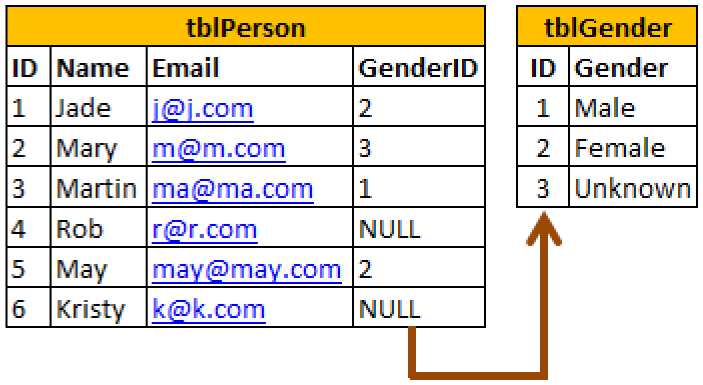
Creating and Working with tables - Part 3

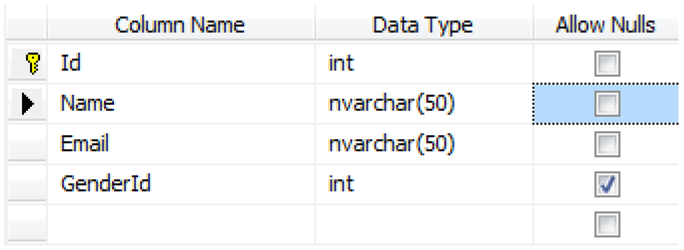
The aim of this article is to create tblPerson and tblGender tables and establish primary key and foreign key constraints. In SQL Server, tables can be created graphically using SQL Server Management Studio (SSMS) or using a query.



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

tblGender	
ID	Gender
1	Male
2	Female
3	Unknown

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



Column Name	Data Type	Allow Nulls
Id	int	☐
Name	nvarchar(50)	☐
Email	nvarchar(50)	☐
GenderId	int	✓
		☐

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

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

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

To graphically add a foreign key reference
1. Right click tblPerson table and select Design

2. In the table design window, right click on GenderId column and select Relationships
3. In the Foreign Key Relationships window, click Add button

4. Now expand, in Tables and Column Specification row, by clicking the, + sign

5. Click on the elipses button, that is present in Tables and Column Specification row
6. From the Primary Key Table, dropdownlist, select tblGender
7. Click on the row below, and select ID column
8. From the column on the right hand side, select GenderId
9. Click OK and then click close.
10. Finally save the table.

To add a foreign key reference using a query

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

The general formula is here

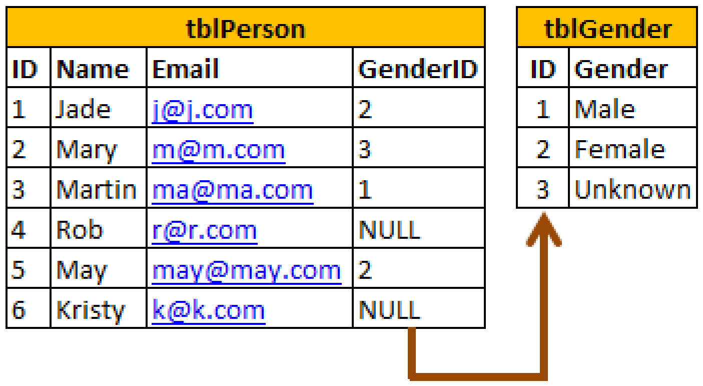
```
Alter table ForeignKeyTable add constraint ForeignKeyTable_ForiegnKeyColumn_FK
FOREIGN KEY (ForiegnKeyColumn) references PrimaryKeyTable (PrimaryKeyColumn)
```

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

Email This

Default constraint in sql server - Part 4

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



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

tblGender	
ID	Gender
1	Male
2	Female
3	Unknown

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

Altering an existing column to add a default constraint:

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

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

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

The following command will add a default constraint, DF_tblPerson_GenderId.

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

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

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

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

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

To drop a constraint

```
ALTER TABLE { TABLE_NAME }
DROP CONSTRAINT { CONSTRAINT_NAME }
```

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

Email This

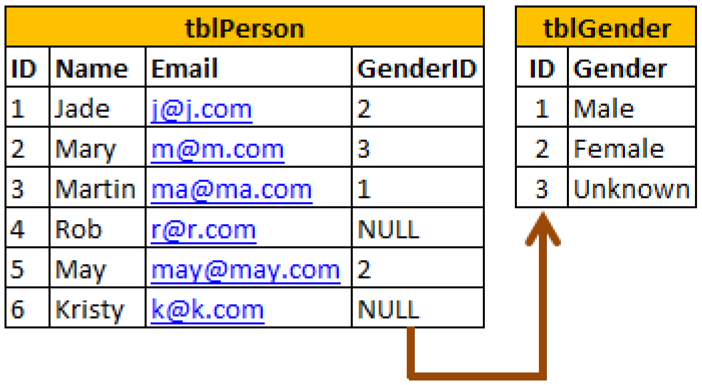
Cascading referential integrity constraint - Part 5

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

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

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

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



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

tblGender	
ID	**Gender**
1	Male
2	Female
3	Unknown

However, you have the following options when setting up Cascading referential integrity constraint

1. No Action: This is the default behaviour. No Action specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, an error is raised and the DELETE or UPDATE is rolled back.

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

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

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

Email This

Check constraint in SQL Server - Part 6

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

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

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

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

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

The general formula for adding check constraint in SQL Server:

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

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

To drop the CHECK constraint:

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

Email This

Identity column in SQL Server - Part 7

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

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

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

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

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

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

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

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

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

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

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

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

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

Example queries for getting the last generated identity value
Select SCOPE_IDENTITY()
Select @@IDENTITY
Select IDENT_CURRENT('tblPerson')

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

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

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

Email This

Unique key constraint - Part 9

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

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

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

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

To drop the constraint
1. Right click the constraint and delete.
Or
2. Using a query

```
Alter Table tblPerson
Drop COnstraint UQ_tblPerson_Email
```

Email This

Select statement - Part 10

Basic select statement syntax

```
SELECT Column_List
FROM Table_Name
```

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

```
SELECT *
FROM Table_Name
```

To Select distinct rows use DISTINCT keyword

```
SELECT DISTINCT Column_List
FROM Table_Name
```

Example: `Select distinct city from tblPerson`

Filtering rows with WHERE clause

```
SELECT Column_List
FROM Table_Name
WHERE Filter_Condition
```

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

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

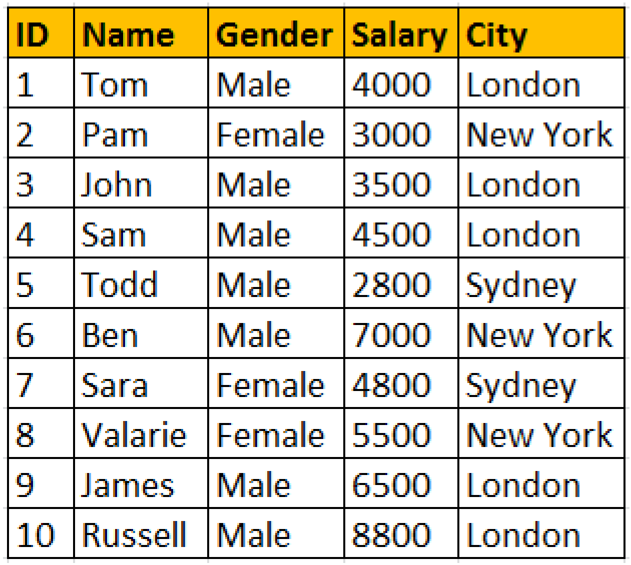
Different operators that can be used in a where clause

Email This

Group By - Part 11

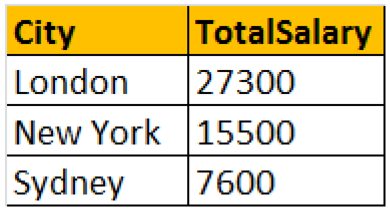
In SQL Server we have got lot of aggregate functions. Examples
1. Count()
2. Sum()
3. avg()
4. Min()
5. Max()

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



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

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



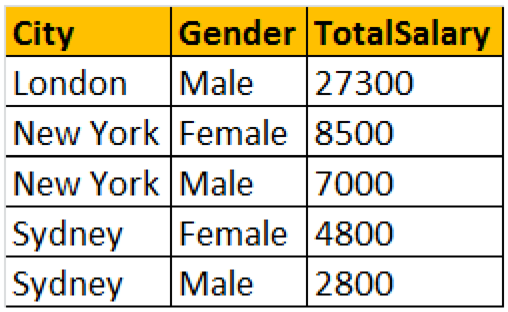
City	TotalSalary
London	27300
New York	15500
Sydney	7600

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

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

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

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

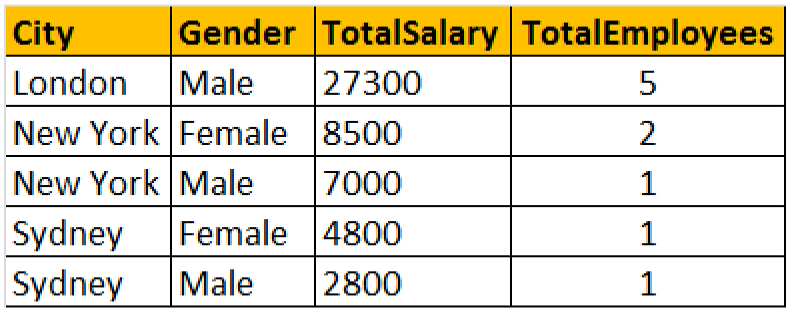


City	Gender	TotalSalary
London	Male	27300
New York	Female	8500
New York	Male	7000
Sydney	Female	4800
Sydney	Male	2800

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

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

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



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

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

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

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

Filtering rows using WHERE clause, before aggrgations take place:

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

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

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

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

It is also possible to combine WHERE and HAVING

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

Difference between WHERE and HAVING clause:
1. WHERE clause can be used with - Select, Insert, and Update statements, where as HAVING clause can only be used with the Select statement.
2. WHERE filters rows before aggregation (GROUPING), where as, HAVING filters groups, after the aggregations are performed.

3. Aggregate functions cannot be used in the WHERE clause, unless it is in a sub query contained in a HAVING clause, whereas, aggregate functions can be used in Having clause.
Email This

Joins in sql server - Part 12

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

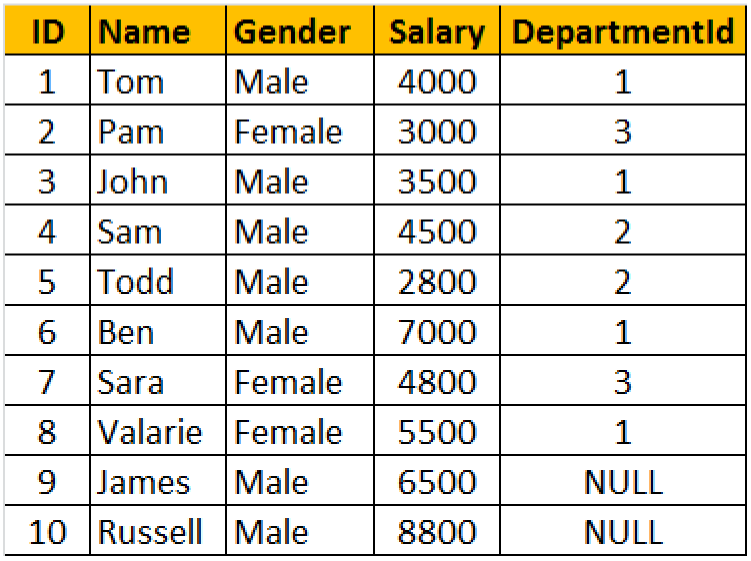
Please watch Parts 3 and 5 in this video series, before continuing with this video.
Part 3 - Creating and working with tables
Part 5 - Cascading referential integrity constraint

In SQL server, there are different types of JOINS.
1. CROSS JOIN

2. INNER JOIN
3. OUTER JOIN

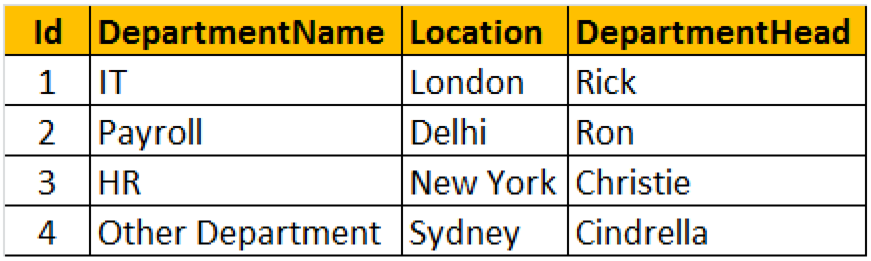
Outer Joins are again divided into 3 types
1. Left Join or Left Outer Join
2. Right Join or Right Outer Join
3. Full Join or Full Outer Join

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



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

Departments Table (tblDepartment)



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

SQL Script to create tblEmployee and tblDepartment tables

```
Create table tblDepartment

(

    ID int primary key,

    DepartmentName nvarchar(50),

    Location nvarchar(50),
    DepartmentHead nvarchar(50)
)
Go

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

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

Insert into tblEmployee values (1, 'Tom', 'Male', 4000, 1)
Insert into tblEmployee values (2, 'Pam', 'Female', 3000, 3)
Insert into tblEmployee values (3, 'John', 'Male', 3500, 1)
Insert into tblEmployee values (4, 'Sam', 'Male', 4500, 2)
Insert into tblEmployee values (5, 'Todd', 'Male', 2800, 2)
Insert into tblEmployee values (6, 'Ben', 'Male', 7000, 1)
Insert into tblEmployee values (7, 'Sara', 'Female', 4800, 3)

Insert into tblEmployee values (8, 'Valarie', 'Female', 5500, 1)

Insert into tblEmployee values (9, 'James', 'Male', 6500, NULL)

Insert into tblEmployee values (10, 'Russell', 'Male', 8800, NULL)

Go
```

General Formula for Joins

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

CROSS JOIN

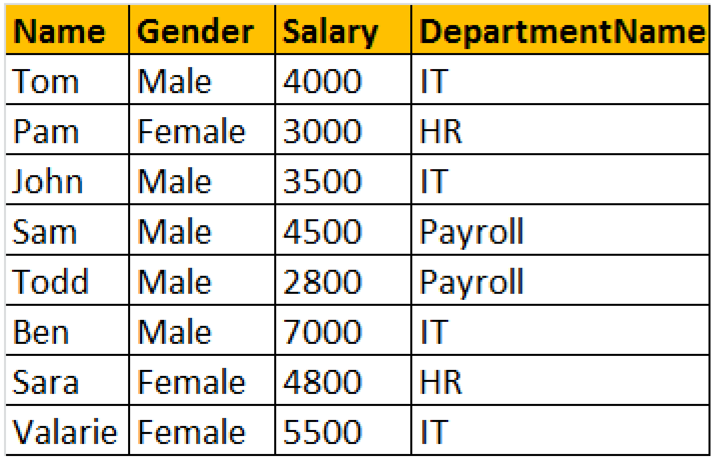
CROSS JOIN, produces the cartesian product of the 2 tables involved in the join. For example, in the Employees table we have 10 rows and in the Departments table we have 4 rows. So, a cross join between these 2 tables produces 40 rows. Cross Join shouldn't have ON clause.

