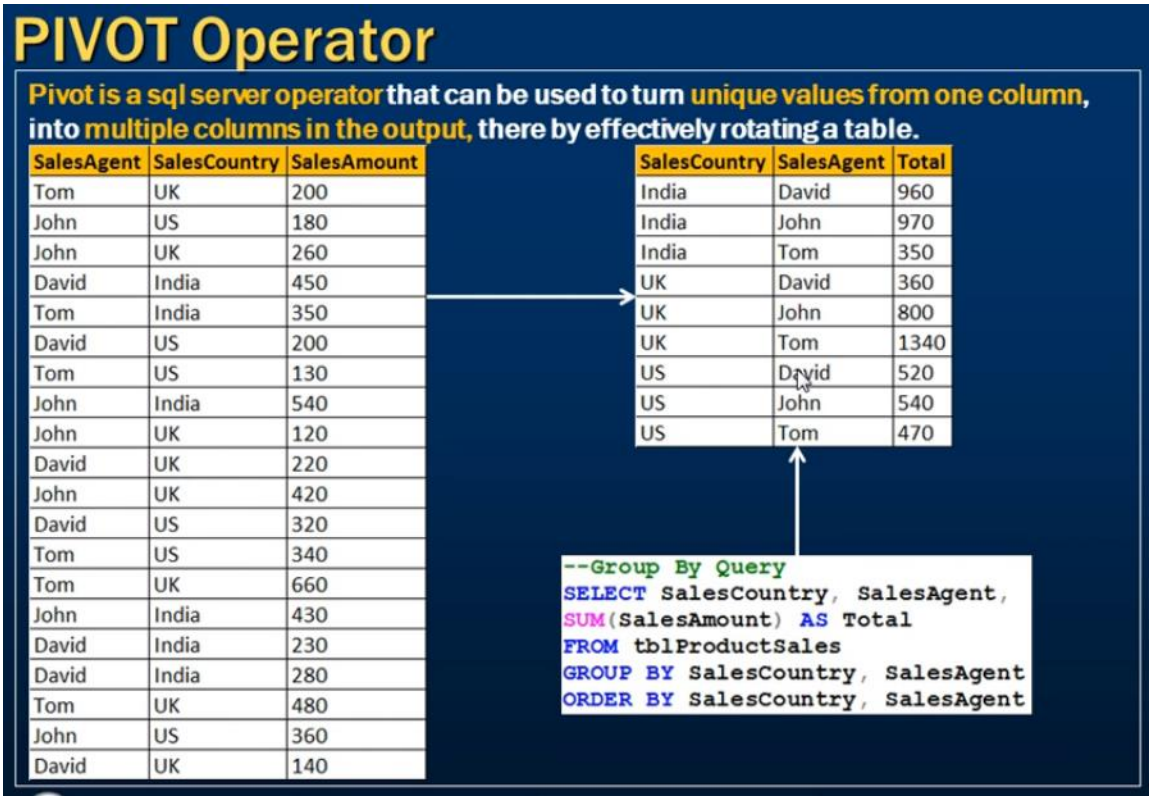
EmpId	EmployeeName	Gender	Salary	DeptName	DeptHead
1	Sam	Male	4500	IT	John
2	Pam	Female	2300	HR	Mike
3	Simon	Male	1345	IT	John
4	Mary	Female	2567	HR	Mike
5	Todd	Male	6890	IT	John

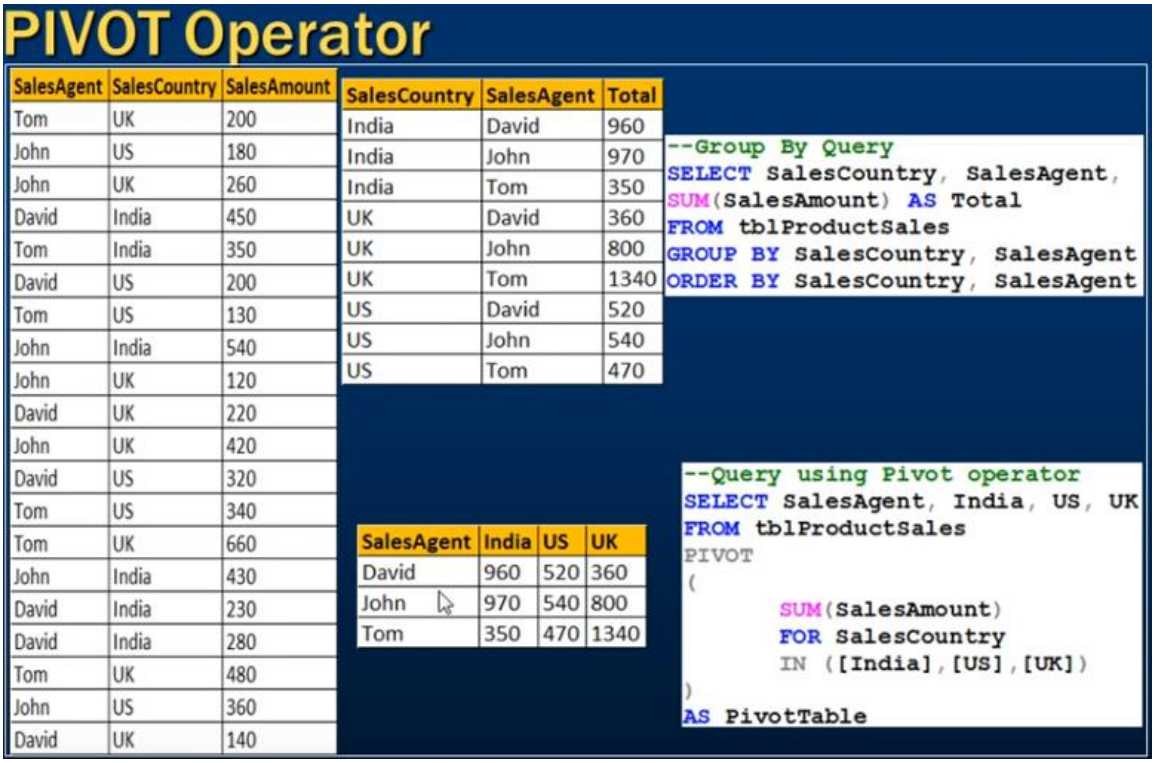
EmpId	EmployeeName	Gender	Salary	DeptId
1	Sam	Male	4500	1
2	Pam	Female	2300	2
3	Simon	Male	1345	1
4	Mary	Female	2567	2
5	Todd	Male	6890	1

DeptId	DeptName	DeptHead
1	IT	John
2	HR	Mike

Pivot

SQL Server `PIVOT` operator rotates a table-valued expression. It turns the unique values in one column into multiple columns in the output and performs aggregations on any remaining column values.





PIVOT Operator

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

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

```
--Group By Query
SELECT SalesCountry, SalesAgent,
SUM(SalesAmount) AS Total
FROM tblProductSales
GROUP BY SalesCountry, SalesAgent
ORDER BY SalesCountry, SalesAgent
```

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
```

PIVOT Operator

Id	SalesAgent	SalesCountry	SalesAmount
1	Tom	UK	200
2	John	US	180
3	John	UK	260
4	David	India	450
5	Tom	India	350
6	David	US	200
7	Tom	US	130
8	John	India	540
9	John	UK	120
10	David	UK	220
11	John	UK	420
12	David	US	320
13	Tom	US	340
14	Tom	UK	660
15	John	India	430
16	David	India	230
17	David	India	280
18	Tom	UK	480
19	John	US	360
20	David	UK	140

```
--Query using Pivot operator
SELECT SalesAgent, India, US, UK
FROM tblProductSales
PIVOT
(
        SUM(SalesAmount)
        FOR SalesCountry
        IN ([India],[US],[UK])
)
AS PivotTable
```

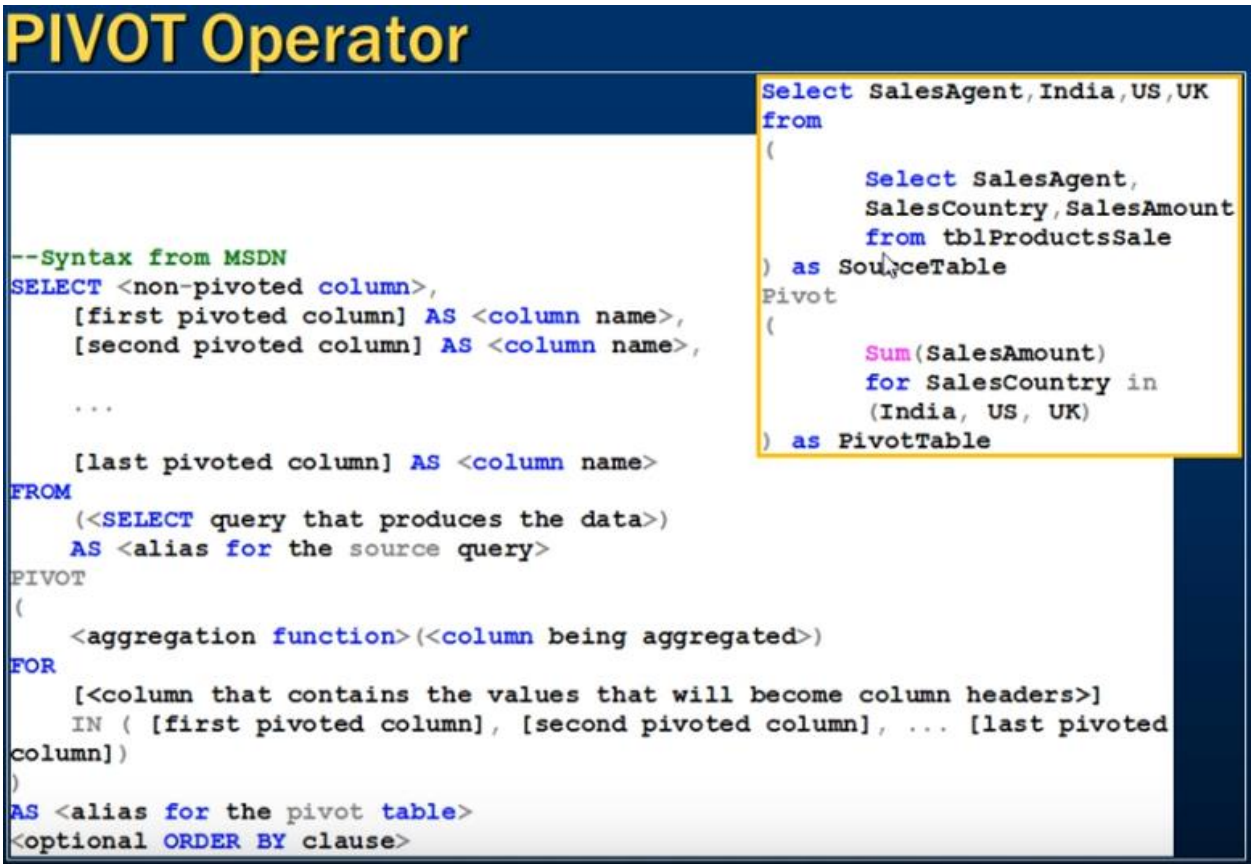
```
Select SalesAgent,India,US,UK
from
(
        Select SalesAgent,
        SalesCountry,SalesAmount
        from tblProductsSale
) as SourceTable
Pivot
(
        Sum(SalesAmount)
        for SalesCountry in
        (India, US, UK)
) as PivotTable
```

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

CROSS TAB FORMAT:

The source table is the derived table containing only the fields that are required to make the pivot table.

```
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
```



Error handling in SQL Server 2000

Error handling

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.

Error Handling in SQL Server 2000 - @@Error
Error Handling in SQL Server 2005 & later - Try...Catch

Note: 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.

tblProduct

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

tblProductSales

ProductSalesId	ProductId	QuantitySold
1	1	10
2	1	10

RAISERROR('Error Message', ErrorSeverity, ErrorState)
Create and return custom errors
Severity level = 16 (indicates general errors that can be corrected by the user)
State = Number between 1 & 255. RAISERROR only generates errors with state from 1 through 127.

```
Execute spSellProduct 1, 10

Create Procedure spSellProduct 1, 10
@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
```

If in the Else block, we comment out Set @MaxProductSalesId += 1

Then, the second insert statement will not take place due to primary key constraint and therefore, we need to introduce @@Error for rolling back the entire transaction to avoid that.

```
      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 transaction
            print 'Transaction rolled back'
        End
        else
        Begin
            Commit Tran
            print 'Transaction Committed'
        End
      End
End
```

@@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.

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.

```
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
if(@@ERROR <> 0)
        Print 'Error Occured'
Else
        Print 'No Errors'
```

```
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 successfully executed
if(@@ERROR <> 0)
        Print 'Error Occured'
Else
        Print 'No Errors'
```

@@Error System function

```
Declare @Error int
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
Set @Error = @@ERROR
Select * from tblProduct
if(@Error <> 0)
        Print 'Error Occured'
Else
        Print 'No Errors'
```

Error handling in sql server 2005 and Later

Error handling

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.

Error Handling in SQL Server 2000 - @@Error
Error Handling in SQL Server 2005 & later - Try...Catch

tblProduct

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

tblProductSales

ProductSalesId	ProductId	QuantitySold
1	1	10
2	1	10

RAISERROR('Error Message', ErrorSeverity, ErrorState)
Create and return custom errors
Severity level = 16 (indicates general errors that can be corrected by the user)
State = Number between 1 & 255. RAISERROR only generates errors with state from 1 through 127.

Try/Catch Syntax

```
--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]
```

```
--System Functions to retrieve
--error information:
ERROR_NUMBER()
ERROR_MESSAGE()
ERROR_PROCEDURE()
ERROR_STATE()
ERROR_SEVERITY()
ERROR_LINE()
```

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.

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

```
Alter 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

Basically, a **transaction** is a unit of work that is performed against a database. This work can be performed manually, such as an UPDATE statement you issue in **SQL Server** Management Studio or an application that INSERTS data into the database. These are all **transactions**.

Transactions

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

Note: NOT able to see the un-committed changes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

Isolation Levels : Covered in a later session

Transactions

tblPhysicalAddress

AddressId	EmployeeNumber	HouseNumber	StreetAddress	City	PostalCode
1	101	#10	King Street	LONDOON	CR27DW

tblMailingAddress

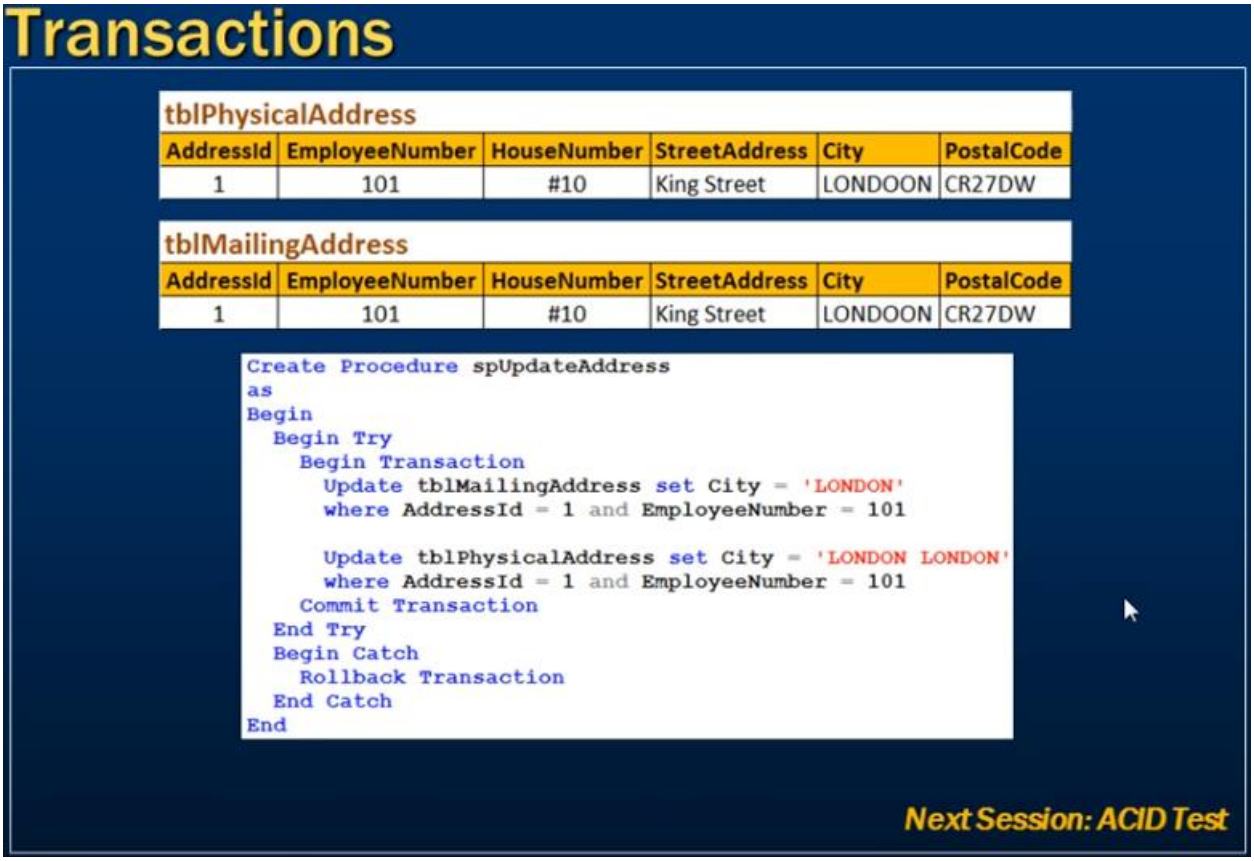
AddressId	EmployeeNumber	HouseNumber	StreetAddress	City	PostalCode
1	101	#10	King Street	LONDOON	CR27DW

```
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
```

The following transaction will succeed.

```
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
        Print 'Transaction Committed'
    End Try
    Begin Catch
        Rollback Transaction
        Print 'Transaction Rolled Back'
    End Catch
End
```

The following transaction will fail and will be rolled back because though the first update succeeds the second update fails as city length is more than 10 hence even the first update will be rolled back.



Transactions ACID Tests

Transaction ACID Test

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

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.

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.

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
```

Transaction ACID Test

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.

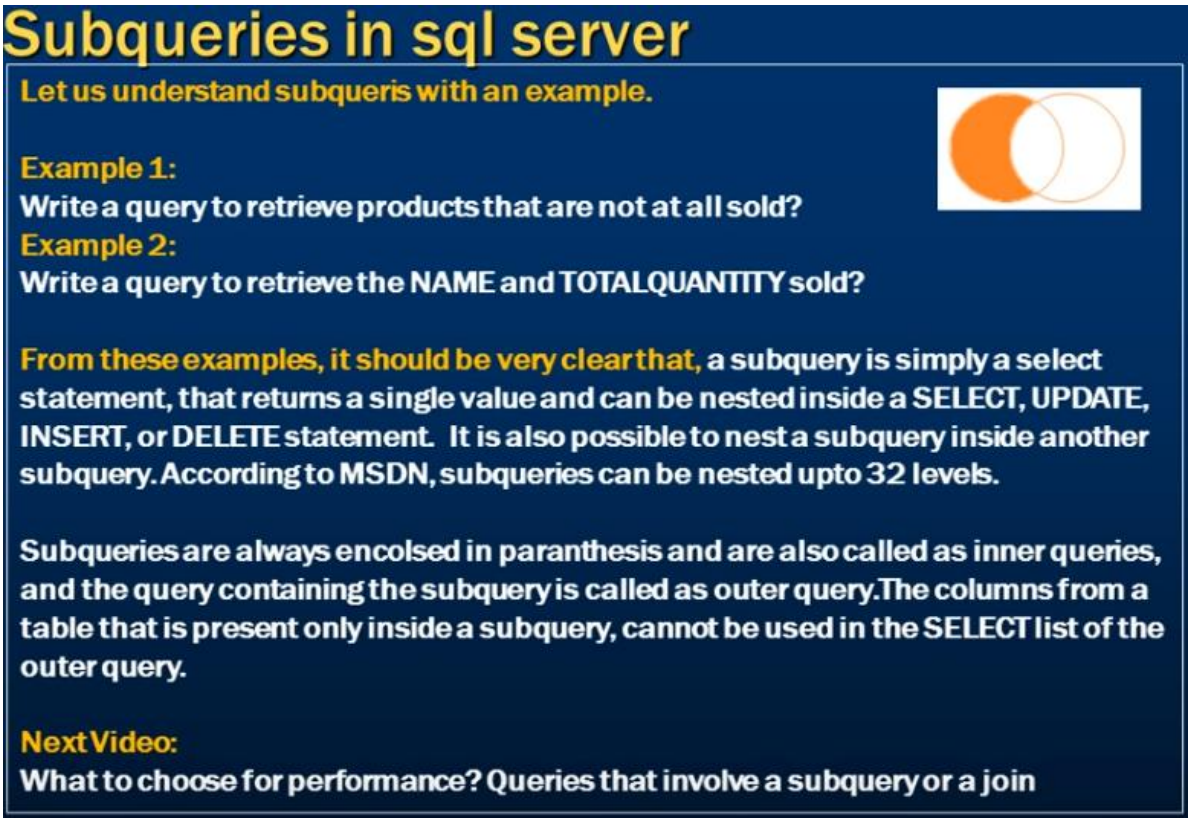
Durable : Once a change is made, it is permanent. If a system error or power failure occurs before a set of commands is complete, those commands are undone and the data is restored to its original state once the system begins running again.

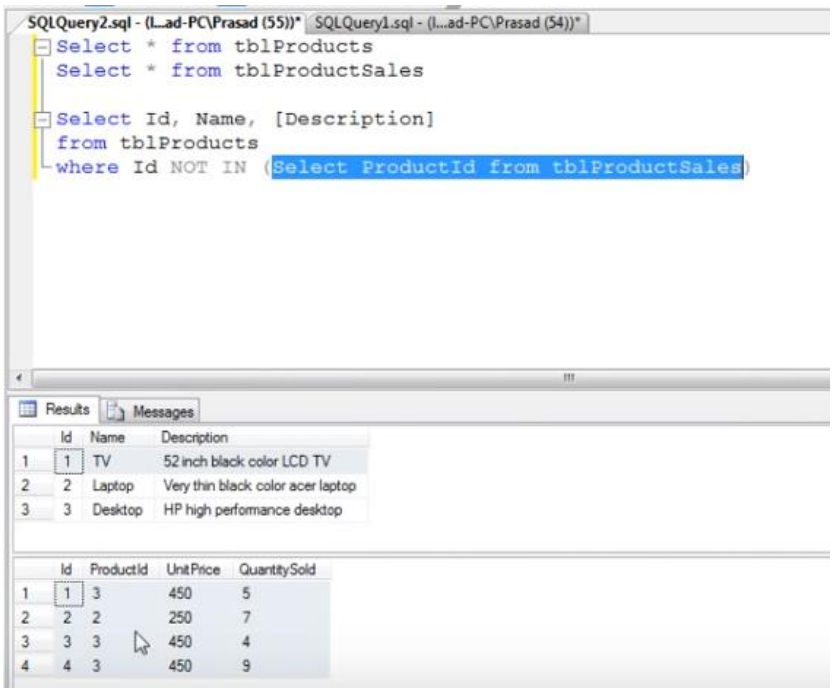
Next Session : Isolation Levels in SQL Server

Want to receive email alerts when new videos are uploaded, subscribe to my youtube channel: www.YouTube.com/kudvenkat

Subqueries

A **Subquery** or Inner query or a Nested query is a query within another **SQL** query and embedded within the WHERE clause. A **subquery** is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.





```
Select * from tblProducts
Select * from tblProductSales

Select Id, Name, [Description]
from tblProducts
where Id NOT IN (Select distinct ProductId from tblProductSales)
```

	Id	Name	Description
1	1	TV	52 inch black color LCD TV

REPLACING JOINS WITH SUBQUERIES:

```
Select * from tblProducts
Select * from tblProductSales

Select Id, Name, [Description]
from tblProducts
where Id NOT IN (Select distinct ProductId from tblProductSales)

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

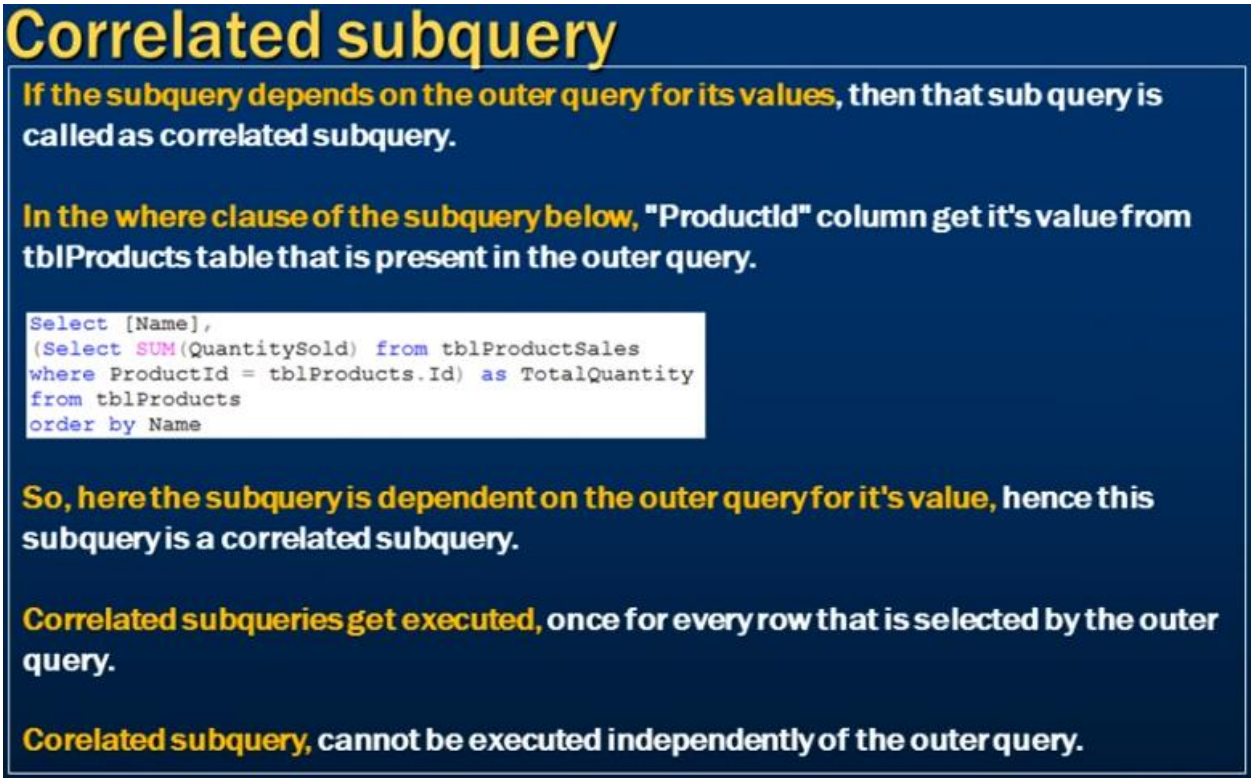
	Id	Name	Description
1	1	TV	52 inch black color LCD TV

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

Select Name, SUM(QuantitySold) as QtySold
from tblProducts
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
Group by Name
```

Name	QtySold
Desktop	18
Laptop	7
TV	NULL

Correlated subquery

In a SQL database query, a correlated subquery is a subquery that uses values from the outer query. Because the subquery may be evaluated once for each row processed by the outer query, it can be slow.




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

Results | Messages

	Id	Name	Description
1	1	TV	52 inch black color LCD TV
2	2	Laptop	Very thin black color acer laptop
3	3	Desktop	HP high performance desktop

	Id	ProductId	UnitPrice	QuantitySold
1	1	3	450	5
2	2	2	250	7
3	3	3	450	4
4	4	3	450	9

```
Select Id, Name, Description
from tblProducts where Id NOT IN (Select distinct ProductId from tblProductSales)
```

Results | Messages

	ProductId
1	2
2	3

```
Select Id, Name, Description
from tblProducts where Id NOT IN (Select distinct ProductId from tblProductSales)
```

Results | Messages

	Id	Name	Description
1	1	TV	52 inch black color LCD TV

AN EXAMPLE OF NON-CORRELATED SUBQUERY WHICH DOES NOT DEPEND UPON THE OUTER QUERY. IT CAN GET EXECUTED BY ITSELF WHETHER THE OUTER QUERY IS EXECUTED OR NOT.

Let us use a non-corelated subquery to find all the products that we have sold and the total quantity. Basically, we have 3 products.