CROSS JOIN Query:

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

JOIN or INNER JOIN

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



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

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

```
ON tblEmployee.DepartmentId = tblDepartment.Id
```

OR

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

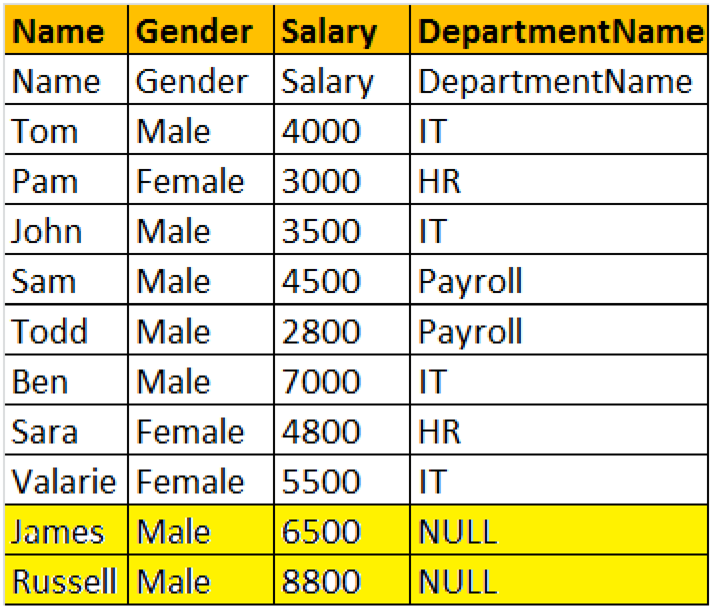
Note: JOIN or INNER JOIN means the same. It's always better to use INNER JOIN, as this explicitly specifies your intention.

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

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

LEFT JOIN or LEFT OUTER JOIN

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



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

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

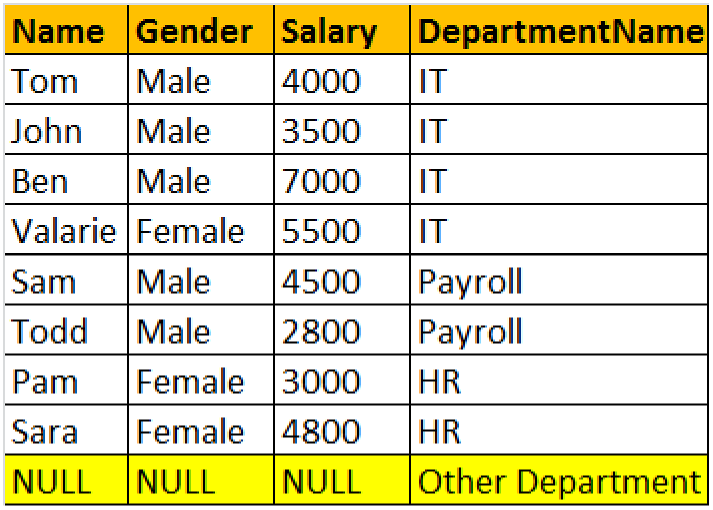
OR

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

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

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

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



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

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

OR

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

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

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

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

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

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

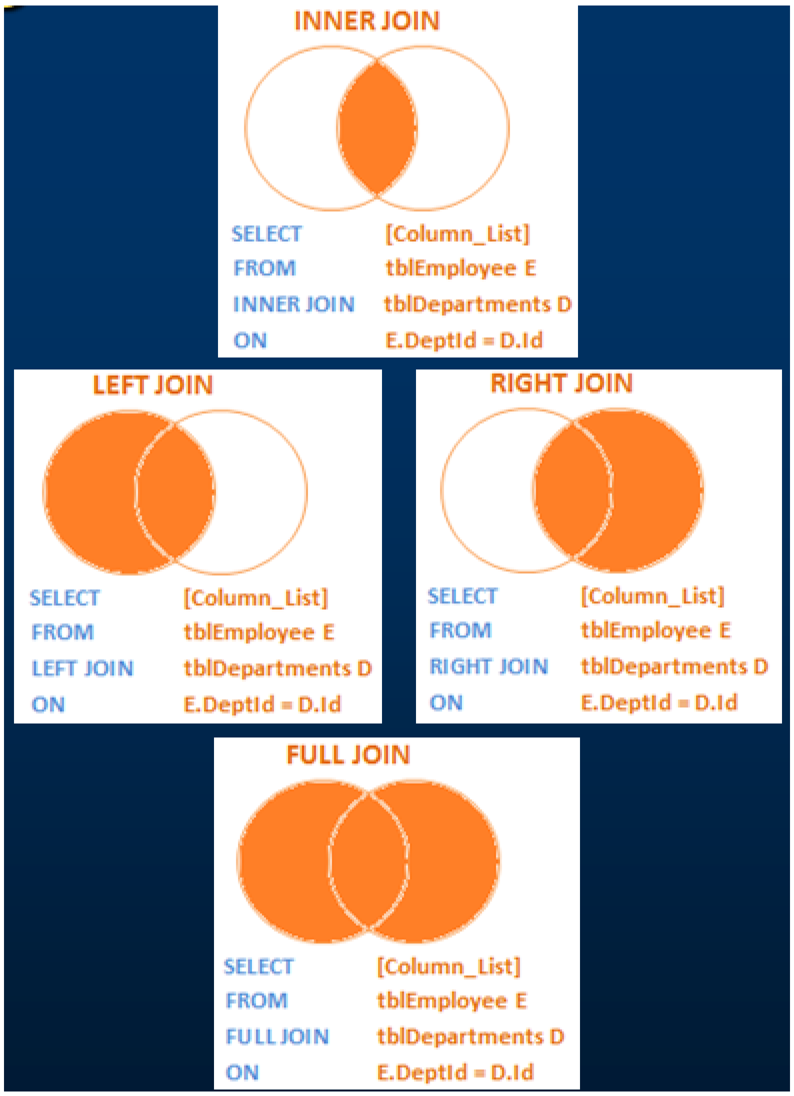
OR

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

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

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

Joins Summary



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

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

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

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

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

Advanced Joins - Part 13

In this video session we will learn about

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

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

Considers Employees (tblEmployee) and Departments (tblDepartment) tables

Employee Table (tblEmployee)

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

Departments Table (tblDepartment)

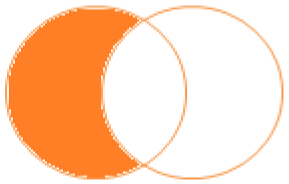
Id	DepartmentName	Location	DepartmentHead
1	IT	London	Rick
2	Payroll	Delhi	Ron
3	HR	New York	Christie
4	Other Department	Sydney	Cindrella

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

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

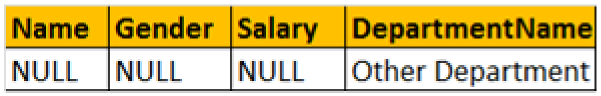
Query:

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



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

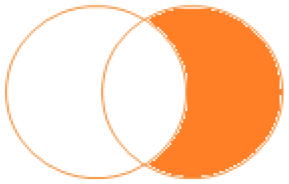
How to retrieve only the non matching rows from the right table



Name	Gender	Salary	DepartmentName
NULL	NULL	NULL	Other Department

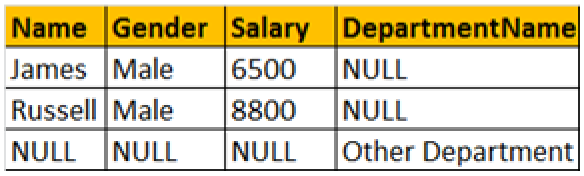
Query:

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



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

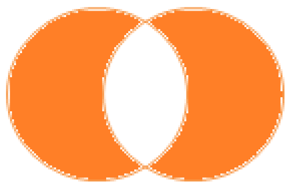
How to retrieve only the non matching rows from both the left and right table. Matching rows should be eliminated.

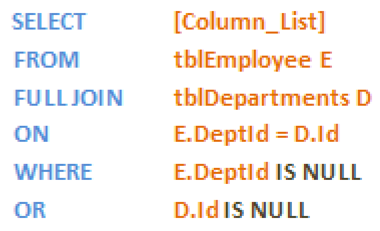


Name	Gender	Salary	DepartmentName
James	Male	6500	NULL
Russell	Male	8800	NULL
NULL	NULL	NULL	Other Department

Query:

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





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

Email This

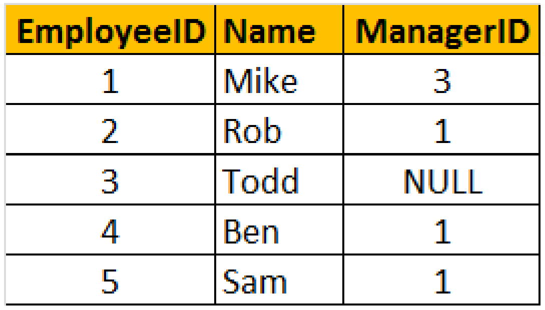
Self join in sql server - Part 14

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

Part 12 - Basic joins

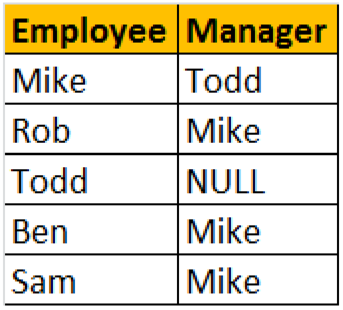
Part 13 - Advanced joins

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



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

Write a query which gives the following result.



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

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

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

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

Inner Self Join tblEmployee table:

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

Cross Self Join tblEmployee table:

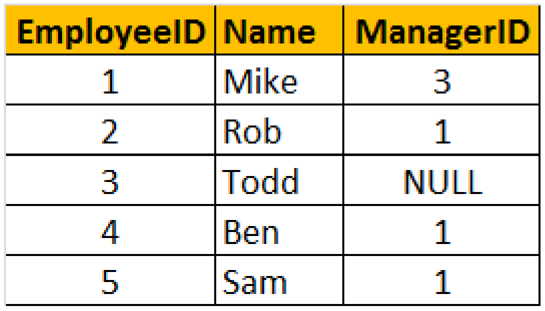
```
Select E.Name as Employee, M.Name as Manager
from tblEmployee
Cross Join tblEmployee
```

Email This

Different ways to replace NULL in sql server - Part 15

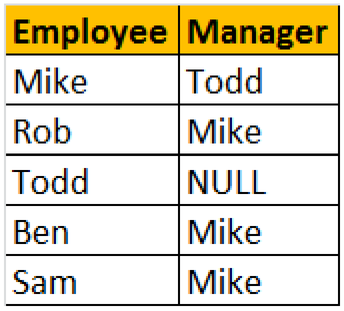
In this video session, we will learn about different ways to replace NULL values in SQL Server. Please watch Part 14, before continuing.

Consider the Employees table below.



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

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



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

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

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

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

Replacing NULL value using CASE Statement:

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

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

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

ON E.ManagerID = M.EmployeeID

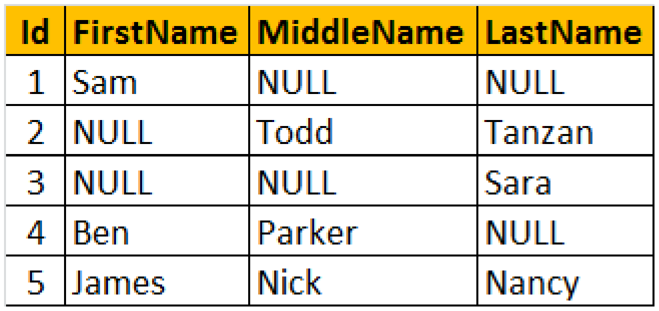
We will discuss about COALESCE() function in detail, in the next session

Email This

Coalesce() function in sql server - Part 16

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

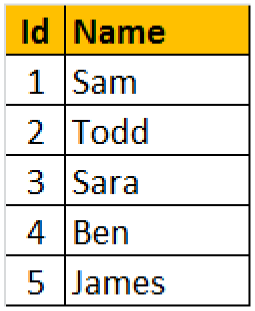
Consider the Employees Table below. Not all employees have their First, Midde and Last Names filled. Some of the employees has First name missing, some of them have Middle Name missing and some of them last name.



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

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

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



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

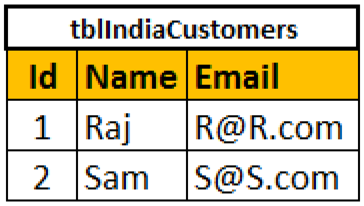
We are passing **FirstName, MiddleName and LastName** columns as parameters to the COALESCE() function. The COALESCE() function returns the first non null value from the 3 columns.