```
Select Name,
(Select SUM(QuantitySold) from tblProductSales where ProductId = tblProducts.Id)
from tblProducts
```

Results | Messages

	Name	(No column name)
1	TV	NULL
2	Laptop	7
3	Desktop	18

Creating a large table with random data for performance testing

- Inserting large amount of random data into sql server tables for performance testing

There are 3 parts of this sql script.

The first part will check if tables exist and if they exist then they will be dropped.

```
-- If Table exists drop and recreate
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
```

The second part will recreate the 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
)
```

And, the final part will insert random data in those tables.

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

While(@Id <= 10000)
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
```

TO CHECK THE NAMES OF ALL THE TABLES IN THE CURRENT DATABASE: When we execute the selected part of the script:

```
-- If Table exists drop and recreate
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
```

This query returns the names of tables present in the Sample database because we executed this query in the context of Sample database (as selected in SSMS).

Results | Messages

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE
1	Sample	dbo	tblProducts	BASE TABLE
2	Sample	dbo	tblEmployees	BASE TABLE
3	Sample	dbo	tblProductSales	BASE TABLE
4	Sample	dbo	tblLog	BASE TABLE
5	Sample	dbo	tblUsers	BASE TABLE
6	Sample	dbo	tblResetPasswordRequests	BASE TABLE

EXISTS() FUNCTION CHECKS IF THE TABLE tblProductSales given in where clause exists.

```
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
```

Thus if table is found by Exists() function then Drop Table statement will drop the table. Then tables are created.

```
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
)
```

Next we insert 10000 products in the table.

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

While(@Id <= 10000)
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
```

When we execute the statement we get 10000 rows.

```
Messages
(1 row(s) affected)
9998

(1 row(s) affected)
9999

(1 row(s) affected)
10000
```

If we look at the table we find the inserted rows.

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

Results | Messages

	Id	Name	Description
1	1	Product - 1	Product - 1 Description
2	2	Product - 2	Product - 2 Description
3	3	Product - 3	Product - 3 Description
4	4	Product - 4	Product - 4 Description
5	5	Product - 5	Product - 5 Description
6	6	Product - 6	Product - 6 Description
7	7	Product - 7	Product - 7 Description

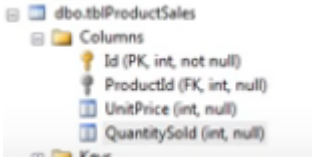
Insert Sample data, for this we use Round() and Rand() functions.

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

While(@Counter <= 15000)
Begin
    select @RandomProductId = Round(((@UpperLimitForProductId - @LowerLimitForProductId) *
    select @RandomUnitPrice = Round(((@UpperLimitForUnitPrice - @LowerLimitForUnitPrice) *
    select @RandomQuantitySold = Round(((@UpperLimitForQuantitySold - @LowerLimitForQuantit

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

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

Generating Random numbers within a given range.

```
Declare @LL int
Set @LL = 1

Declare @UL int
Set @UL = 5

Declare @Rand int

While(1=1)
Begin
    Select @Rand = Round(((@UL - @LL) * RAND() + 1), 0)
    Print @Rand

    If(@Rand < 1 OR @Rand >= 5)
    Begin
        Print 'Error - ' + Cast(@Rand as nvarchar(4))
        break
    End
End
```

CREATING Table tblProductSales WITH FOLLOWING COLUMNS:



We create 2 variables: UpperLimitForProductId and LowerLimitForProductId

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

set @LowerLimitForProductId = 1
set @UpperLimitForProductId = 8500

-- 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
```

Finally, we are looping through a While loop and inserting 15000 records with random data.

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

While(@Counter <= 15000)
Begin
    select @RandomProductId = Round(((@UpperLimitForProductId - @LowerLimitForProductId) *
    select @RandomUnitPrice = Round(((@UpperLimitForUnitPrice - @LowerLimitForUnitPrice) *
    select @RandomQuantitySold = Round(((@UpperLimitForQuantitySold - @LowerLimitForQuantit

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

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

SubQuery or Joins

Performance - SubQueries or Joins

According to MSDN, in most cases, there is usually no performance difference between queries that uses sub-queries and equivalent queries using joins.

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.

In general joins work faster than sub-queries, but in reality it all depends on the execution plan that is generated by SQL Server. Id does not matter how we have written the query, SQL Server will always transform it on an execution plan. If it is "smart" enough to generate the same plan from both queries, you 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.

When we execute the following query it takes 7 seconds.

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

Results | Messages

	Id	Name	Description
1	1	Product - 1	Product - 1 Description
2	2	Product - 2	Product - 2 Description
3	3	Product - 3	Product - 3 Description
4	4	Product - 4	Product - 4 Description
5	6	Product - 6	Product - 6 Description
6	8	Product - 8	Product - 8 Description
7	9	Product - 9	Product - 9 Description
8	10	Product - 10	Product - 10 Description
9	12	Product - 12	Product - 12 Description
10	14	Product - 14	Product - 14 Description
11	15	Product - 15	Product - 15 Description
12	17	Product - 17	Product - 17 Description
13	18	Product - 18	Product - 18 Description
14	20	Product - 20	Product - 20 Description
15	21	Product - 21	Product - 21 Description
16	23	Product - 23	Product - 23 Description
17	24	Product - 24	Product - 24 Description

Subscribe

Query executed successfully. | (local) (10.0 RTM) | Prasad-PC\Prasad (52) | Sample | 00:00:07 | 306199 rows

When we run the same query using Joins

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

We should clear the query cache before running the above query as it may impact the query running time. HOW TO CLEAR QUERY CACHE?

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

Now run the above query using Joins, and this too takes 7 seconds.

Results | Messages

	Id	Name	Description
1	1	Product - 1	Product - 1 Description
2	2	Product - 2	Product - 2 Description
3	3	Product - 3	Product - 3 Description
4	4	Product - 4	Product - 4 Description
5	6	Product - 6	Product - 6 Description
6	8	Product - 8	Product - 8 Description
7	9	Product - 9	Product - 9 Description
8	10	Product - 10	Product - 10 Description
9	12	Product - 12	Product - 12 Description
10	14	Product - 14	Product - 14 Description
11	15	Product - 15	Product - 15 Description
12	17	Product - 17	Product - 17 Description
13	18	Product - 18	Product - 18 Description
14	20	Product - 20	Product - 20 Description
15	21	Product - 21	Product - 21 Description
16	23	Product - 23	Product - 23 Description
17	24	Product - 24	Product - 24 Description

Query executed successfully. | (local) (10.0 RTM) | Prasad-PC\Prasad (52) | Sample | 00:00:07

IN ABOVE CASE IT TAKES 7 SECONDS IN BOTH CASES.

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.

```
-- 4 seconds 93801
Select Id, Name, [Description]
from tblProducts
where Not Exists(Select * from tblProductSales where ProductId = tblProducts.Id)

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

Cursors

A **cursor** is a temporary work area created in system memory when a **SQL** statement is executed. A **cursor** is a set of rows together with a **pointer** that identifies a current row. ... In other words, a **cursor** can hold more than one row, but can process only one row at a time.

Cursors in SQL Server

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

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

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

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.

Cursors in SQL Server

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

DEMO

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.

CREATING A CURSOR FOR SELECT STATEMENT: The select statement will give a result set and we are asking for a cursor (kind of pointer) pointing to that result set.

```
Declare @ProductId int
Declare @Name nvarchar(30)

Declare ProductCursor CURSOR FOR
Select Id,Name from tblProducts where Id <= 1000

Open ProductCursor

Fetch Next from ProductCursor into @ProductId, @Name

While(@@FETCH_STATUS = 0)
Begin


    Fetch Next from ProductCursor into @ProductId, @Name

End

CLOSE ProductCursor
DEALLOCATE ProductCursor
```

- **When we say Open ProductCursor, the Select statement will be executed, and all the rows that match the where clause of select statement will be retrieved into the result set and we will have our Cursor pointing just above the first record. The cursor will be ready to retrieve the rows but not pointing to any row.**
- **When we say Fetch then cursor moves to first record and retrieves it requested data from that row. THE 'FETCH_STATUS = 0' UNTIL THE CURSOR REACHES THE END OF RESULT SET.**
- **CURSOR MOVE ROW BY ROW**
- **IT IS IMPORTANT TO CLOSE RESULT SET AFTER COMPLETION AND DEALLOCATE RESOURCES.**

```
Declare @ProductId int
Declare @Name nvarchar(30)

Declare ProductCursor CURSOR FOR
```

SQLQuery4.sql - (L...ad-PC\Prasad (54))* SQLQuery3.sql - (L...ad-PC\Prasad (55))*

```
Select ProductId from tblProductsales

Open ProductCursor

Fetch Next from ProductCursor into @ProductId

While(@@FETCH_STATUS = 0)
Begin
    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 ProductCursor into @ProductId
End

CLOSE ProductCursor
```

Replacing cursors using joins

Replacing cursors using joins

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.

NOTE: IT IS IMPORTANT TO PROVIDE ELSE PART IN CASE STATEMENT, OTHERWISE THE ROWS THAT WILL NOT MATCH THE CONDITIONS IN CASE STATEMENT WILL GET UPDATED TO <u>NULL VALUES.</u>

Let us run a query using Joins, this we also ran using cursors handling 600000 rows.

```
Update tblProductSales
set UnitPrice = Case
                    When Name='Product - 55' Then 55
                    When Name='Product - 65' Then 65
                    When Name LIKE 'Product - 100%' Then 1000
                    Else
                        UnitPrice
                End
from tblProductSales
join tblProducts
on tblProducts.Id = tblProductSales.ProductId
```