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

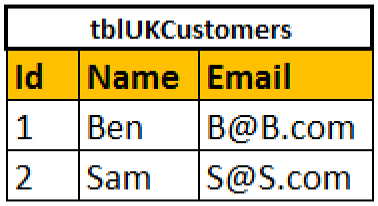
Email This

Union and union all in sql server - Part 17

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



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

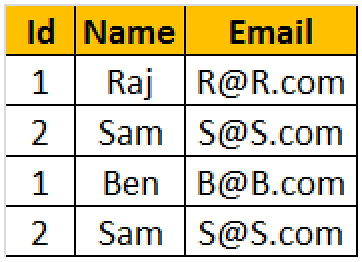


tblUKCustomers		
Id	**Name**	**Email**
1	Ben	B@B.com
2	Sam	S@S.com

Combining the rows of tblIndiaCustomers and tblUKCustomers using UNION ALL

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

Query Results of UNION ALL

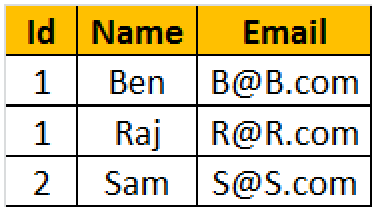


Id	Name	Email
1	Raj	R@R.com
2	Sam	S@S.com
1	Ben	B@B.com
2	Sam	S@S.com

Combining the rows of tblIndiaCustomers and tblUKCustomers using UNION

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

Query Results of UNION



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

Differences between UNION and UNION ALL (Common Interview Question)

From the output, it is very clear that, **UNION removes duplicate** rows, where as **UNION ALL**

does not. When use UNION, to remove the duplicate rows, sql server has to to do a distinct sort, which is time consuming. For this reason, UNION ALL is much faster than UNION.

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

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

If you want to sort, the results of UNION or UNION ALL, the ORDER BY caluse should be used on the last SELECT statement as shown below.

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

The following query, raises a syntax error

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

Difference between JOIN and UNION

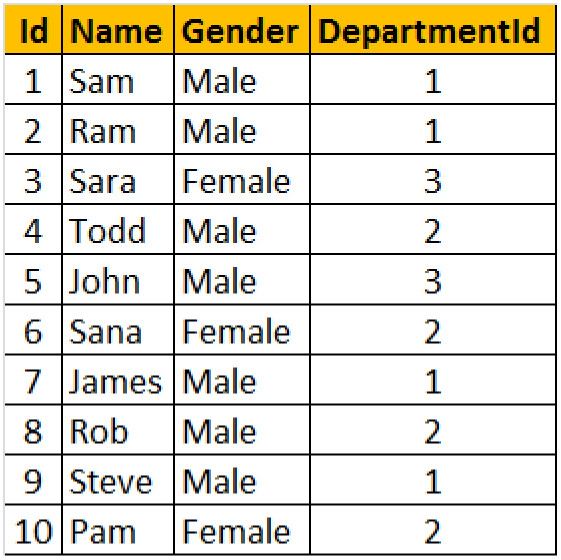
JOINS and **UNIONS** are different things. However, this question is being asked very frequently now. UNION combines the result-set of two or more select queries into a single result-set which includes all the rows from all the queries in the union, where as JOINS, retrieve data from two or more tables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

Email This

Stored procedures - Part 18

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

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



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

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

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

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

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

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

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

```
Create Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
as
Begin
```

```
  Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId =
@DepartmentId
End
```

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

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

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

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

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

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

To change the stored procedure, use ALTER PROCEDURE statement:

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

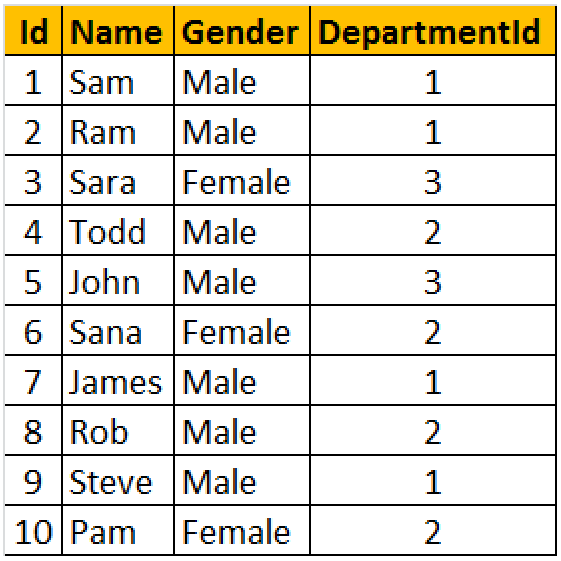
To encrypt the text of the SP, use WITH ENCRYPTION option. Once, encrypted, you cannot view the text of the procedure, using sp_helptext system stored procedure. There are ways to obtain the original text, which we will talk about in a later session.

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

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

Stored procedures with output parameters - Part 19

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



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

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

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

To execute this stored procedure with OUTPUT parameter

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

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

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

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

```
if(@EmployeeTotal is null)
 Print '@EmployeeTotal is null'
else
 Print '@EmployeeTotal is not null'
```

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

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

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

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

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

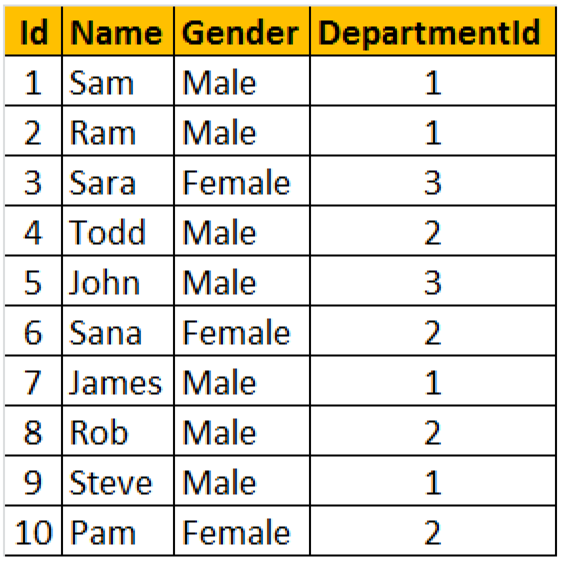
Stored procedure output parameters or return values - Part 20

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

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

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

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



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

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

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

Executing spGetTotalCountOfEmployees1 returns 3.

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

Re-written stored procedure using return variables

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

Executing spGetTotalCountOfEmployees2 returns 3.

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

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

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

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

Executing spGetNameById1, prints the name of the employee

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

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

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

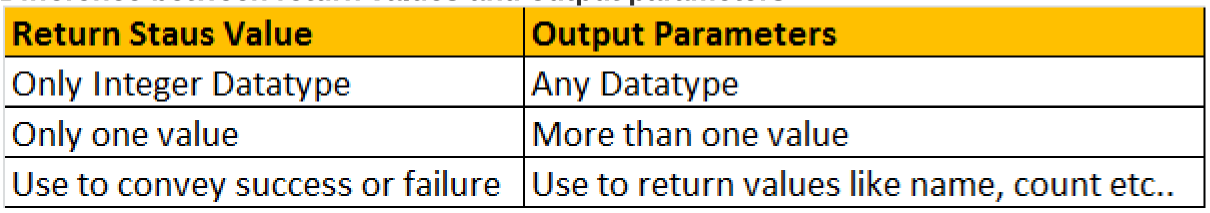
Executing spGetNameById2 returns an error stating 'Conversion failed when converting the nvarchar value 'Sam' to data type int.'. The return status variable is an integer, and hence, when we select Name of an employee and try to return that we get a converion error.

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

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

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

Difference between return values and output parameters



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

Email This

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

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

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

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

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

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

Built in string functions in sql server 2008 - Part 22
Functions in SQL server can be broadly divided into 2 categoris
1. Built-in functions
2. User Defined functions

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

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

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

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

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

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

Printing uppercase alphabets using CHAR() function:

```
Declare @Number int
Set @Number = 65
While(@Number <= 90)
Begin
 Print CHAR(@Number)
 Set @Number = @Number + 1
End
```

Printing lowercase alphabets using CHAR() function:

```
Declare @Number int
Set @Number = 97
While(@Number <= 122)
Begin
 Print CHAR(@Number)
 Set @Number = @Number + 1
End
```

Another way of printing lower case alphabets using CHAR() and LOWER() functions.

```
Declare @Number int
Set @Number = 65
While(@Number <= 90)
Begin
 Print LOWER(CHAR(@Number))
 Set @Number = @Number + 1
End
```

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

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

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

Output: Hello

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

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

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

Output: Hello

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

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

Output: Hello

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

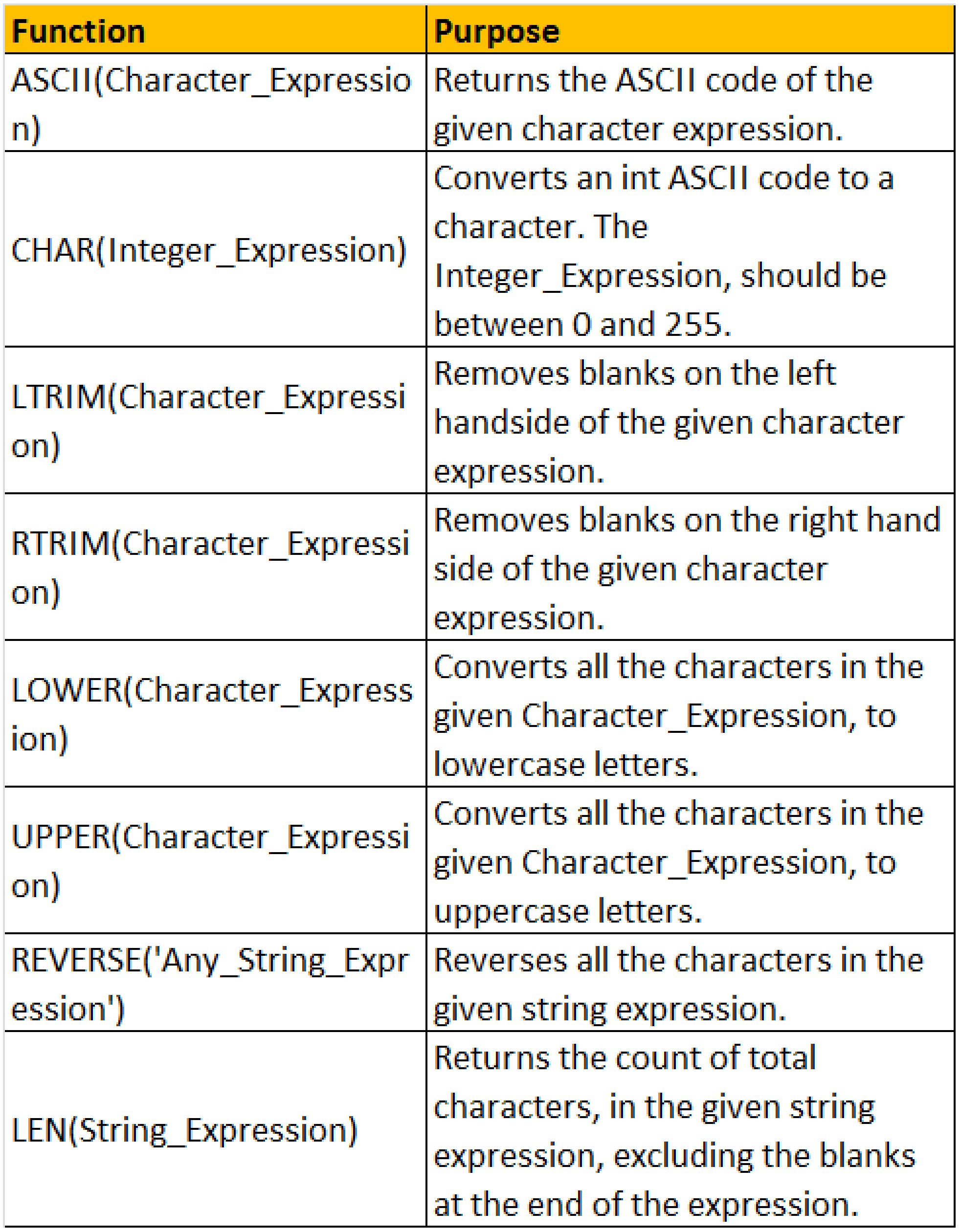
Example: Select LOWER('CONVERT This String Into Lower Case')
Output: convert this string into lower case

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

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

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

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



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

LEFT, RIGHT, CHARINDEX and SUBSTRING functions - Part 23

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

Part 11 – Group By

Part 22 – Built in string functions

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

Example: Select LEFT('ABCDE', 3)
Output: ABC

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

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

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

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

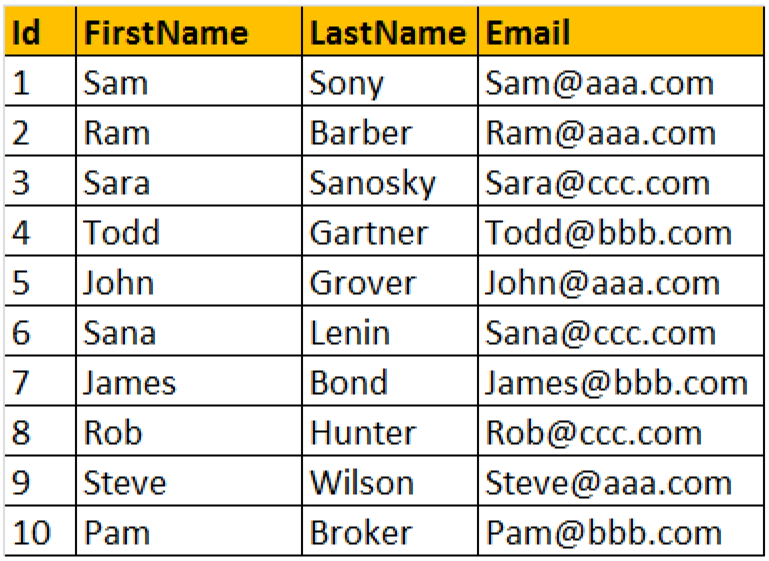
SUBSTRING('Expression', 'Start', 'Length') - As the name, suggests, this function returns substring (part of the string), from the given expression. You specify the starting location using the 'start' parameter and the number of characters in the substring using 'Length' parameter. All the 3 parameters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

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

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

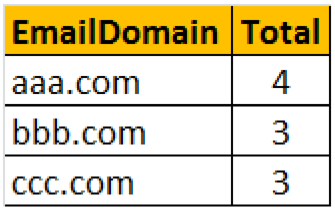
Example:
Select SUBSTRING('John@bbb.com',(CHARINDEX('@', 'John@bbb.com') + 1), (LEN('John@bbb.com') - CHARINDEX('@','John@bbb.com')))
Output: bbb.com

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



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

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



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

Query

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

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

Before watching this video, please watch

Part 22 – Built in string functions in sql server

Part 23 – Left, Right, CharIndex and Substring functions

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

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

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

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

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

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

Query:

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

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

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

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

Output:

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

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

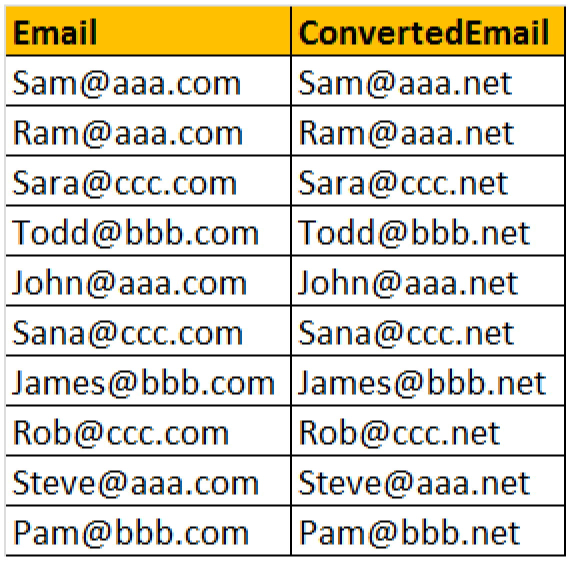
Example:
```
Select Email, PATINDEX('%@aaa.com', Email) as FirstOccurence
from tblEmployee
Where PATINDEX('%@aaa.com', Email) > 0
```

Output:

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

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

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

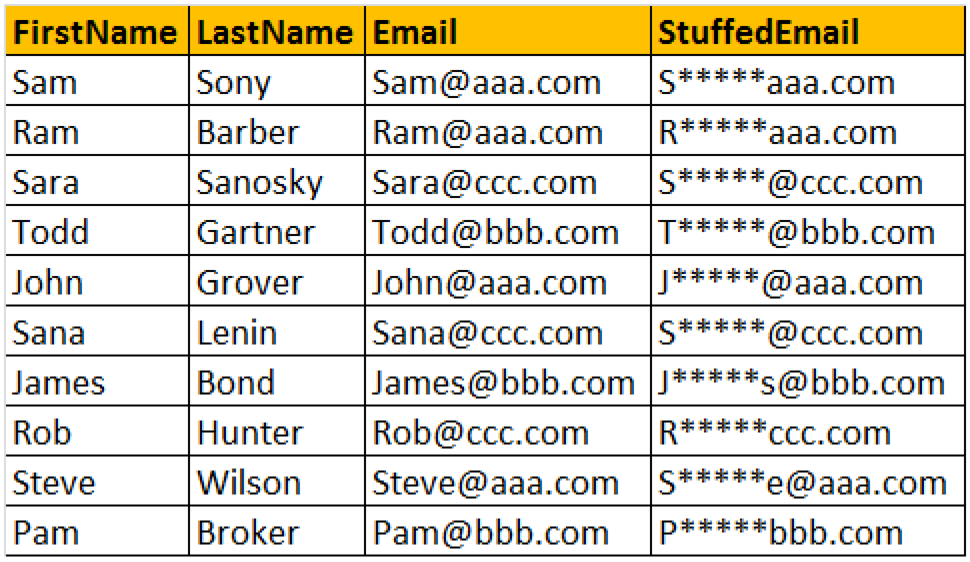


Email	ConvertedEmail
Sam@aaa.com	Sam@aaa.net
Ram@aaa.com	Ram@aaa.net
Sara@ccc.com	Sara@ccc.net
Todd@bbb.com	Todd@bbb.net
John@aaa.com	John@aaa.net
Sana@ccc.com	Sana@ccc.net
James@bbb.com	James@bbb.net
Rob@ccc.com	Rob@ccc.net
Steve@aaa.com	Steve@aaa.net
Pam@bbb.com	Pam@bbb.net

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

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

Output:



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

DateTime functions in SQL Server - Part 25

In this video session we will learn about
1. DateTime data types

2. DateTime functions available to select the current system date and time
3. Understanding concepts - UTC time and Time Zone offset
There are several built-in DateTime functions available in SQL Server. All the following functions can be used to get the current system date and time, where you have sql server installed.

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

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

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

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

To Insert some sample data, execute the following query.

```
INSERT
INTO tblDateTime VALUES (GETDATE(),GETDATE(),GETDATE(),GETDATE(),GETDATE(),G
ETDATE())
```

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

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

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

Examples:

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

Note: For datetime2 values, IsDate returns ZERO.

Example:
Select ISDATE('2012-09-01 11:34:21.1918447') -- returns 0.
Day() - Returns the **'Day number of the Month'** of the given date

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

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

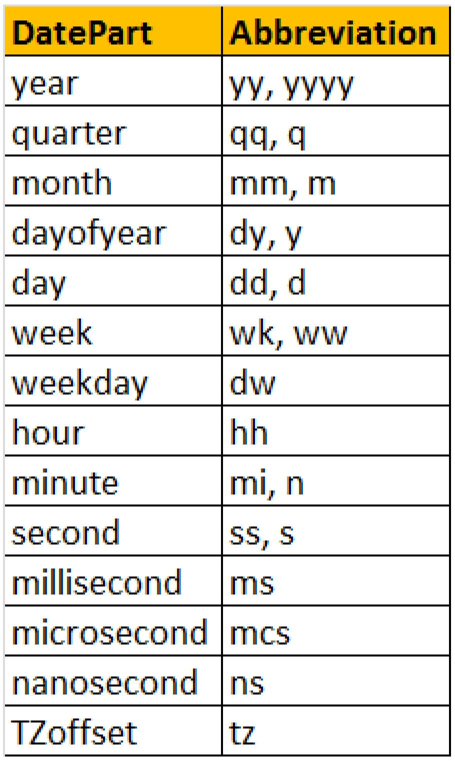
Examples:
Select Month(GETDATE()) -- Returns the **Month number of the year**, based on the current system date and time
Select Month('01/31/2012') -- Returns 1

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

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

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

Valid Datepart parameter values

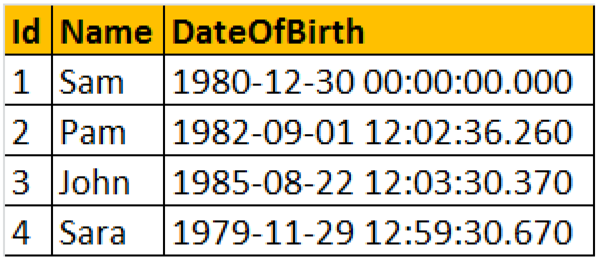


DatePart	Abbreviation
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns
TZoffset	tz

Examples:

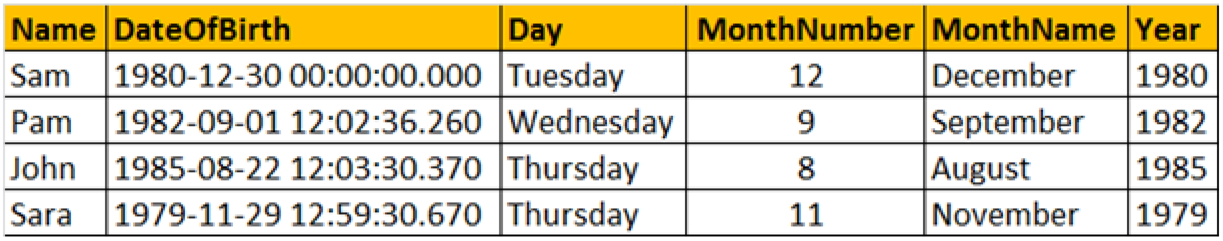
```
Select DATENAME(Day, '2012-09-30 12:43:46.837') -- Returns 30
Select DATENAME(WEEKDAY, '2012-09-30 12:43:46.837') -- Returns Sunday
Select DATENAME(MONTH, '2012-09-30 12:43:46.837') -- Returns September
```

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



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

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



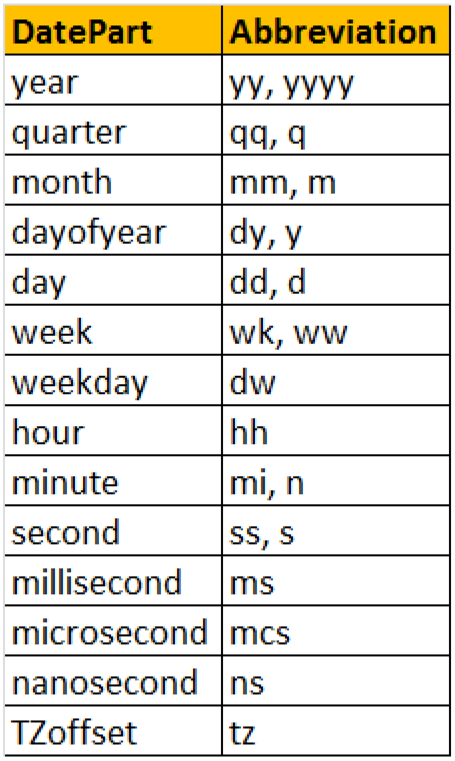
Name	DateOfBirth	Day	MonthNumber	MonthName	Year
Sam	1980-12-30 00:00:00.000	Tuesday	12	December	1980
Pam	1982-09-01 12:02:36.260	Wednesday	9	September	1982
John	1985-08-22 12:03:30.370	Thursday	8	August	1985
Sara	1979-11-29 12:59:30.670	Thursday	11	November	1979

Query:

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

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

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



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

Examples:

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

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

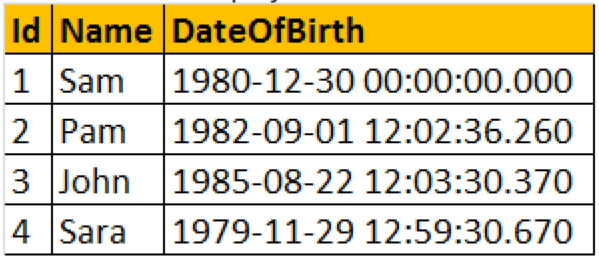
Examples:

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

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

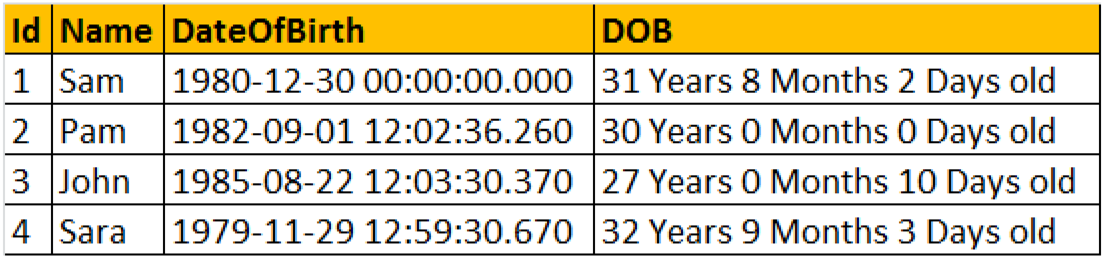
Examples:
```
Select DATEDIFF(MONTH, '11/30/2005','01/31/2006') -- returns 2
Select DATEDIFF(DAY, '11/30/2005','01/31/2006') -- returns 62
```

Consider the emaployees table below.



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

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



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

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

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

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

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

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

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

```
End
```

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

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

Email This

Cast and Convert functions in SQL Server - Part 28

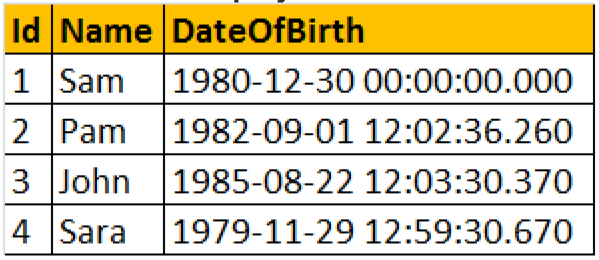
To convert one data type to another, CAST and CONVERT functions can be used.