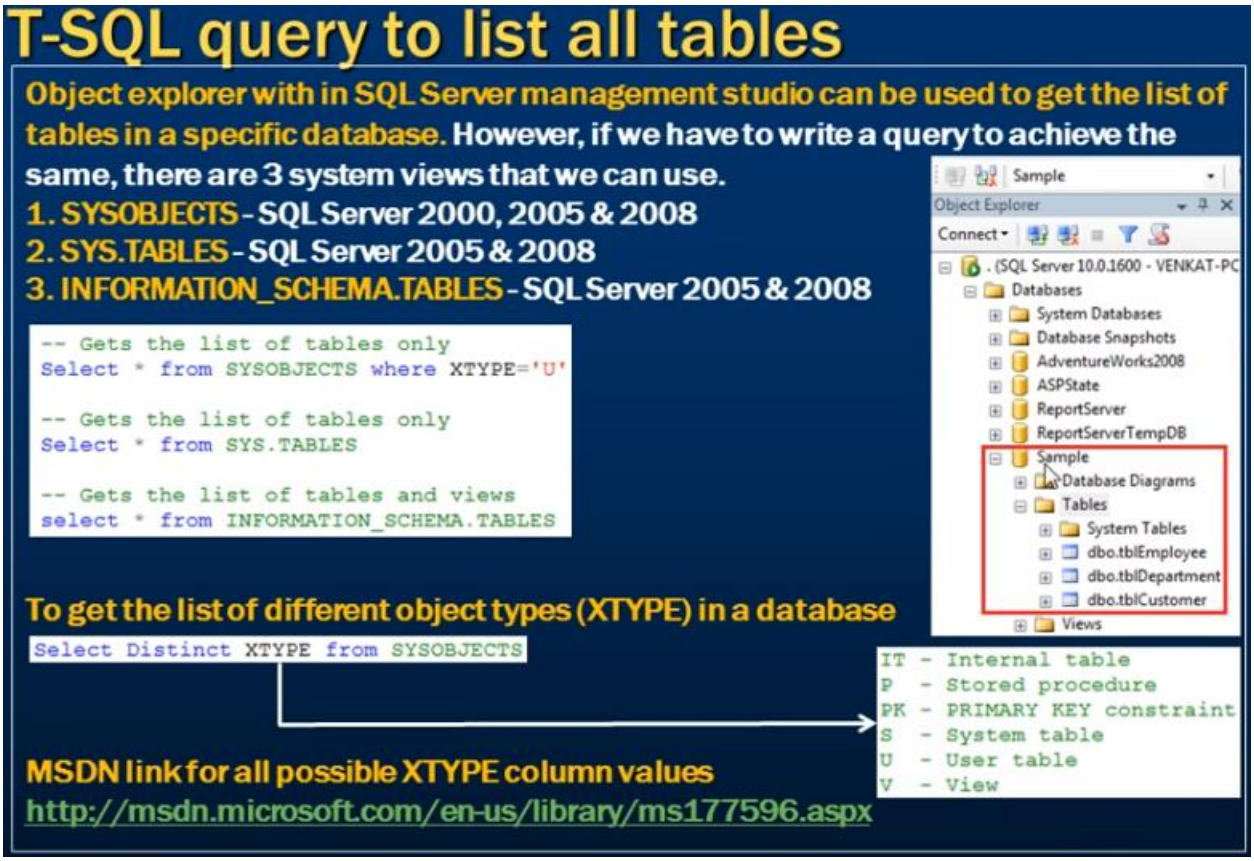
Messages

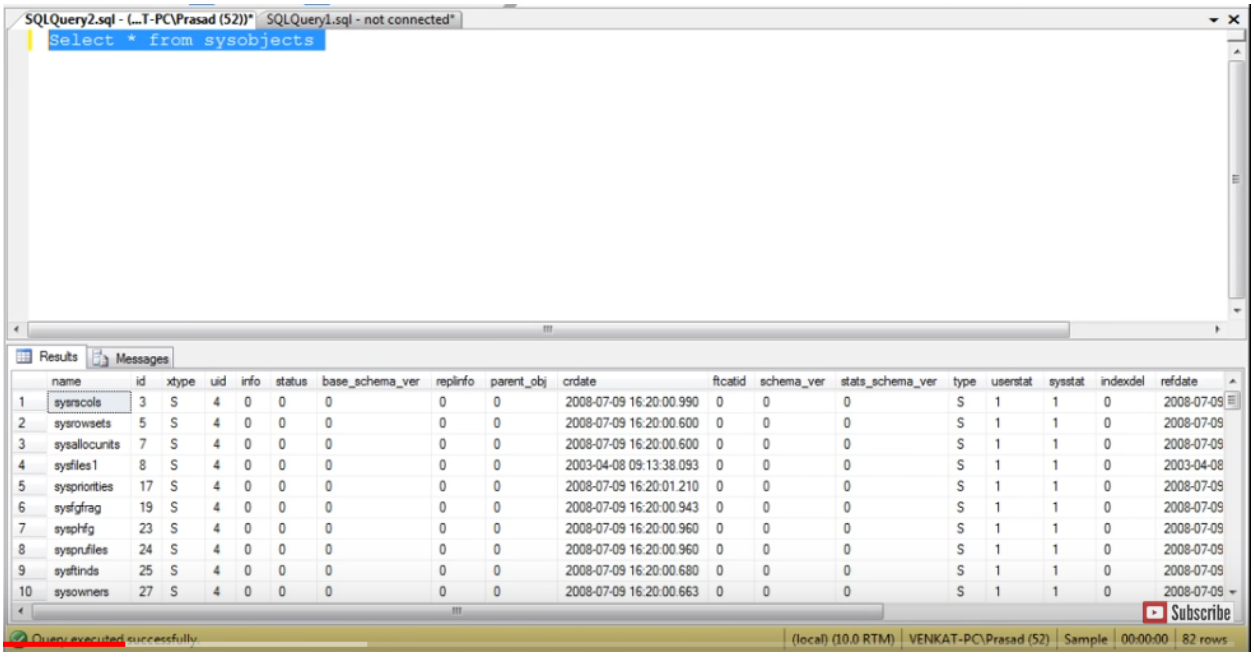
```
(600000 row(s) affected)
```

Query executed successfully. (local) (10.0 RTM) Prasad-PC\Prasad (53) Sample 00:00:06

As we can see, the query with Joins took only 6 seconds while the query using cursors took 43 seconds.

List all tables in a sql server database





	name	id	xtype	uid	info	status	base_schema_ver	replinfo	parent_obj	crdate	ftcatid	schema_ver	stats_schema_ver	type	userstat	sysstat	indexdel	refdate
1	sysrscols	3	S	4	0	0	0	0	0	2008-07-09 16:20:00.990	0	0	0	S	1	1	0	2008-07-09
2	sysrowsets	5	S	4	0	0	0	0	0	2008-07-09 16:20:00.600	0	0	0	S	1	1	0	2008-07-09
3	sysallocunits	7	S	4	0	0	0	0	0	2008-07-09 16:20:00.600	0	0	0	S	1	1	0	2008-07-09
4	sysfiles1	8	S	4	0	0	0	0	0	2003-04-08 09:13:38.093	0	0	0	S	1	1	0	2003-04-08
5	syspriorities	17	S	4	0	0	0	0	0	2008-07-09 16:20:01.210	0	0	0	S	1	1	0	2008-07-09
6	sysfgfrag	19	S	4	0	0	0	0	0	2008-07-09 16:20:00.943	0	0	0	S	1	1	0	2008-07-09
7	sysphfg	23	S	4	0	0	0	0	0	2008-07-09 16:20:00.960	0	0	0	S	1	1	0	2008-07-09
8	sysprufiles	24	S	4	0	0	0	0	0	2008-07-09 16:20:00.960	0	0	0	S	1	1	0	2008-07-09
9	sysftinds	25	S	4	0	0	0	0	0	2008-07-09 16:20:00.680	0	0	0	S	1	1	0	2008-07-09
10	sysowners	27	S	4	0	0	0	0	0	2008-07-09 16:20:00.663	0	0	0	S	1	1	0	2008-07-09

SQLQuery2.sql - (...T-PC\Prasad (52))* | SQLQuery1.sql - not connected*

```
Select * from sysobjects where xtype = 'U'
```

Results | Messages

	name	id	xtype	uid	info	status	base_schema_ver	replinfo	parent_obj	crdate	ftcatid	schema_ver	stats_schema_ver	type	userstat	sysstat	indexdel	refd
1	tblEmployee	1821249543	U	1	0	0	0	0	0	2013-06-21 19:26:32.990	0	0	0	U	1	3	0	20
2	tblDepartment	1885249771	U	1	0	0	0	0	0	2013-06-24 18:45:23.373	0	0	0	U	1	3	0	20
3	tblCustomer	1901249828	U	1	0	0	0	0	0	2013-06-24 18:45:45.783	0	0	0	U	1	3	0	20

We set xtype = 'U', in the query what does 'U' mean? In the given MSDN we find details.

xtype	**char(2)**	Object type. Can be one of the following object types: AF = Aggregate function (CLR) C = CHECK constraint D = Default or DEFAULT constraint F = FOREIGN KEY constraint L = Log FN = Scalar function FS = Assembly (CLR) scalar-function FT = Assembly (CLR) table-valued function IF = In-lined table-function IT = Internal table P = Stored procedure PC = Assembly (CLR) stored-procedure PK = PRIMARY KEY constraint (type is K) RF = Replication filter stored procedure S = System table SN = Synonym SQ = Service queue TA = Assembly (CLR) DML trigger TF = Table function TR = SQL DML Trigger TT = Table type U = User table UQ = UNIQUE constraint (type is K) V = View X = Extended stored procedure

SYS.VIEWS ALLOWS TO SEE ALL THE VIEWS IN DATABASE

SYS.PROCEDURES TO CHECK ALL STORED PROCEDURES

```
Select * from INFORMATION_SCHEMA.ROUTINES
```

Re-runnable scripts

Re-runnable SQL Server Scripts

What is a re-runnable SQL Script?

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

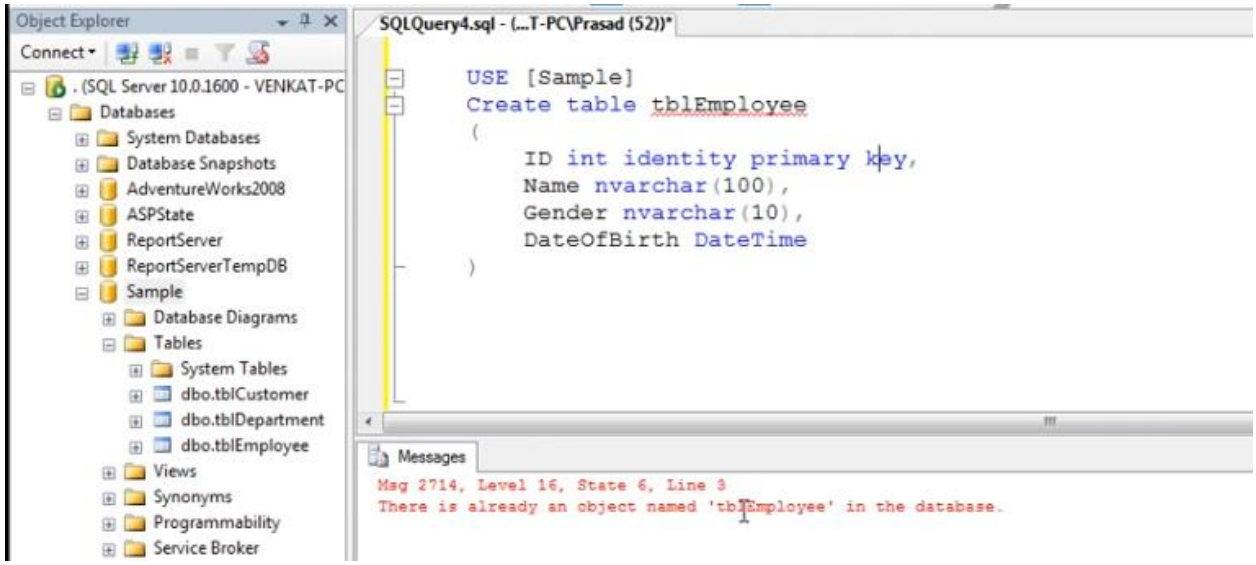
```
-- This script is not re-runnable
USE [Sample]
Create table tblEmployee
(
    ID int identity primary key,
    Name nvarchar(100),
    Gender nvarchar(10),
    DateOfBirth DateTime
)
```

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
```

When we run the following query again it throws error because the table already exists.



We make a script re-runnable by adding an if not exists clause.

```
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
```

Re-runnable SQL Server Scripts

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

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

Drop (if the table already exists) and re-create

```
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
)
```

Re-runnable SQL Server Scripts

This script is not re-runnable because, if the column exists we get a script error.

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

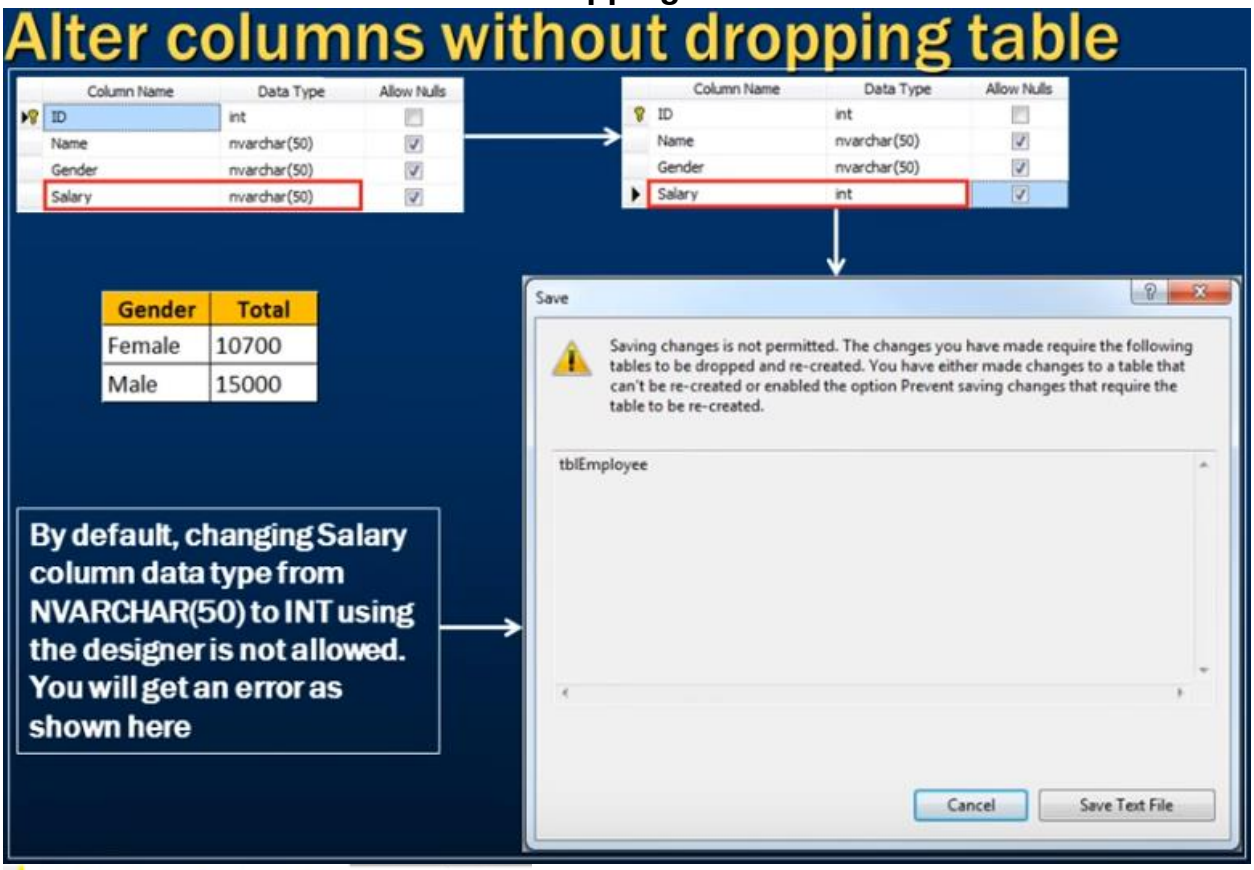
To make this script re-runnable, check for the column existence

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

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

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

Alter database table columns without dropping table