Syntax of CAST and CONVERT functions from MSDN:

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

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

Consider the Employees Table below

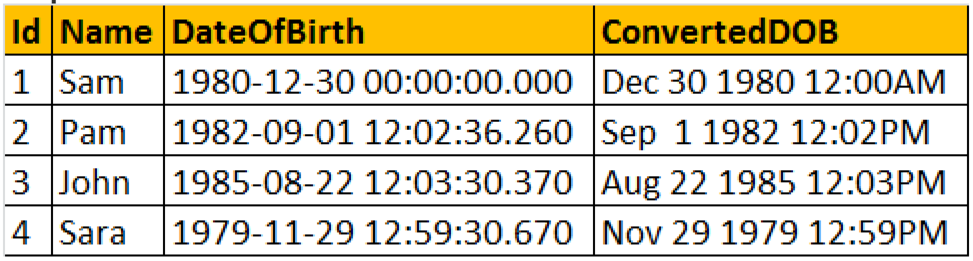


Id	Name	DateOfBirth
1	Sam	1980-12-30 00:00:00.000
2	Pam	1982-09-01 12:02:36.260
3	John	1985-08-22 12:03:30.370
4	Sara	1979-11-29 12:59:30.670

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

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

Output:



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

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

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

Output:

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

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

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

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

To get just the date part, from DateTime

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

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

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

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

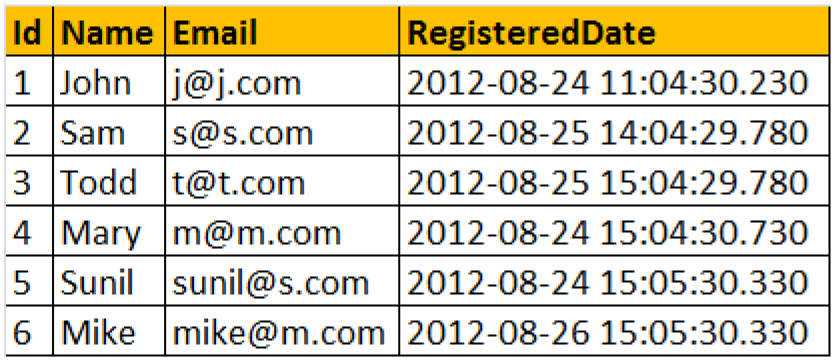
Now, let's write a query which produces the following output:

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

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

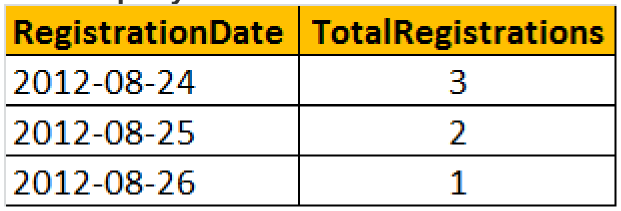
```
Select Id, Name, Name + ' - ' + CAST(Id AS NVARCHAR) AS [Name-Id]
FROM tblEmployees
```

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



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

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



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

Query:

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

The following are the differences between the 2 functions.

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

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

Mathematical functions in sql server - Part 29

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

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

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

CEILING (numeric_expression) and FLOOR (numeric_expression)
CEILING and **FLOOR** functions accept a numeric expression as a single parameter. CEILING() returns the smallest integer value greater than or equal to the parameter, whereas FLOOR() returns the largest integer less than or equal to the parameter.

Examples:

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

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

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

Example: The following example calculates '2 TO THE POWER OF 3' = 2*2*2 = 8

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

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

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

Example:

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

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

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

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

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

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

Example:

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

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

Example:

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

ROUND (numeric_expression , length [,function]) - Rounds the given numeric expression based on the given length. This function takes 3 parameters.

1. Numeric_Expression is the number that we want to round.

2. Length parameter, specifies the number of the digits that we want to round to. If the length is

a positive number, then the rounding is applied for the decimal part, where as if the length is negative, then the rounding is applied to the number before the decimal.
3. The optional function parameter, is used to indicate rounding or truncation operations. A value of 0, indicates rounding, where as a value of non zero indicates truncation. Default, if not specified is 0.

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

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

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

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

-- Round the last 2 places before (to the left) the decimal point
Select ROUND(850.556, -2) -- 900.000

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

Scalar User Defined Functions in sql server - Part 30

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

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

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

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

To create a function, we use the following syntax:

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

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

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

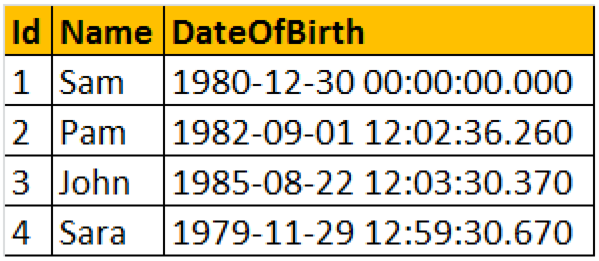
When calling a scalar user-defined function, you must supply a two-part name, **OwnerName.FunctionName**. **dbo** stands for database owner.

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

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

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

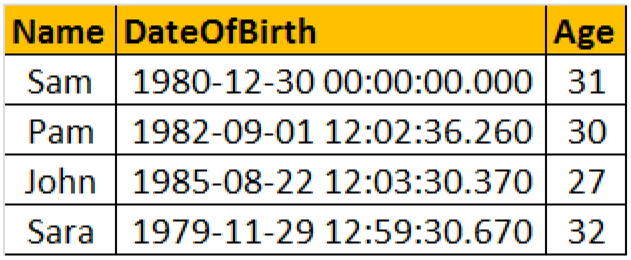
Consider the Employees table below.



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

Scalar user defined functions can be used in the Select clause as shown below.

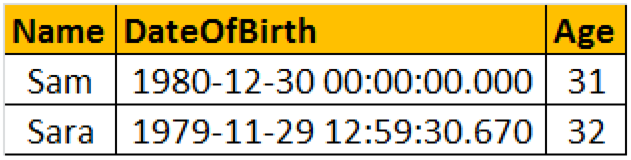
```
Select Name, DateOfBirth, dbo.Age(DateOfBirth) as Age from tblEmployees
```



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

Scalar user defined functions can be used in the Where clause, as shown below.

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



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

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

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

To view the text of the function use sp_helptext FunctionName

Inline table valued functions - Part 31

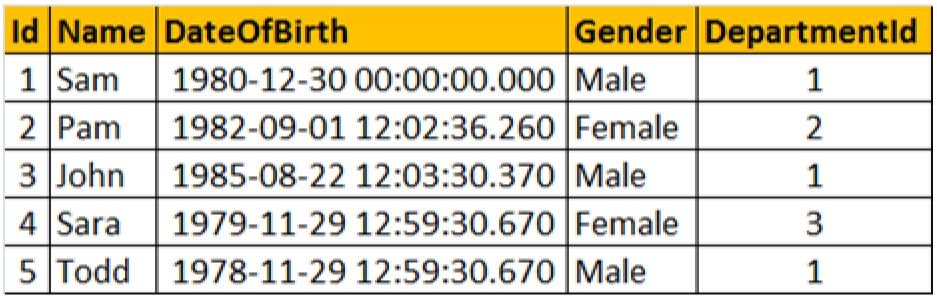
In Part 30 of this video series we have seen how to create and call '**scalar user defined functions**'. In this part of the video series, we will learn about '**Inline Table Valued Functions**'.

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

Syntax for creating an inline table valued function

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

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



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

Create a function that returns EMPLOYEES by GENDER.

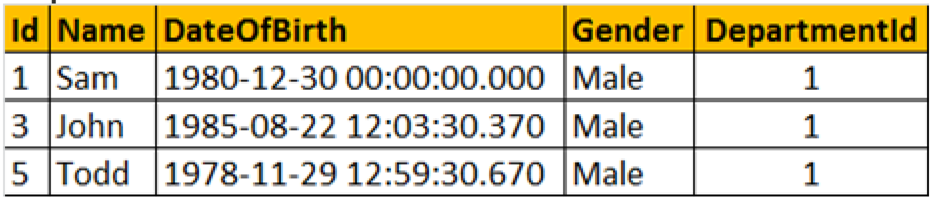
```
CREATE FUNCTION fn_EmployeesByGender(@Gender nvarchar(10))
RETURNS TABLE
AS
RETURN (Select Id, Name, DateOfBirth, Gender, DepartmentId
    from tblEmployees
    where Gender = @Gender)
```

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

Calling the user defined function

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

Output:



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

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

Where can we use Inline Table Valued functions
1. Inline Table Valued functions can be used to achieve the functionality of parameterized views. We will talk about views, in a later session.
2. The table returned by the table valued function, can also be used in joins with other tables.

Consider the Departments Table

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

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

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

Executing the above query should produce the following output

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

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

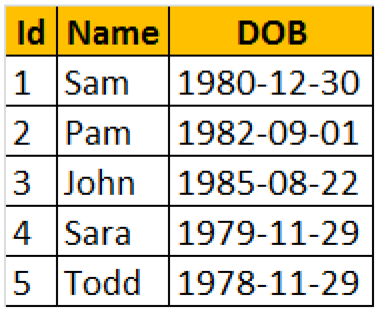
Multi-Statement Table Valued Functions in SQL Server - Part 32

We have discussed about **scalar functions in Part 29** and **Inline Table Valued functions in Part 30**. In this video session, we will discuss about Multi-Statement Table Valued functions. Multi statement table valued functions are very similar to Inline Table valued functions, with a few differences. Let's look at an example, and then note the differences.

Employees Table:

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

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



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

Inline Table Valued function(ILTVF):

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

Multi-statement Table Valued function(MSTVF):

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

 Return
End
```

Calling the Inline Table Valued Function:

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

Calling the Multi-statement Table Valued Function:

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

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

1. In an Inline Table Valued function, the RETURNS clause cannot contain the structure of the table, the function returns. Where as, with the multi-statement table valued function, we specify the structure of the table that gets returned

2. Inline Table Valued function cannot have BEGIN and END block, where as the multi-statement function can have.
3. Inline Table valued functions are better for performance, than multi-statement table valued functions. If the given task, can be achieved using an inline table valued function, always prefer to use them, over multi-statement table valued functions.
4. It's possible to update the underlying table, using an inline table valued function, but not possible using multi-statement table valued function.

Updating the underlying table using inline table valued function:
This query will change **Sam** to **Sam1**, in the underlying table **tblEmployees**. When you try do the same thing with the multi-statement table valued function, you will get an error stating 'Object 'fn_MSTVF_GetEmployees' cannot be modified.'

```
Update fn_ILTVF_GetEmployees() set Name='Sam1' Where Id = 1
```

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

Important concepts related to Functions in sql server - Part 33

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

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

Note: All aggregate functions are deterministic functions.

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

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

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

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

Insert rows into the table using the insert script below.

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

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

Scalar Function without encryption option:
```
Create Function fn_GetEmployeeNameById(@Id int)
Returns nvarchar(20)
as
Begin
 Return (Select Name from tblEmployees Where Id = @Id)
End
```

To view text of the function:
```
sp_helptex fn_GetEmployeeNameById
```

Now, let's alter the function to use WITH ENCRYPTION OPTION
```
Alter Function fn_GetEmployeeNameById(@Id int)
Returns nvarchar(20)
With Encryption
as
Begin
 Return (Select Name from tblEmployees Where Id = @Id)
End
```

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

Creating a function WITH SCHEMABINDING option:
1. **The function fn_GetEmployeeNameById**(), is dependent on tblEmployees table.
2. Delete the table **tblEmployees** from the database.
```
Drop Table tblEmployees
```
3. Now, execute the function fn_GetEmployeeNameById(), you will get an error stating 'Invalid object name tblEmployees'. So, we are able to delete the table, while the function is still refrencing it.
4. Now, **recreate the table** and insert data, using the scripts provided.

5. Next, **Alter the function fn_GetEmployeeNameById()**, to use WITH SCHEMABINDING option.
```
Alter Function fn_GetEmployeeNameById(@Id int)
Returns nvarchar(20)
With SchemaBinding
as
Begin
 Return (Select Name from dbo.tblEmployees Where Id = @Id)
End
```

Note: You have to use the **2 part object name** i.e, dbo.tblEmployees, to use WITH SCHEMABINDING option. dbo is the schema name or owner name, tblEmployees is the table

name.
6. Now, **try to drop the table using** - Drop Table tblEmployees. You will get a message stating, 'Cannot DROP TABLE tblEmployees because it is being referenced by object fn_GetEmployeeNameById.'

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

Temporary tables in SQL Server - Part 34

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

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

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

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

Insert Data into the temporary table:

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

Select the data from the temporary table:

```
Select * from #PersonDetails
```

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

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

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

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

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

```
DROP TABLE #PersonDetails
```

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

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

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

Select * from #PersonDetails
End
```

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

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

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

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

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

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

2. SQL Server appends some random numbers at the end of the local temp table name, where

this is not done for global temp table names.

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

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

Indexes in sql server - Part 35

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

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

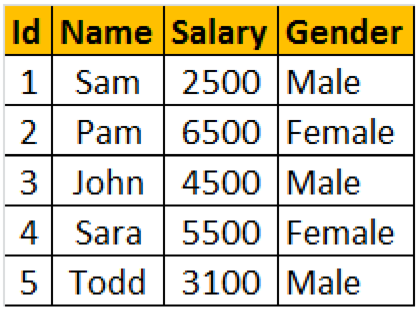
On, the other hand, if you have the index, you lookup the page number of the chapter in the index, and then directly go to that page number to locate the chapter.

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

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

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

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



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

Consider, the following query

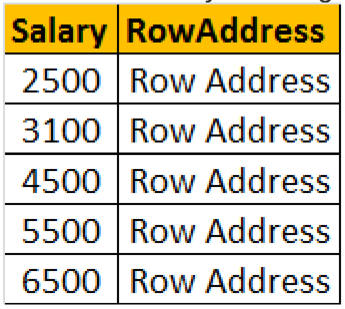
```
Select * from tblEmployee where Salary > 5000 and Salary < 7000
```

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

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

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

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



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

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

An Index can also be created graphically using SQL Server Management Studio

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

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

```
Execute sp_helptext tblEmployee
```

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

```
Drop Index tblEmployee.IX_tblEmployee_Salary
```

Email This

Clustered and Non-Clustered indexes - Part 36

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

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

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

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

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

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

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

Now execute the following insert queries. Note that, the values for Id column are not in a sequential order.
```
Insert into tblEmployee Values(3,'John',4500,'Male','New York')
Insert into tblEmployee Values(1,'Sam',2500,'Male','London')

Insert into tblEmployee Values(4,'Sara',5500,'Female','Tokyo')
Insert into tblEmployee Values(5,'Todd',3100,'Male','Toronto')
Insert into tblEmployee Values(2,'Pam',6500,'Female','Sydney')
```

Execute the following SELECT query
```
Select * from tblEmployee
```

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

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

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

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

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

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

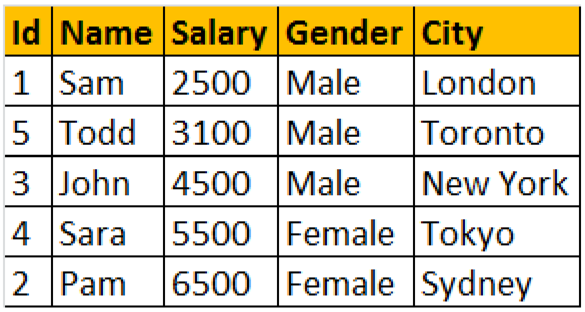
```
Drop index tblEmployee.PK__tblEmplo__3214EC070A9D95DB
```

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

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

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

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



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

Non Clustered Index:

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

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

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

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

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

Unique and Non-Unique Indexes - Part 37

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

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

Create the Employee table using the script below

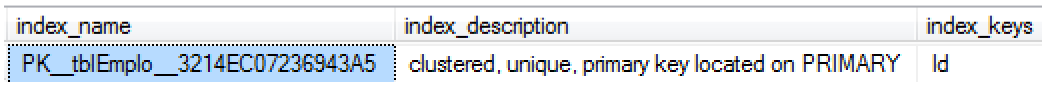
```
CREATE TABLE [tblEmployee]
(
 [Id] int Primary Key,
 [FirstName] nvarchar(50),
 [LastName] nvarchar(50),
 [Salary] int,
 [Gender] nvarchar(10),
 [City] nvarchar(50)
)
```

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

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

```
Execute sp_helpindex tblEmployee
```

Output:



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

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

Example: The following insert queries will fail

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

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

```
Drop index tblEmployee.PK__tblEmplo__3214EC07236943A5
```

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

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

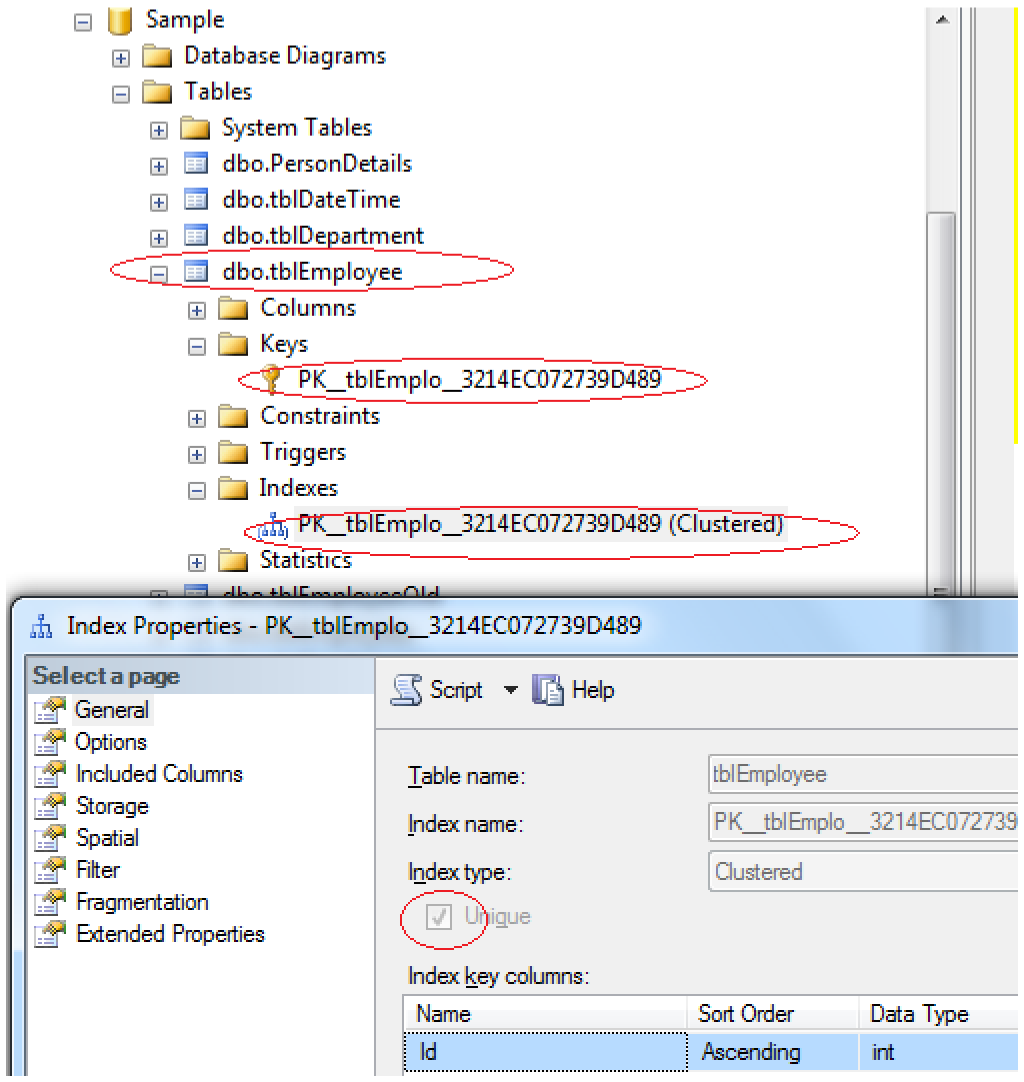


Table name: tblEmployee

Index name: PK__tblEmplo__3214EC072739

Index type: Clustered

[✓ Unique]

Index key columns:

Name	Sort Order	Data Type
Id	Ascending	int

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

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

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

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

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

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

```
Create Unique NonClustered Index UIX_tblEmployee_FirstName_LastName
```

```
On tblEmployee(FirstName, LastName)
```

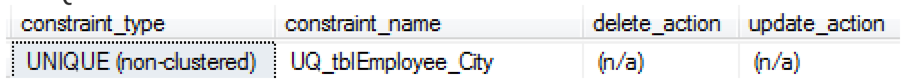
This unique non clustered index, ensures that no 2 entires in the index has the same first and last names. In Part 9, of this video series, we have learnt that, a Unique Constraint, can be used to enforce the uniqueness of values, across one or more columns. There are no major differences between a unique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

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

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

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

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



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

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

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

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

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

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

Advantages and disadvantages of indexes - Part 38

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

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

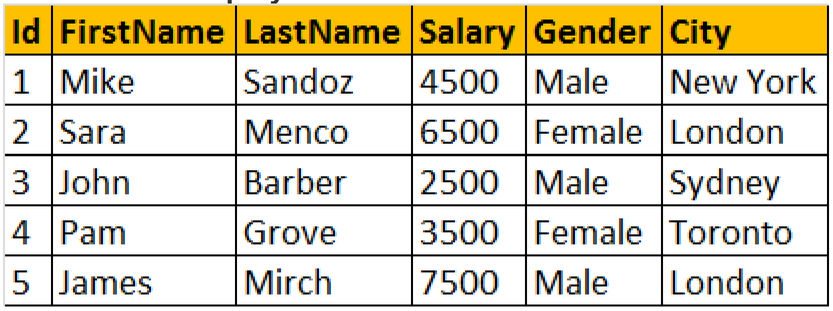
In Part 35, we have learnt that, Indexes are used by queries to find data quickly. In this part, we will learn about the different queries that can benefit from indexes.

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

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

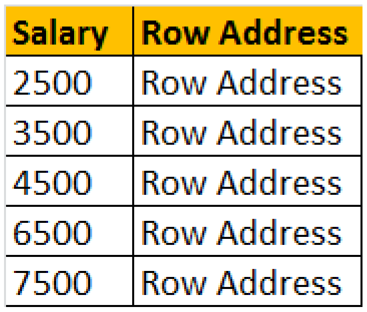
Create a Non-Clustered Index on Salary Column
```
Create NonClustered Index IX_tblEmployee_Salary
On tblEmployee (Salary Asc)
```

Data from tblEmployee table



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

NonClustered Index



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Views in sql server - Part 39

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

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

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

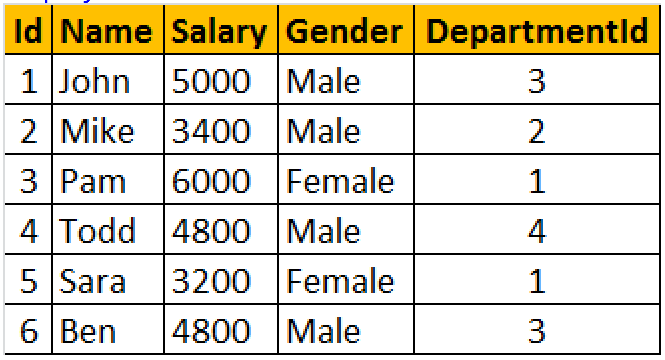
SQL Script to create tblDepartment table:
```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

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

Insert data into tblEmployee table
```
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)

Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)

Insert into tblEmployee values (4,'Todd', 4800, 'Male', 4)
Insert into tblEmployee values (5,'Sara', 3200, 'Female', 1)
Insert into tblEmployee values (6,'Ben', 4800, 'Male', 3)
```

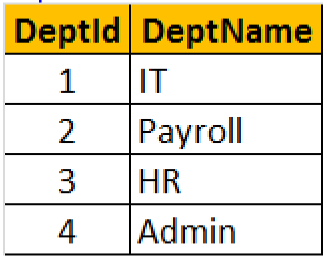
At this point Employees and Departments table should look like this.

Employees Table:



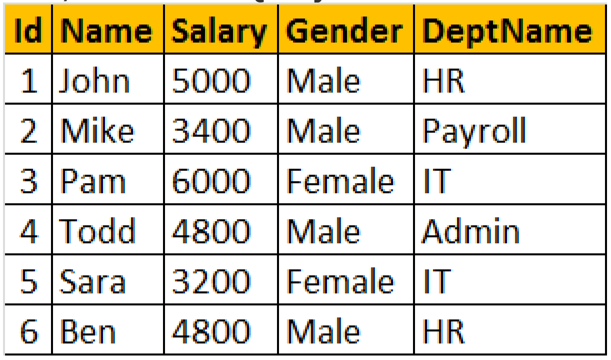
Id	Name	Salary	Gender	DepartmentId
1	John	5000	Male	3
2	Mike	3400	Male	2
3	Pam	6000	Female	1
4	Todd	4800	Male	4
5	Sara	3200	Female	1
6	Ben	4800	Male	3

Departments Table:



DeptId	DeptName
1	IT
2	Payroll
3	HR
4	Admin

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



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

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

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

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

```
Create View vWEmployeesByDepartment
as
Select Id, Name, Salary, Gender, DeptName
```

```
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

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

```
SELECT * from vWEmployeesByDepartment
```

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

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

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

View that returns only IT department employees:

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

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

View that returns all columns except Salary column:

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

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

View that returns summarized data, Total number of employees by Department.

```
Create View vWEmployeesCountByDepartment
```

```
as
Select DeptName, COUNT(Id) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
Group By DeptName
```

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

Updateable Views - Part 40

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

Create Table tblEmployee Script:

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

Script to insert data:

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

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

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

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

```
Select * from vWEmployeesDataExceptSalary
```

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

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

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

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

```
Delete from vWEmployeesDataExceptSalary where Id = 2


Insert into vWEmployeesDataExceptSalary values (2, 'Mikey', 'Male', 2)
```

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

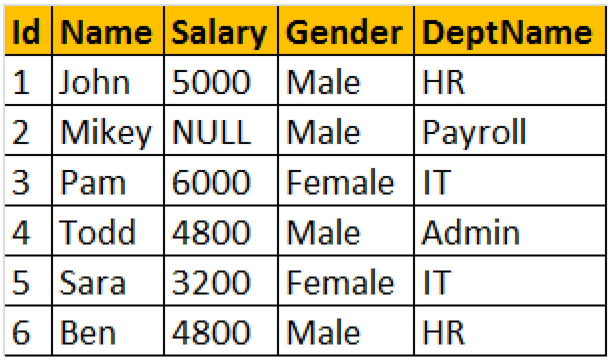
SQL Script to create tblDepartment table

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

Insert data into tblDepartment table

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

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



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

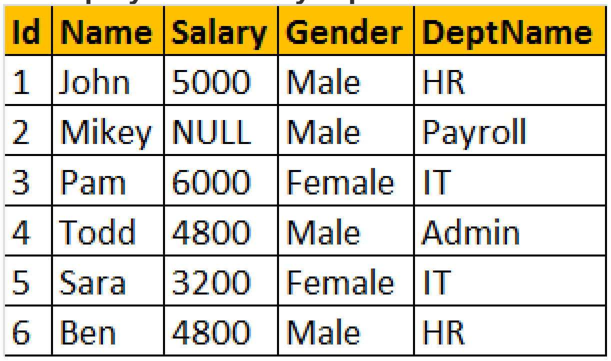
View that joins tblEmployee and tblDepartment

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

Select Data from view vwEmployeeDetailsByDepartment

```
Select * from vwEmployeeDetailsByDepartment
```

vwEmployeeDetailsByDepartment Data:

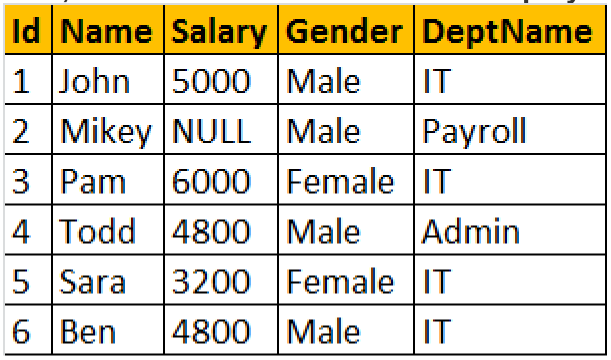


Id	Name	Salary	Gender	DeptName
1	John	5000	Male	HR
2	Mikey	NULL	Male	Payroll
3	Pam	6000	Female	IT
4	Todd	4800	Male	Admin
5	Sara	3200	Female	IT
6	Ben	4800	Male	HR