```
Select * from tblEmployee
```

Results | Messages

	ID	Name	Gender	Salary
1	1	Sara Nani	Female	4500
2	2	James Histo	Male	5300
3	3	Mary Jane	Female	6200
4	4	Paul Sensit	Male	4200
5	5	Mike Jen	Male	5500

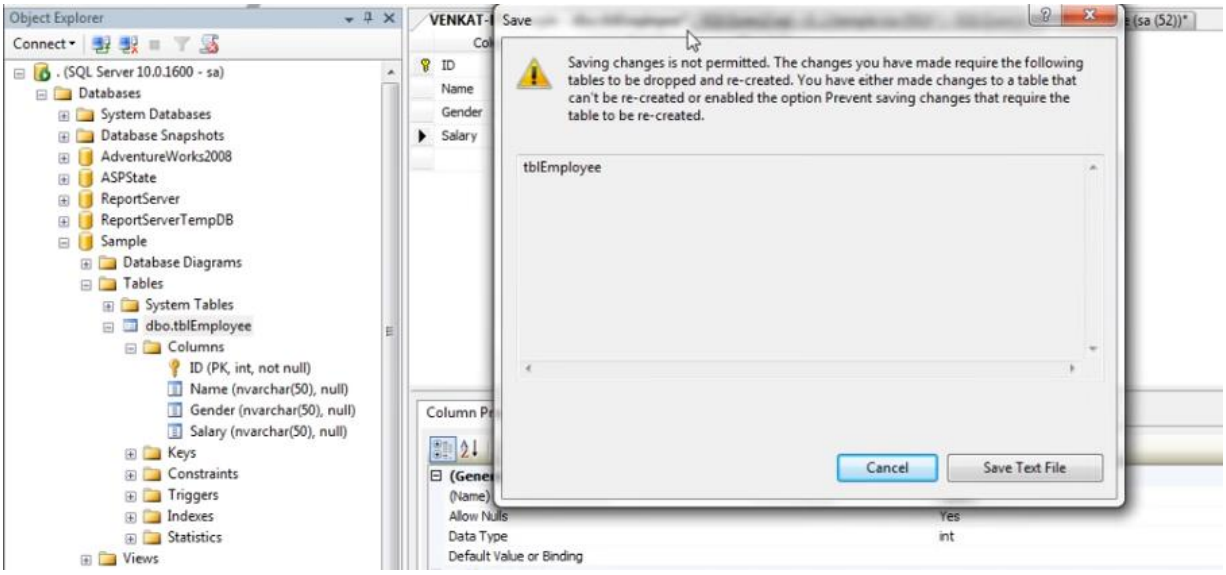
SQLQuery2.sql - (l...).Sample (sa (53))* | SQLQuery1.sql - (l...).Sample (sa (52))*

```
Select * from tblEmployee

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

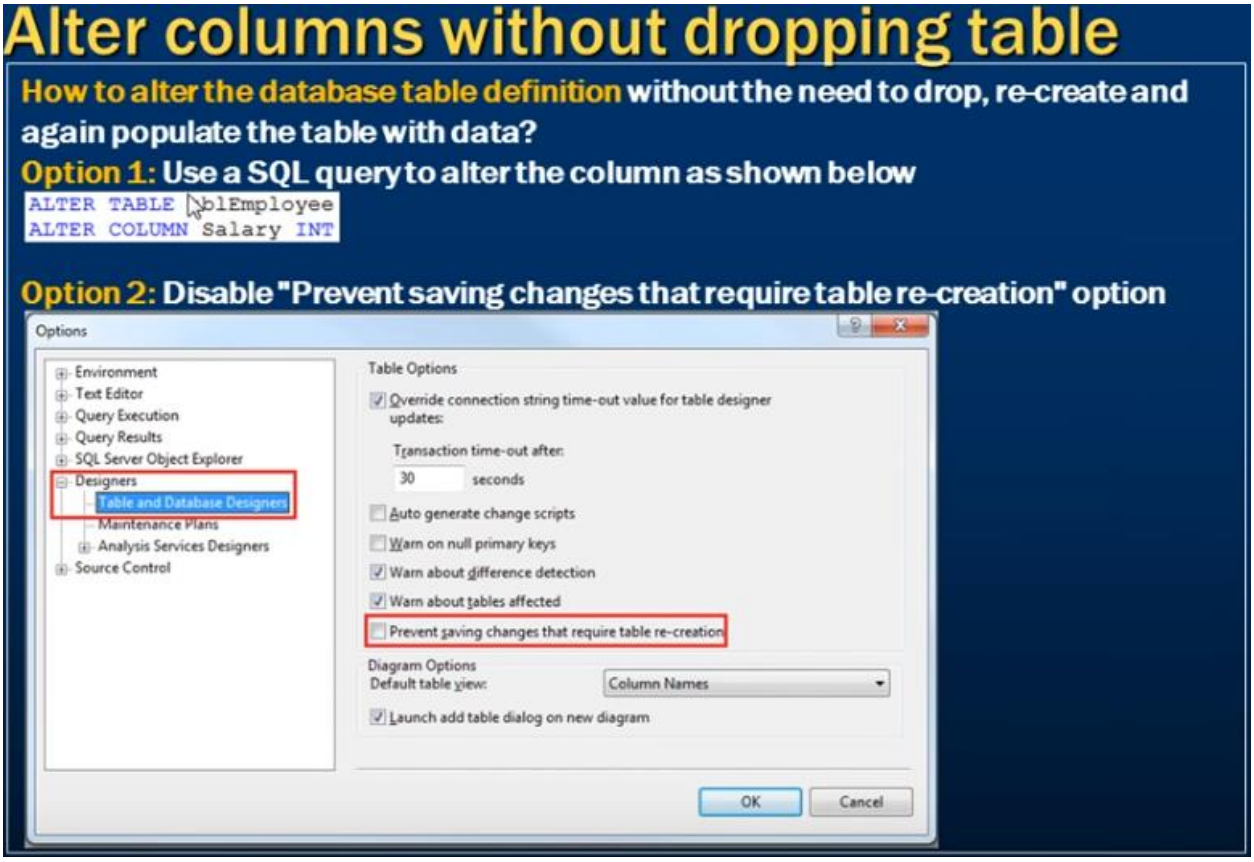
Messages

```
Msg 8117, Level 16, State 1, Line 1
Operand data type nvarchar is invalid for sum operator.
```

We got above error because salary column is nvarchar and used SUM function on it. So, we try to change the column property using SSMS designer and get error.



So how do we change column property, basically there are 2 ways as shown below.




```
Alter Table tblEmployee
Alter column Salary int
```

Optional parameters in sql server stored procedures

```
Select * from tblEmployee

Execute spSearchEmployees

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

```
Msg 201, Level 16, State 4, Procedure spSearchEmployees, Line 0
Procedure or function 'spSearchEmployees' expects parameter '@Name', which was not supplied.
```

SPECIFY DEFAULT VALUES FOR PARAMETERS TO AVOID ERROR

```
Select * from tblEmployee

Execute spSearchEmployees

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

Id	Name	Email	Age	Gender	HireDate

THIS WILL RETRIEVE ALL THE ROWS FROM TBLEMPLOYEE TABLE AS CONDITION EVALUATES TO TRUE FOR ALL THE FIELDS

SQLQuery2.sql - (l...).Sample (sa (53))* | SQLQuery1.sql - (l...).Sample (sa (52))*

```
Select * from tblEmployee

Execute spSearchEmployees @Gender 'Male'

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

Results | Messages

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

Merge Statement

Merge in SQL Server

If you are in need of the DVD with all the videos and PPT's, please visit http://pragimtech.com/order.aspx

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

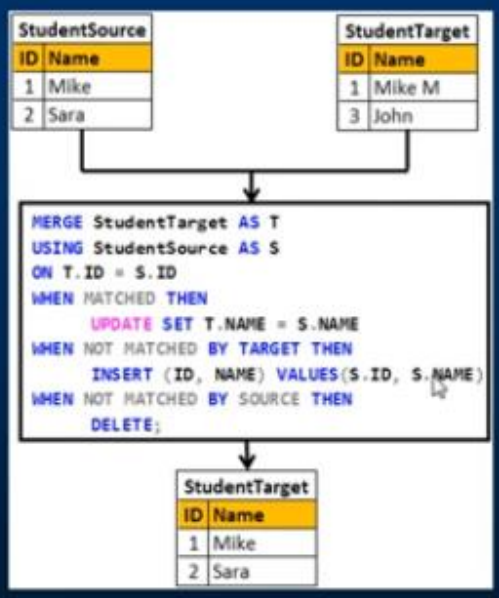
With merge statement we require 2 tables

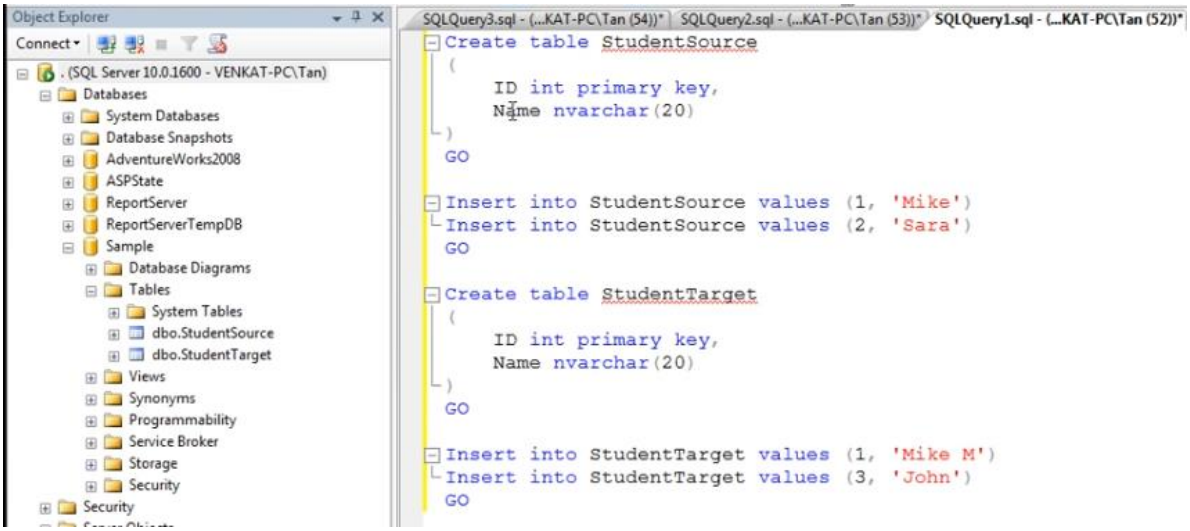
1. Source Table - Contains the changes that needs to be applied to the target table
2. Target Table - The table that require changes (Inserts, Updates and Deletes)

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, we can perform insert, update, and delete on the target table

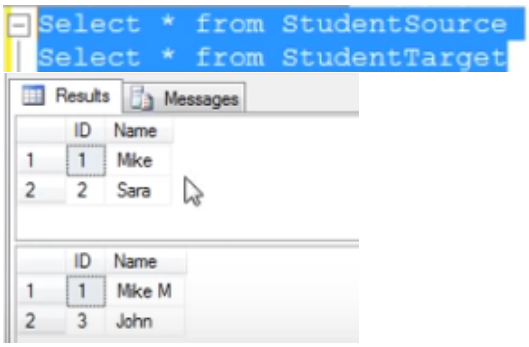
Merge in SQL Server - Example 1

```
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]
```

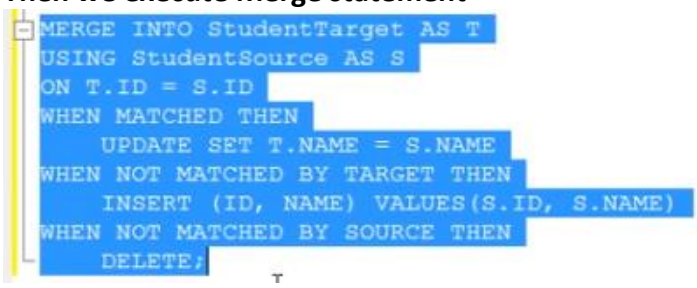




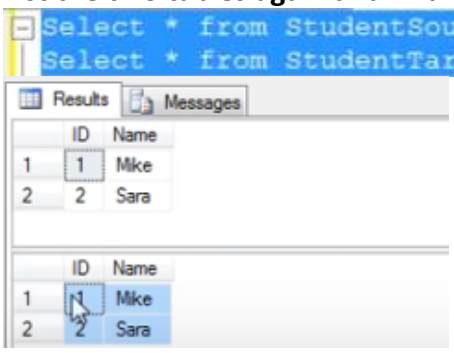
List the two tables



Then we execute Merge statement



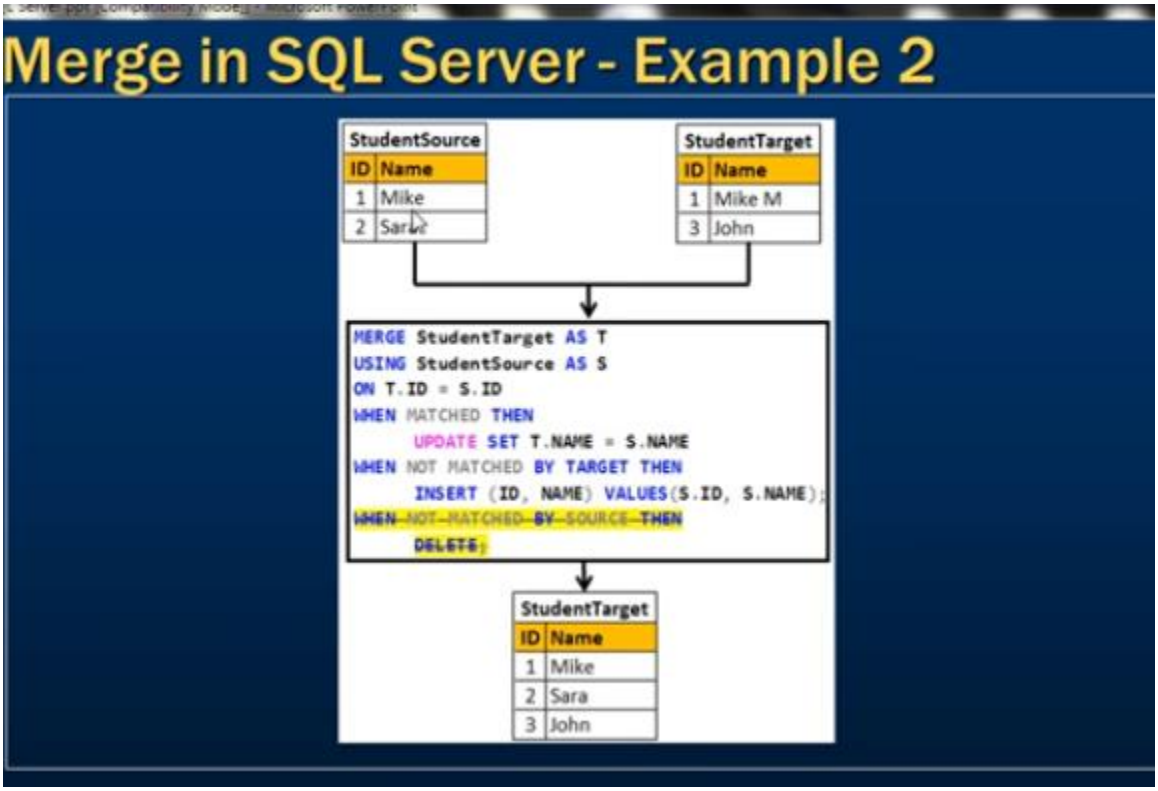
List the two tables again and find target table has been updated.



Thus update, insert, delete all have been executed in one Merge statement.

Imp Note: Merge statement must end with a semicolon.

WHEN WE DON'T WANT TO DELETE DATA IN TARGET TABLE




```
MERGE INTO 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);
```

We can see that John record is not deleted.