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

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

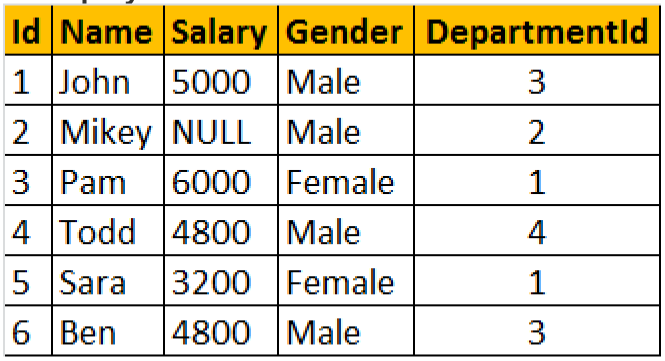
Now, Select data from the view vwEmployeeDetailsByDepartment:



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

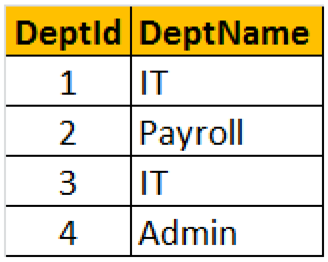
Notice, that Ben's department is also changed to IT. To understand the reasons for incorrect UPDATE, select Data from tblDepartment and tblEmployee base tables.

tblEmployee Table



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

tblDepartment



DeptId	DeptName
1	IT
2	Payroll
3	IT
4	Admin

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

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

Indexed views in sql server - Part 41

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

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

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

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

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

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

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

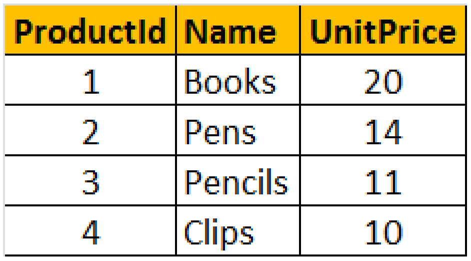
Script to create table tblProductSales

```
Create Table tblProductSales
(
 ProductId int,
 QuantitySold int
)
```

Script to pouplate tblProductSales, with sample data

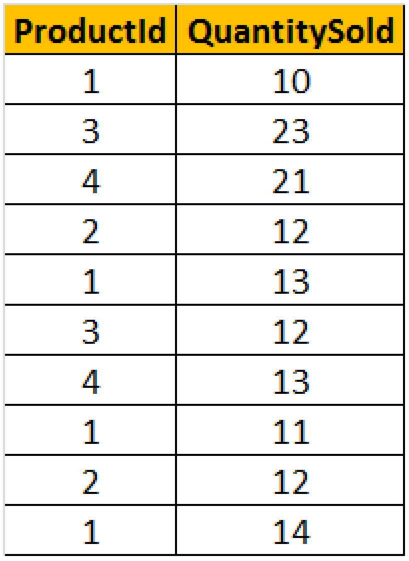
```
Insert into tblProductSales values(1, 10)
Insert into tblProductSales values(3, 23)
Insert into tblProductSales values(4, 21)
Insert into tblProductSales values(2, 12)
Insert into tblProductSales values(1, 13)
Insert into tblProductSales values(3, 12)
Insert into tblProductSales values(4, 13)
Insert into tblProductSales values(1, 11)
Insert into tblProductSales values(2, 12)
Insert into tblProductSales values(1, 14)
```

tblProduct Table



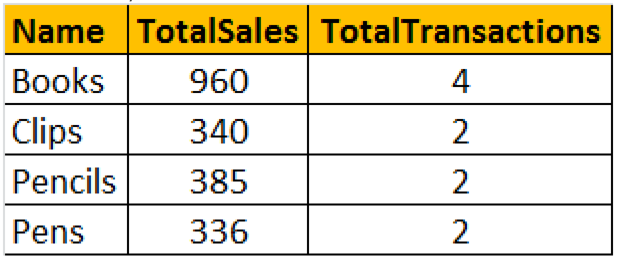
ProductId	Name	UnitPrice
1	Books	20
2	Pens	14
3	Pencils	11
4	Clips	10

tblProductSales Table



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

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



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

Script to create view vWTotalSalesByProduct

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

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

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

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

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

Now, let's create an Index on the view:
The first index that you create on a view, must be a unique clustered index. After the unique clustered index has been created, you can create additional nonclustered indexes.

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

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

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

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

Limitations of views - Part 42

Suggested SQL Server Videos before watching this Video

Part 39 - View basics
Part 40 - Updateable views
Part 41 - Indexed views

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

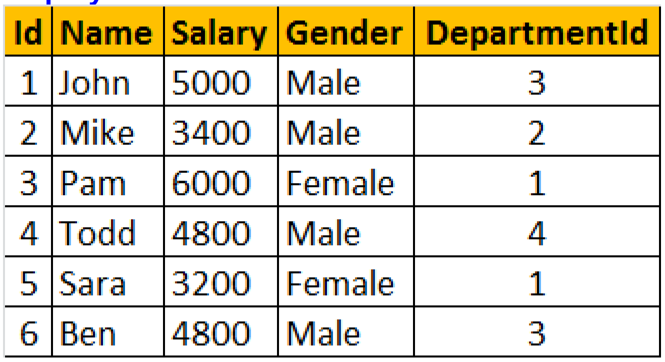
We will use tblEmployee table for our examples. SQL Script to create tblEmployee table:

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

Insert data into tblEmployee table

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

Employee Table



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

```
-- Error : Cannot pass Parameters to Views
Create View vWEmployeeDetails
@Gender nvarchar(20)
as
```

```
Select Id, Name, Gender, DepartmentId
from  tblEmployee
where Gender = @Gender
```

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

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

Calling the function

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

2. Rules and Defaults cannot be associated with views.

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

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

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

4. Views cannot be based on temporary tables.

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

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

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

Email This

DML Triggers - Part 43

In SQL server there are 3 types of triggers

1. DML triggers
2. DDL triggers
3. Logon trigger

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

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

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

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

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

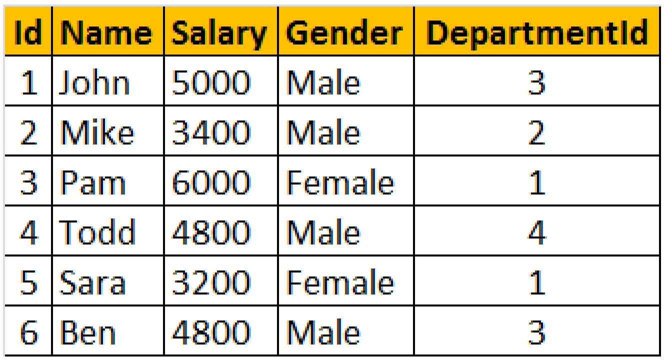
On ther hand, as the name says, INSTEAD of triggers, fires instead of the triggering action. The INSERT, UPDATE, and DELETE statements, can cause an INSTEAD OF trigger to fire INSTEAD OF the respective statement execution.

We will use tblEmployee and tblEmployeeAudit tables for our examples

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

Insert data into tblEmployee table
```
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)

Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
```

tblEmployee



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

SQL Script to create tblEmployeeAudit table:

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

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

Example for AFTER TRIGGER for INSERT event on tblEmployee table:

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

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

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

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

Insert into tblEmployee values (7,'Tan', 2300, 'Female', 3)

Along, the same lines, let us now capture audit information, when a row is deleted from

the table, tblEmployee.
Example for AFTER TRIGGER for DELETE event on tblEmployee table:

```
CREATE TRIGGER tr_tblEMployee_ForDelete
ON tblEmployee
FOR DELETE
AS
BEGIN
 Declare @Id int
 Select @Id = Id from deleted

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

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

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

After update trigger - Part 44

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

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

Create AFTER UPDATE trigger script:

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

Now, execute this query:

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

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

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

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

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

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

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

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

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

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

```
                Set @AuditString = @AuditString + ' NAME from ' + @OldName + ' to ' +
@NewName

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

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

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

            insert into tblEmployeeAudit values(@AuditString)

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

Instead of insert trigger - Part 45

Suggested SQL Server Videos before watching this Video

Part 39 - Views
Part 40 - Updateable Views
Part 43 - DML triggers
Part 44 - DML After Update Trigger

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

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

SQL Script to create tblEmployee table:

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

SQL Script to create tblDepartment table

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

Insert data into tblDepartment table

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

Insert data into tblEmployee table

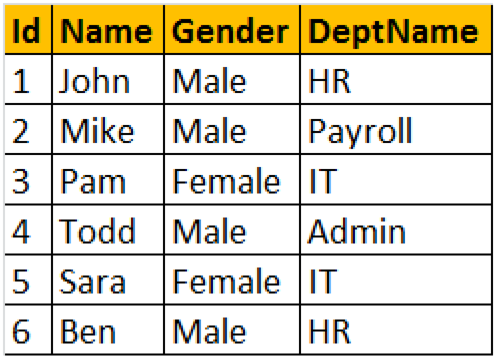
```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)


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

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

Script to create the view:

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

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



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

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

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

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

Script to create INSTEAD OF INSERT trigger:

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

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

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

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

Now, let's execute the insert query:

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

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

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

Email This

Instead of update triggers - Part 46

Suggested SQL Server Videos before watching this Video
Part 43 - DML triggers
Part 44 - DML After Update Trigger
Part 45 - Instead of Insert Trigger

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

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

SQL Script to create tblEmployee table:

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

SQL Script to create tblDepartment table

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

Insert data into tblDepartment table

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

Insert data into tblEmployee table

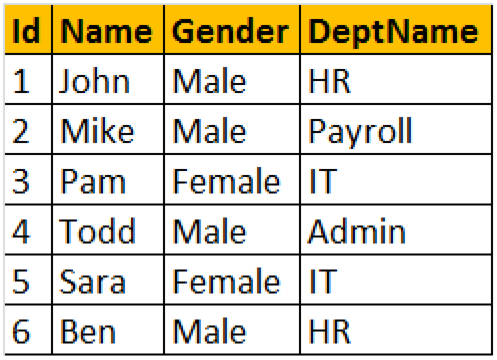
```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)
Insert into tblEmployee values (3,'Pam', 'Female', 1)


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

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

Script to create the view:

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

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



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

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

Now, let's try to update the view, in such a way that, it affects, both the underlying tables, and see, if we get the same error. The following UPDATE statement changes **Name column** from **tblEmployee** and **DeptName column** from **tblDepartment**. So, when we execute this query, we get the same error.

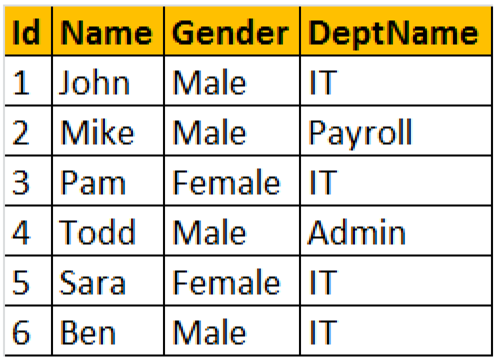
```
Update vWEmployeeDetails
set Name = 'Johny', DeptName = 'IT'
where Id = 1
```

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

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

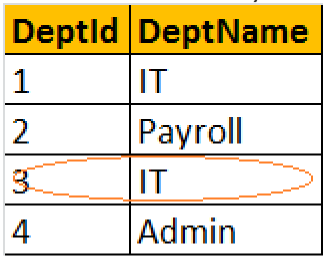
After executing the query, select the data from the view, and notice that **BEN's DeptName** is also changed to **IT**. We intended to just change **JOHN's DeptName**. So, the UPDATE didn't work as expected. This is because, the UPDATE query, updated the **DeptName from HR to IT**, in tblDepartment table.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

Incorrectly Updated View



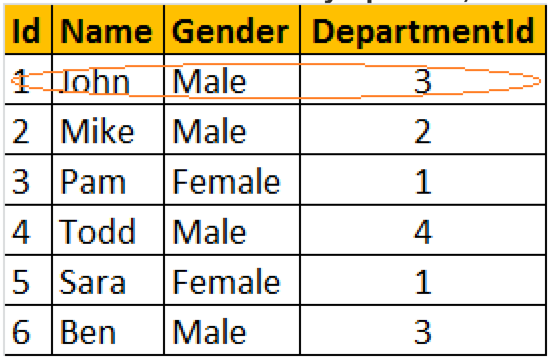
Id	Name	Gender	DeptName
1	John	Male	IT
2	Mike	Male	Payroll
3	Pam	Female	IT
4	Todd	Male	Admin
5	Sara	Female	IT
6	Ben	Male	IT

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



DeptId	DeptName
1	IT
2	Payroll
3	IT
4	Admin

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



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

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

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

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

Script to create INSTEAD OF UPDATE trigger:

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

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

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

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

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

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

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

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

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

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

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

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

Update() function used in the trigger, returns true, even if you update with the same value. For this reason, I recomend to compare values between inserted and deleted tables, rather than relying on Update() function. The Update() function does not operate on a per row basis, but across all rows.
Email This

Instead of delete trigger - Part 47

Suggested SQL Server Videos before watching this Video
Part 45 - Instead of Insert Trigger
Part 46 - Instead of Update Trigger

In this video we will learn about, INSTEAD OF DELETE trigger. An INSTEAD OF DELETE trigger gets fired instead of the DELETE event, on a table or a view. For example, let's say we have, an INSTEAD OF DELETE trigger on a view or a table, and then when you try to update a row from that view or table, instead of the actual DELETE event, the trigger gets fired automatically. INSTEAD OF DELETE TRIGGERS, are used, to delete records from a view, that is based on multiple tables.