```
Select * from StudentSource
Select * from StudentTarget
```

Results | Messages

	ID	Name
1	1	Mike
2	2	Sara

	ID	Name
1	1	Mike
2	2	Sara
3	3	John

Concurrent transactions

A transaction is a group of commands that changes data in database and ensures that either all the commands in that transaction succeed or none of them.

SQL Server Concurrent Transactions

If you are in need of the DVD with all the videos and PPT's, please visit http://pragimtech.com/order.aspx

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

Id	AccountName	Balance
1	Mark	1000
2	Mary	1000

```
-- 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
```

SQL Server Concurrent Transactions

Common concurrency problems

- Dirty Reads
- Lost Updates
- Nonrepeatable Reads
- Phantom Reads

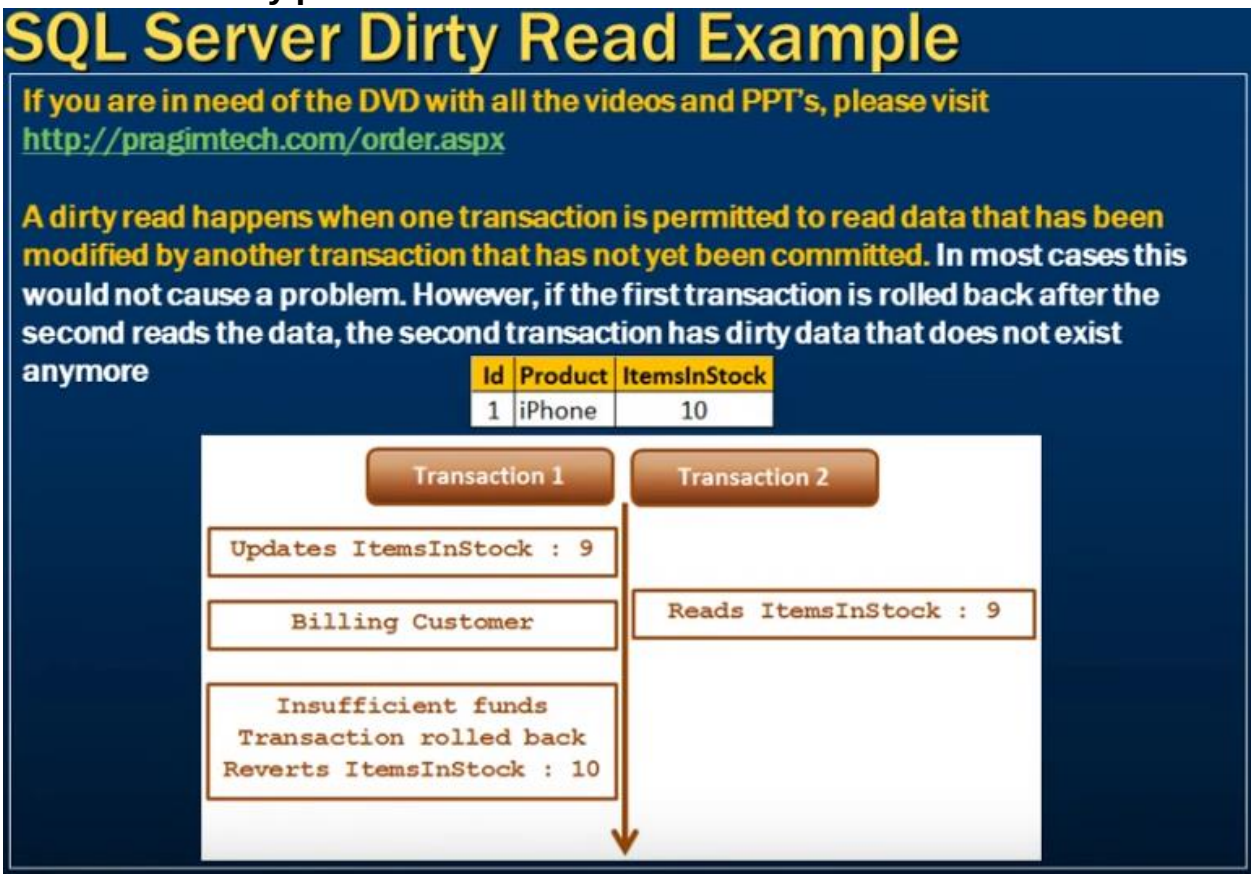
SQL Server Transaction Isolation Levels

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

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

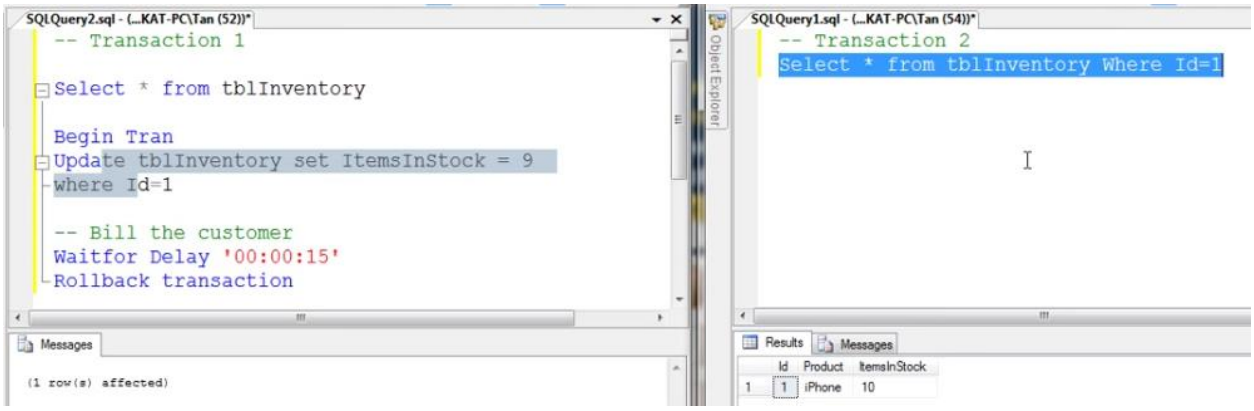
Upcoming Videos : Concurrency problems in detail with examples

Dirty read concurrency problem



DEAFULT Transaction Isolation Level IS READ COMMITTED.

TRANSACTION 2 WILL NOT HAPPEN UNTIL TRANSACTION 1 IS COMMITTED.



HOWEVER, WE CAN CHANGE THAT TRANSACTION ISOLATION LEVEL TO READ UNCOMITTED DATA ALSO TO READ THE DIRTY DATA.

BELOW TRANSACTION ISOLATION LEVEL IS SET TO: READ UNCOMMITTED DATA:

```
-- Transaction 1

Select * from tblInventory

Begin Tran
Update tblInventory set ItemsInStock = 9
where Id=1

-- Bill the customer
Waitfor Delay '00:00:15'
Rollback transaction
```

Results | Messages

Id	Product	ItemsInStock
1	iPhone	10

```
-- Transaction 2
Set transaction isolation level read uncommitted
Select * from tblInventory Where Id=1
```

Results | Messages

Id	Product	ItemsInStock
1	iPhone	9

SQL Server Dirty Read Example

Id	Product	ItemsInStock
1	iPhone	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
```

The other way to read uncommitted data is to use NOLOCK.

```
-- Transaction 1

Select * from tblInventory

Begin Tran
Update tblInventory set ItemsInStock = 9
where Id=1

-- Bill the customer
Waitfor Delay '00:00:15'
Rollback transaction
```

Messages

(1 row(s) affected)

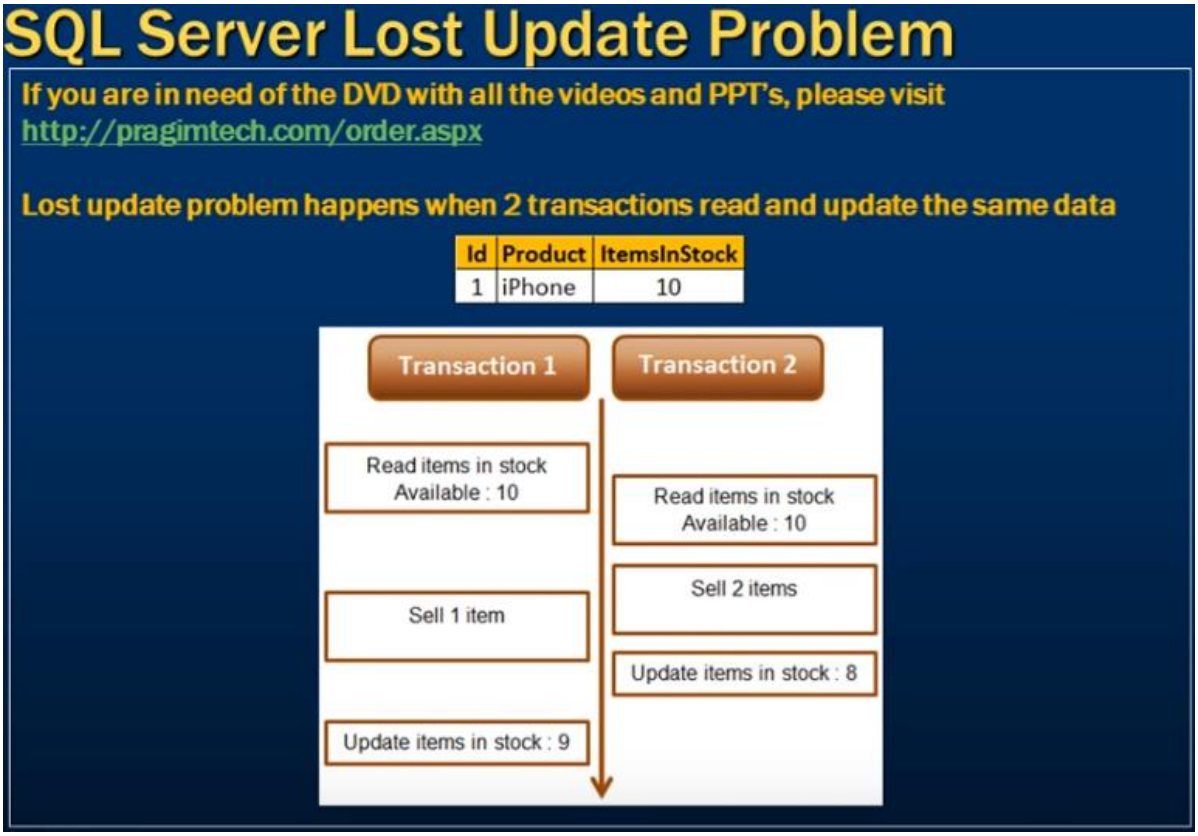
```
-- Transaction 2
Set transaction isolation level read committed
Select * from tblInventory (NOLOCK) Where Id=1
```

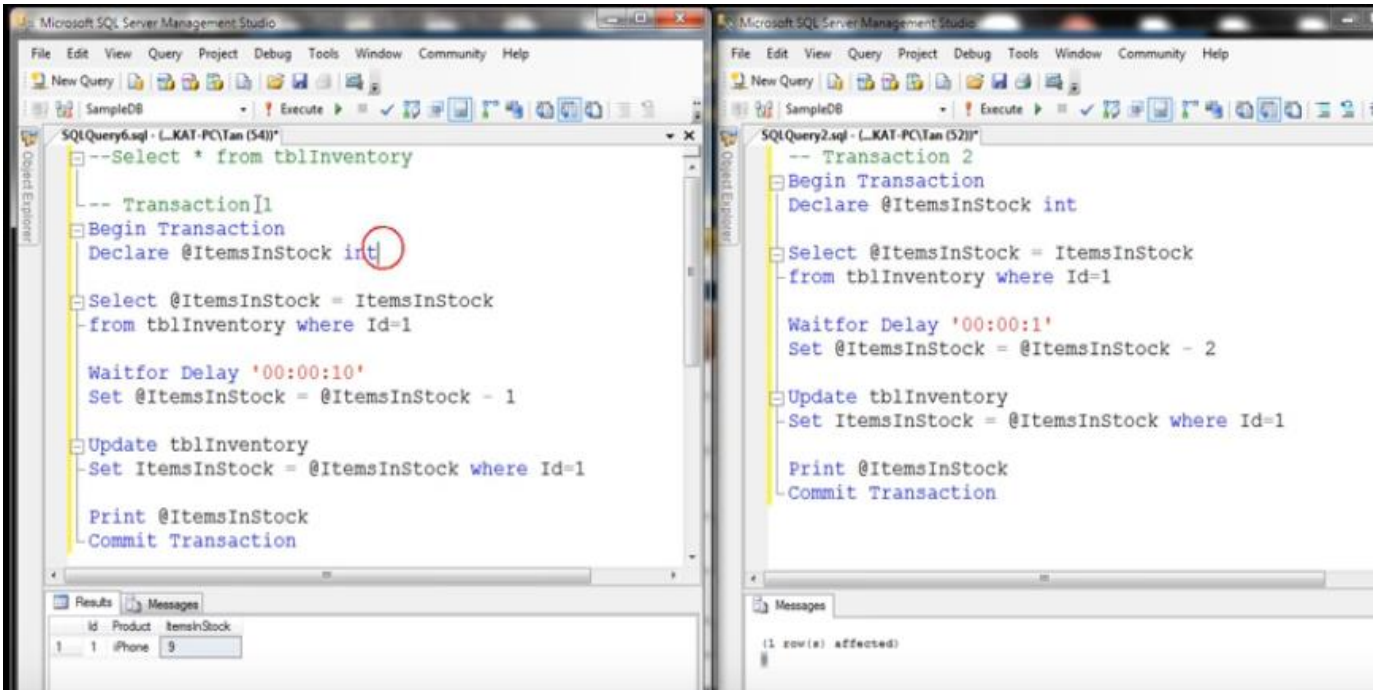
Results | Messages

Id	Product	ItemsInStock
1	iPhone	9

Lost update problem

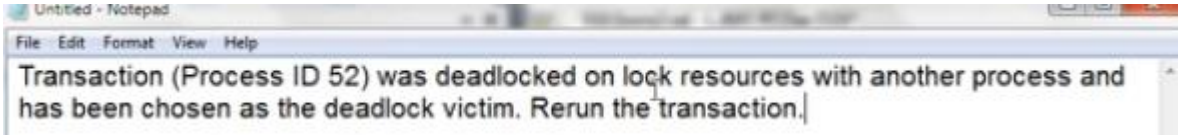
LOST UPDATE PROBLEM when trying to update items in one transaction at the same another transaction is updating to the same table. It seems like transaction updated to the table but in reality it did not as transaction 1 is not yet committed.

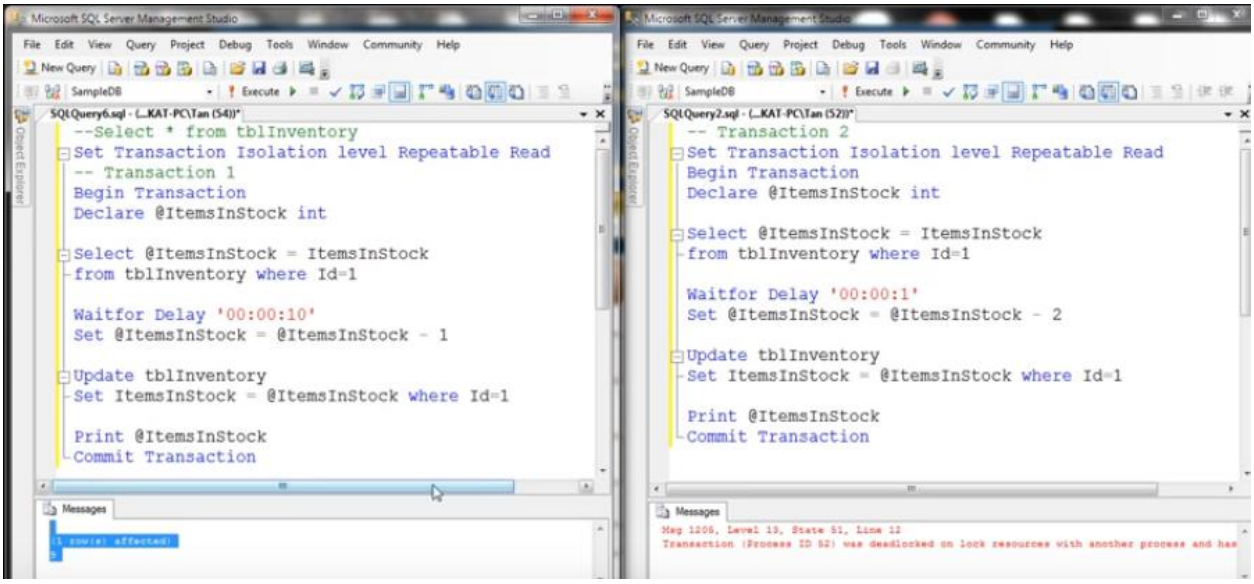


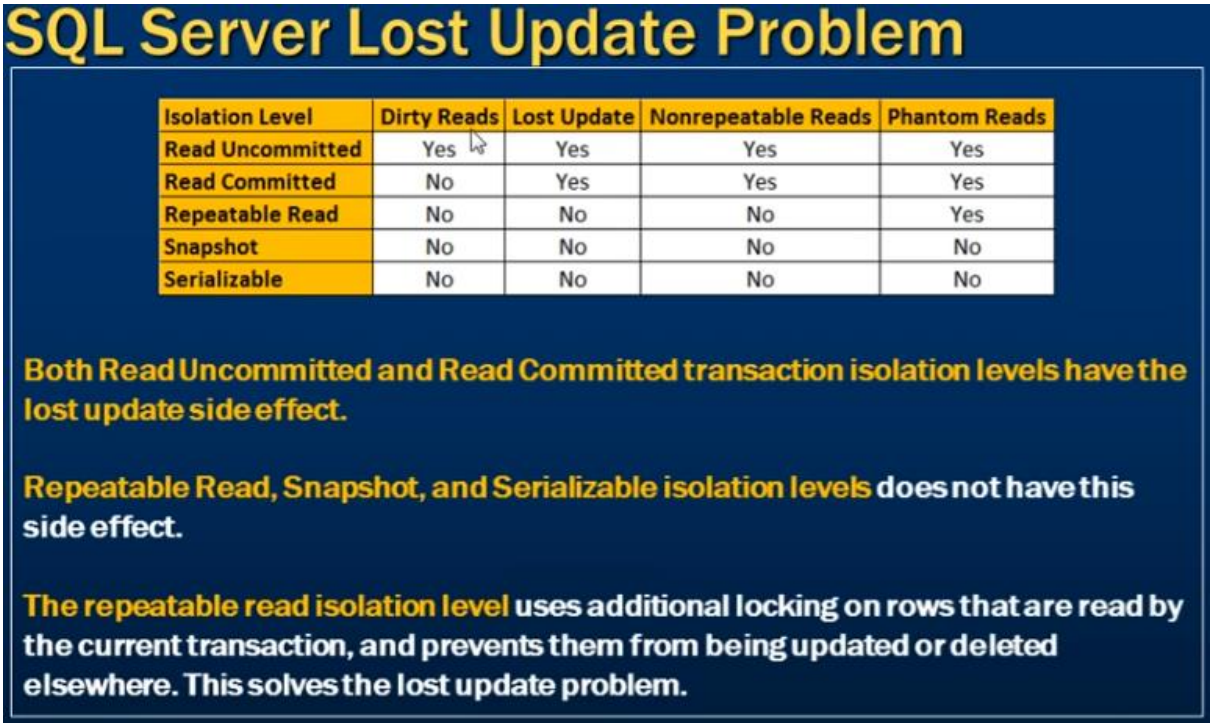


Waitfor Delay makes the next step delayed by that many seconds

Repeatable Read transaction level ensures that the data that Transaction 1 has already read, transaction 2 should not be able to make any modifications on that data. Therefore, only transaction 1 is able to commit. Transaction after 1 second is not able to change the data as transaction is still going on. It gives the following error:

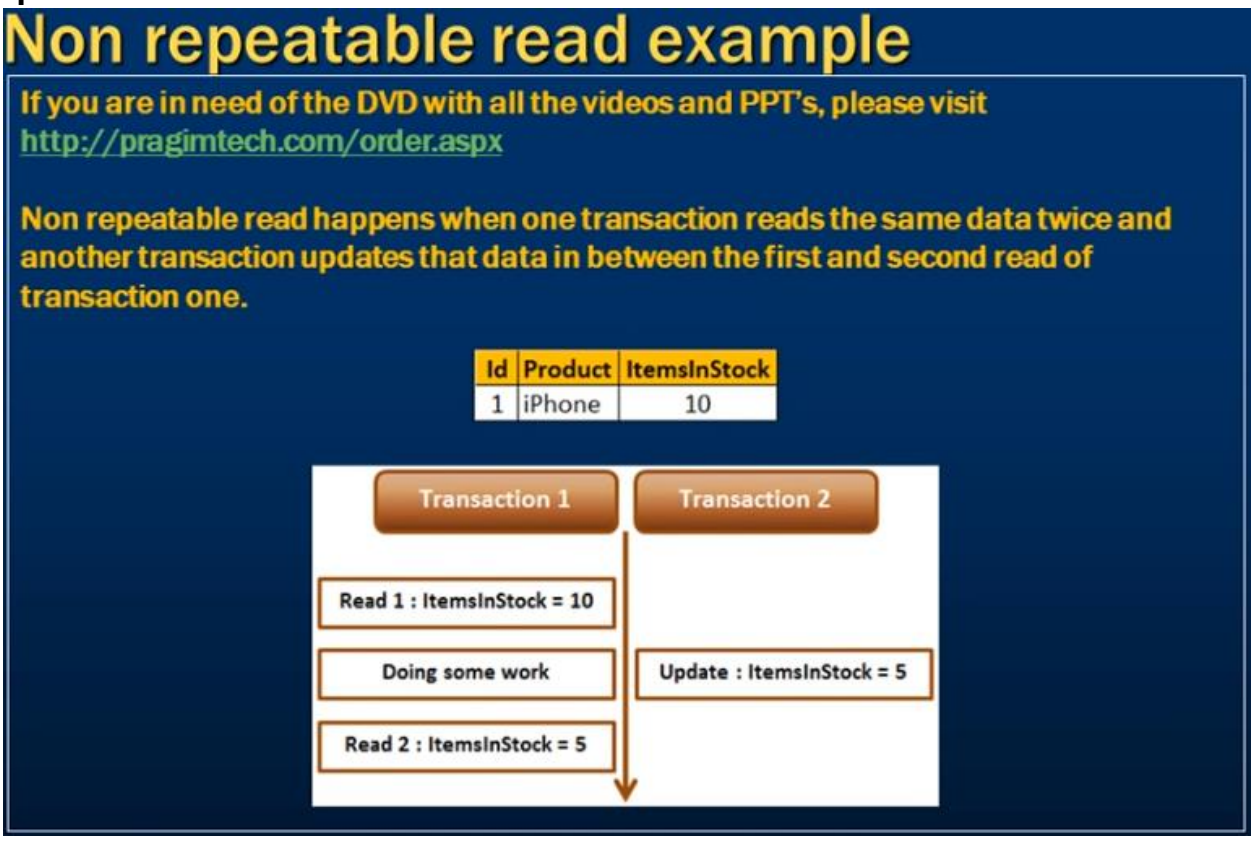




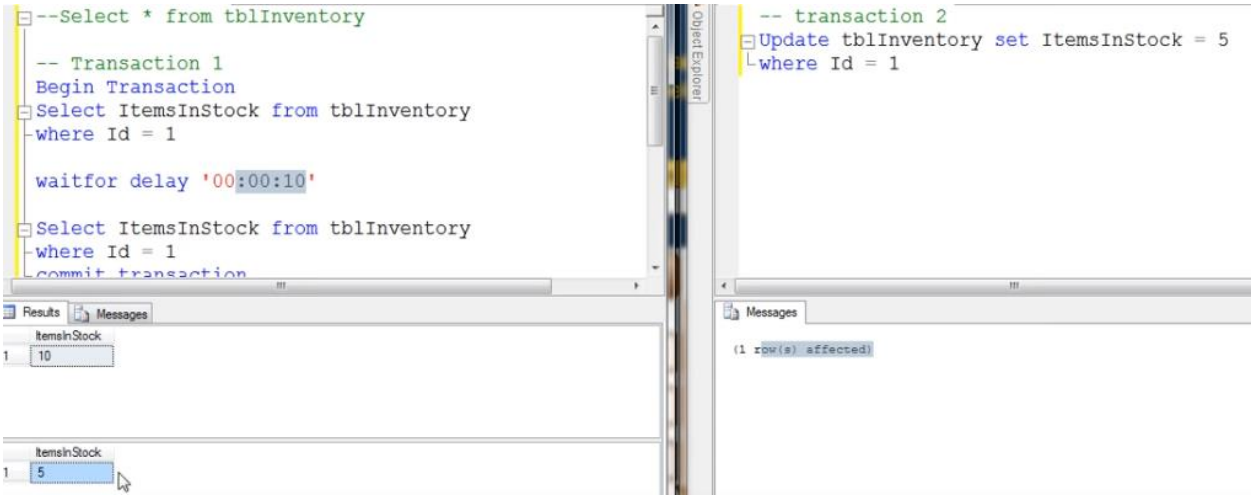


Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

Non repeatable read



WITHOUT REPEATABLE READ TRANSACTION LEVEL



Non repeatable read example

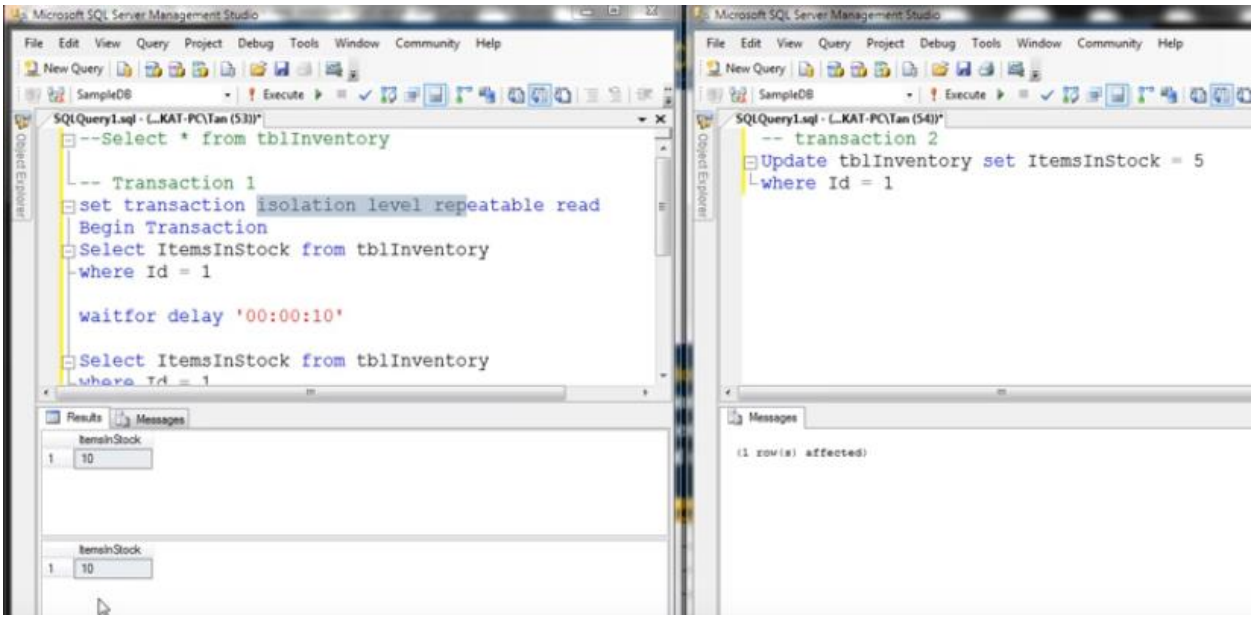
```
-- Transaction 1
Begin Transaction
Select ItemsInStock
from tblInventory where Id = 1

-- 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
```

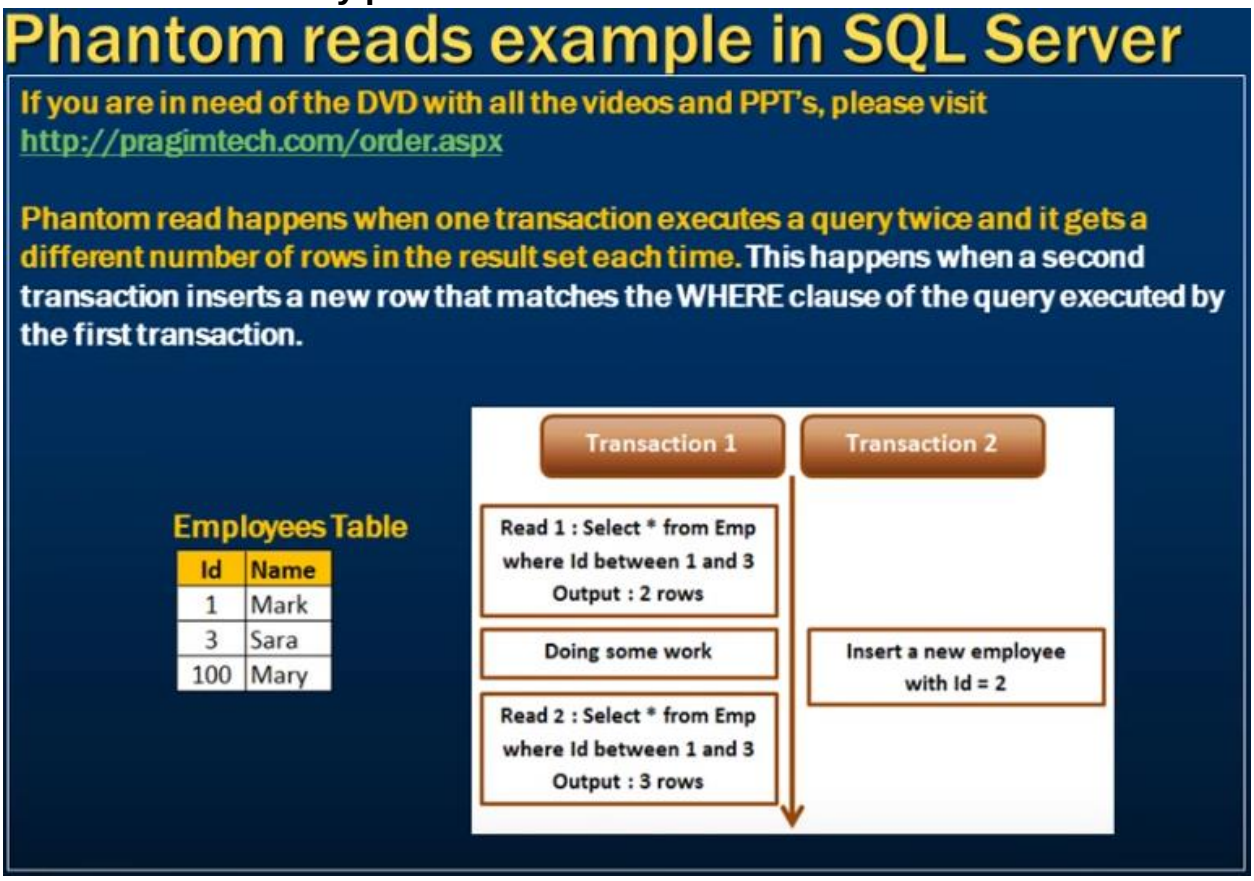
Repeatable read or any other higher isolation level should solve the non-repeatable read problem. To fix the non-repeatable read problem, set transaction isolation level of Transaction 1 to repeatable read. This will ensure that the data that Transaction 1 has read, will be prevented from being updated or deleted elsewhere. This solves the non-repeatable read problem

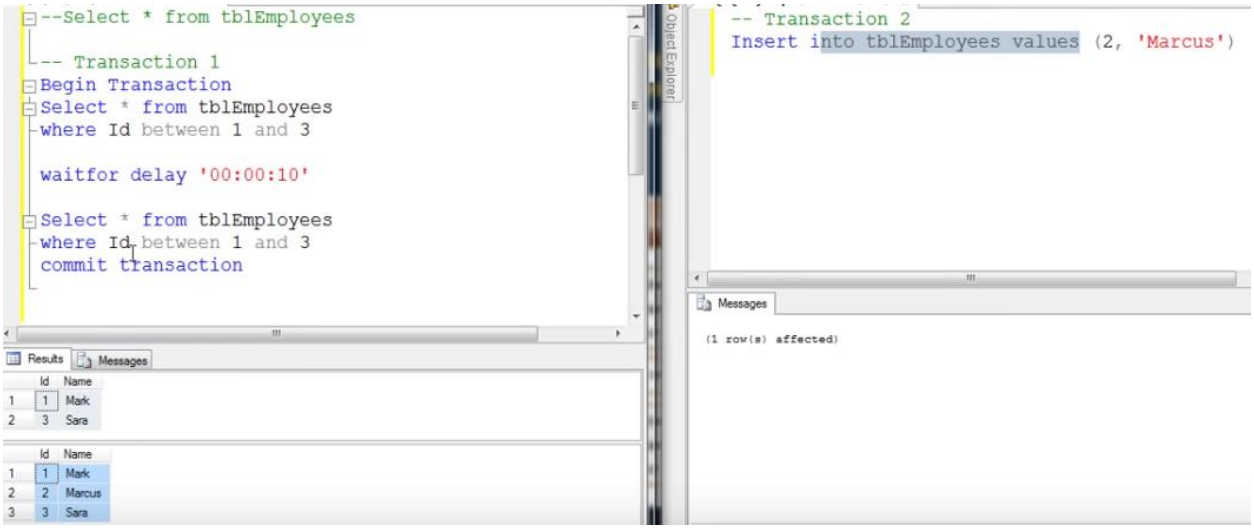
Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

To rectify the problem with non-repeatable read, we use transaction level Repeatable read:



Phantom read concurrency problem





To fix the Phantom Read problem we can either use Snapshot or Serializable Isolation Level.

Phantom reads example in SQL Server

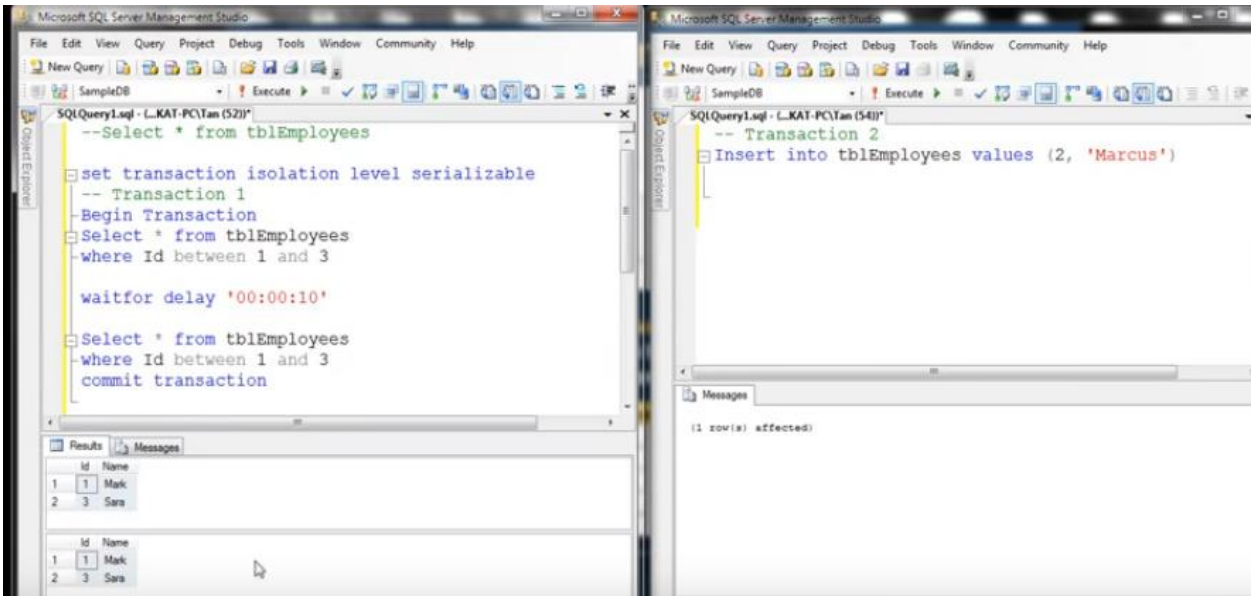
```
-- Transaction 1
Begin Transaction

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')
```

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.

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

SET TRANSACTION ISOLATION LEVEL TO SERIALIZABLE: BELOW TRANSACTION 2 WILL BE ALLOWED TO CONTINUE ONLY AFTER TRANSACTION 1 HAS COMPLETED.



Repeatable Read v/s 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.

Snapshot isolation level

Snapshot Isolation Level in SQL Server

If you are in need of the DVD with all the videos and PPT's, please visit http://pragimtech.com/order.aspx

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

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.

Snapshot Isolation Level in SQL Server

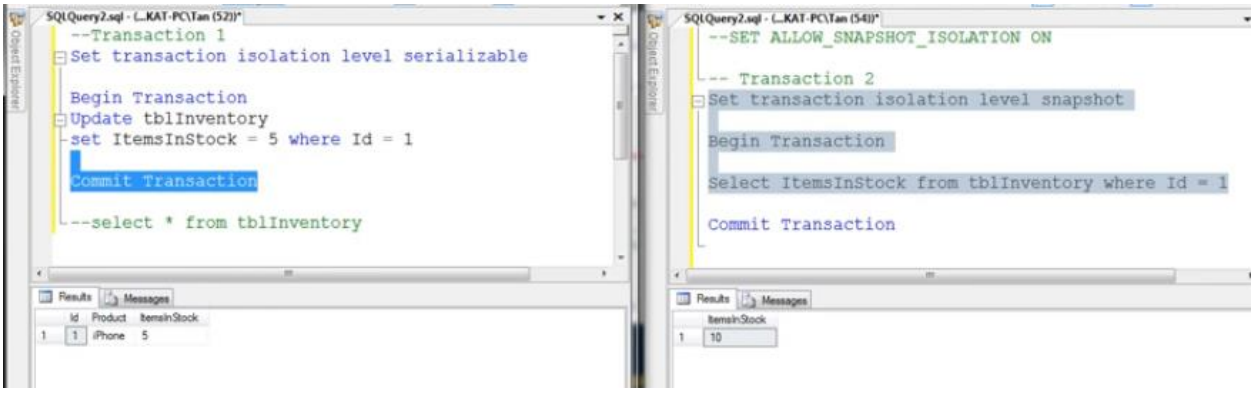
Example : Reading data with Snapshot Isolation level

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

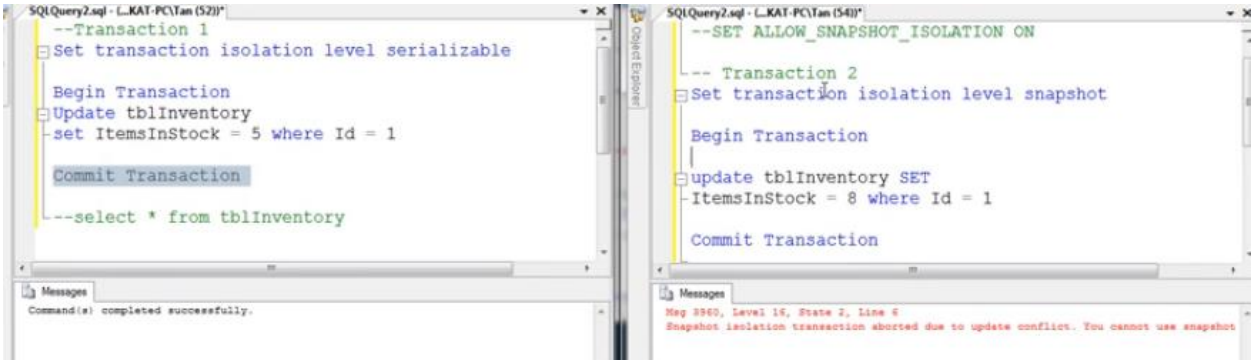
```
-- Transaction 2
-- Enable snapshot isloation for the database
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
-- Set the transaction isolation level to snapshot
Set transaction isolation level snapshot

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

UNLESS WE COMMIT THE TRANSACTION, THE COPY/ VERSION OF THE DATA IN THE TABLE GETS STORED IN THE TEMP DB AS A COPY WHEN WE USE SNAPSHOT. THEREFORE, DATA IS NOT LOCKED FOR THE SECOND TRANSACTION AND WE CAN STILL READ IT FROM THE SECOND TRANSACTION USING THE COPY OF THE TABLE.



HOWEVER WE CAN NOT UPDATE IN THE TABLE USING SNAPSHOT.



Msg 3960, Level 16, State 2, Line 6
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.