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

SQL Script to create tblEmployee table:

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

SQL Script to create tblDepartment table

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

Insert data into tblDepartment table

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

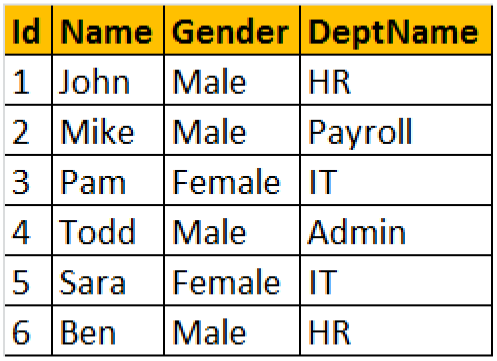
Insert data into tblEmployee table

```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)

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

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

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

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



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

In Part 45, we tried to insert a row into the view, and we got an error stating - 'View or function vWEmployeeDetails is not updatable because the modification affects multiple base tables'. Along, the same lines, in **Part 46**, when we tried to update a view that is based on multiple tables, we got the same error. To get the error, the UPDATE should affect both the base tables. If the update affects only one base table, we don't get the error, but the UPDATE does not work correctly, if the **DeptName** column is updated.

Now, let's try to delete a row from the view, and we get the same error.

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

Script to create INSTEAD OF DELETE trigger:

```
Create Trigger tr_vWEmployeeDetails_InsteadOfDelete
on vWEmployeeDetails
instead of delete
as
```

```
Begin
 Delete tblEmployee
 from tblEmployee
 join deleted
 on tblEmployee.Id = deleted.Id

 --Subquery
 --Delete from tblEmployee
 --where Id in (Select Id from deleted)
End
```

Notice that, the trigger tr_vWEmployeeDetails_InsteadOfDelete, makes use of DELETED table. DELETED table contains all the rows, that we tried to DELETE from the view. So, we are joining the DELETED table with tblEmployee, to delete the rows. You can also use sub-queries to do the same. In most cases JOINs are faster than SUB-QUERIEs. However, in cases, where you only need a subset of records from a table that you are joining with, sub-queries can be faster.

Upon executing the following DELETE statement, the row gets DELETED as expected from tblEmployee table

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

Trigger	INSERTED or DELETED?
Instead of Insert	DELETED table is always empty and the INSERTED table contains the newly inserted data.
Instead of Delete	INSERTED table is always empty and the DELETED table contains the rows deleted
Instead of Update	DELETED table contains OLD data (before update), and inserted table contains NEW data(Updated data)

Derived table and CTE in sql server - Part 48

In this video we will learn about, Derived tables and common table expressions (CTE's). We will also explore the differences between Views, Table Variable, Local and Global Temp Tables, Derived tables and common table expressions.
Let's create the required Employee and Department tables, that we will be using for this demo.

SQL Script to create tblEmployee table:

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

SQL Script to create tblDepartment table

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

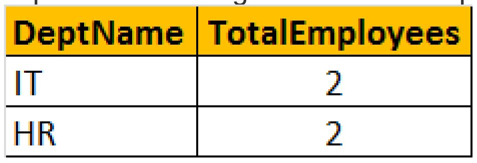
Insert data into tblDepartment table

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

Insert data into tblEmployee table

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

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

Now, we want to write a query which would return the following output. The query should return, the Department Name and Total Number of employees, with in the department. The departments with greatar than or equal to 2 employee should only be returned.



DeptName	TotalEmployees
IT	2
HR	2

Obviously, there are severl ways to do this. Let's see how to achieve this, with the help of a view

Script to create the View

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

Query using the view:

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

Note: Views get saved in the database, and can be available to other queries and stored procedures. However, if this view is only used at this one place, it can be easily eliminated using other options, like CTE, Derived Tables, Temp Tables, Table Variable etc.

Now, let's see, how to achieve the same using, temporary tables. We are using local temporary tables here.

```
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
```

```
into #TempEmployeeCount
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId

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

Drop Table #TempEmployeeCount
```

Note: Temporary tables are stored in TempDB. Local temporary tables are visible only in the current session, and can be shared between nested stored procedure calls. Global temporary tables are visible to other sessions and are destroyed, when the last connection referencing the table is closed.

Using Table Variable:

```
Declare @tblEmployeeCount table
(DeptName nvarchar(20),DepartmentId int, TotalEmployees int)

Insert @tblEmployeeCount
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId

Select DeptName, TotalEmployees
From @tblEmployeeCount
where  TotalEmployees >= 2
```

Note: Just like TempTables, a table variable is also created in TempDB. The scope of a table variable is the batch, stored procedure, or statement block in which it is declared. They can be passed as parameters between procedures.

Using Derived Tables

```
Select DeptName, TotalEmployees
from
 (
  Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
  from tblEmployee
  join tblDepartment
  on tblEmployee.DepartmentId = tblDepartment.DeptId
  group by DeptName, DepartmentId
 )
as EmployeeCount
where TotalEmployees >= 2
```

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

Using CTE

```
With EmployeeCount(DeptName, DepartmentId, TotalEmployees)
as
(
 Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
 from tblEmployee
 join tblDepartment
 on tblEmployee.DepartmentId = tblDepartment.DeptId
 group by DeptName, DepartmentId
)

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

Note: A CTE can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is similar to a derived table in that it is not stored as an object and lasts only for the duration of the query.

Email This

Common Table Expressions - Part 49

Common table expression (CTE) is introduced in SQL server 2005. A **CTE** is a temporary result set, that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement, that immediately follows the **CTE.**

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

SQL Script to create tblEmployee table:

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

SQL Script to create tblDepartment table

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

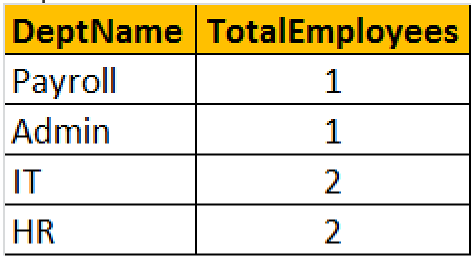
Insert data into tblDepartment table

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

Insert data into tblEmployee table

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

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

Write a query using CTE, to display the total number of Employees by Department Name. The output should be as shown below.



DeptName	TotalEmployees
Payroll	1
Admin	1
IT	2
HR	2

Before we write the query, let's look at the syntax for creating a CTE.

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

SQL query using CTE:

```
With EmployeeCount(DepartmentId, TotalEmployees)
as
(
 Select DepartmentId, COUNT(*) as TotalEmployees
 from tblEmployee
 group by DepartmentId
)

Select DeptName, TotalEmployees
from tblDepartment
join EmployeeCount
on tblDepartment.DeptId = EmployeeCount.DepartmentId
order by TotalEmployees
```

We define a CTE, using **WITH** keyword, followed by the name of the CTE. In our example, **EmployeeCount** is the name of the CTE. Within parentheses, we specify the columns that make up the CTE. **DepartmentId** and **TotalEmployees** are the columns of **EmployeeCount** CTE. These 2 columns map to the columns returned by the **SELECT CTE query**. The CTE column names and CTE query column names can be different. Infact, CTE column names are optional. However, if you do specify, the number of **CTE columns** and the **CTE SELECT query** columns should be same. Otherwise you will get an error stating - 'EmployeeCount has fewer columns than were specified in the column list'. The column list, is followed by the as keyword, following which we have the CTE query within a pair of parentheses.

EmployeeCount CTE is being joined with **tblDepartment** table, in the SELECT query, that immediately follows the CTE. Remember, a CTE can only be referenced by a SELECT, INSERT, UPDATE, or DELETE statement, **that immediately follows the CTE**. If you try to do something else in between, we get an error stating - 'Common table expression defined but not used'. The following SQL, raise an error.

```
With EmployeeCount(DepartmentId, TotalEmployees)
as
(
 Select DepartmentId, COUNT(*) as TotalEmployees
 from tblEmployee
 group by DepartmentId
)

Select 'Hello'

Select DeptName, TotalEmployees
from tblDepartment
join EmployeeCount
on tblDepartment.DeptId = EmployeeCount.DepartmentId
order by TotalEmployees
```

It is also, possible to create multiple CTE's using a single WITH clause.

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

Email This

Updatable CTE - Part 50

Is it possible to UPDATE a CTE?

Yes & No, depending on the number of base tables, the CTE is created upon, and the number of base tables affected by the UPDATE statement. If this is not clear at the moment, don't worry. We will try to understand this with an example.

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

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

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

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

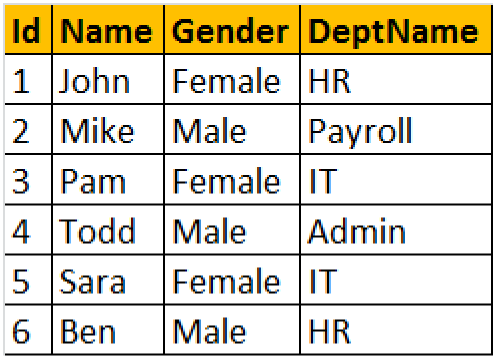
Insert data into tblEmployee table
```
Insert into tblEmployee values (1,'John', 'Male', 3)
Insert into tblEmployee values (2,'Mike', 'Male', 2)

Insert into tblEmployee values (3,'Pam', 'Female', 1)


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

Let's create a simple common table expression, based on tblEmployee table. **Employees_Name_Gender** CTE is getting all the required columns from one base table tblEmployee.
```
With Employees_Name_Gender
as
(
 Select Id, Name, Gender from tblEmployee
)
Select * from Employees_Name_Gender
```

Let's now, UPDATE JOHN's gender from Male to Female, using the **Employees_Name_Gender CTE**
```
With Employees_Name_Gender
as
(
 Select Id, Name, Gender from tblEmployee
)
Update Employees_Name_Gender Set Gender = 'Female' where Id = 1
```

Now, query the tblEmployee table. JOHN's gender is actually UPDATED. So, if a CTE is created on one base table, then it is possible to UPDATE the CTE, which in turn will update the underlying base table. In this case, UPDATING **Employees_Name_Gender** CTE, updates **tblEmployee** table.

Now, let's create a CTE, on both the tables - tblEmployee and tblDepartment. The CTE should return, Employee Id, Name, Gender and Department. In short the output should be as shown below.



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

CTE, that returns Employees by Department

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Select * from EmployeesByDepartment
```

Let's update this CTE. Let's change JOHN's Gender from **Female to Male**. Here, the CTE is based on 2 tables, but the UPDATE statement affects only one base table **tblEmployee**. So the UPDATE succeeds. So, if a CTE is based on more than one table, and if the UPDATE affects only one base table, then the UPDATE is allowed.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set Gender = 'Male' where Id = 1
```

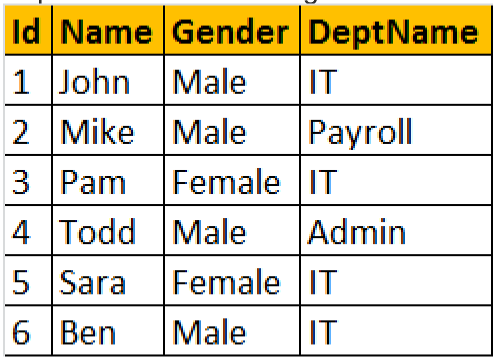
Now, let's try to **UPDATE the CTE**, in such a way, that the update affects both the tables - **tblEmployee and tblDepartment**. This UPDATE statement changes **Gender** from **tblEmployee** table and **DeptName** from **tblDepartment** table. When you execute this UPDATE, you get an error stating - 'View or function EmployeesByDepartment is not updatable because the modification affects multiple base tables'. So, if a CTE is based on multiple tables, and if the UPDATE statement affects more than 1 base table, then the UPDATE is not allowed.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set
Gender = 'Female', DeptName = 'IT'
where Id = 1
```

Finally, let's try to UPDATE just the DeptName. Let's change JOHN's DeptName from HR to IT. Before, you execute the UPDATE statement, notice that BEN is also currently in HR department.

```
With EmployeesByDepartment
as
(
 Select Id, Name, Gender, DeptName
 from tblEmployee
 join tblDepartment
 on tblDepartment.DeptId = tblEmployee.DepartmentId
)
Update EmployeesByDepartment set
DeptName = 'IT' where Id = 1
```

After you execute the UPDATE. Select data from the CTE, and you will see that BEN's DeptName is also changed to IT.



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

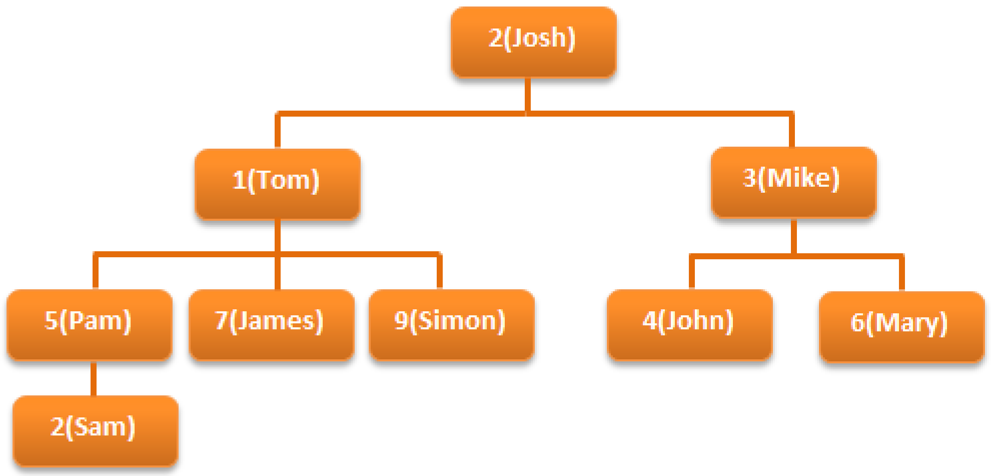
This is because, when we updated the **CTE**, the UPDATE has actually changed the **DeptName** from **HR** to **IT**, in **tblDepartment** table, instead of changing the **DepartmentId** column (from 3 to 1) in **tblEmployee** table. So, if a CTE is based on multiple tables, and if the UPDATE statement affects only one base table, the update succeeds. But the update may not work as you expect.

So in short if,
1. A CTE is based on a single base table, then the UPDATE suceeds and works as expected.
2. A CTE is based on more than one base table, and if the UPDATE affects multiple base tables, the update is not allowed and the statement terminates with an error.
3. A CTE is based on more than one base table, and if the UPDATE affects only one base table, the UPDATE succeeds(but not as expected always)
Email This

Recursive CTE - Part 51

Suggested SQL Server videos
Part 49 - Common Table Expressions (CTE)
Part 50 - Updatable CTE

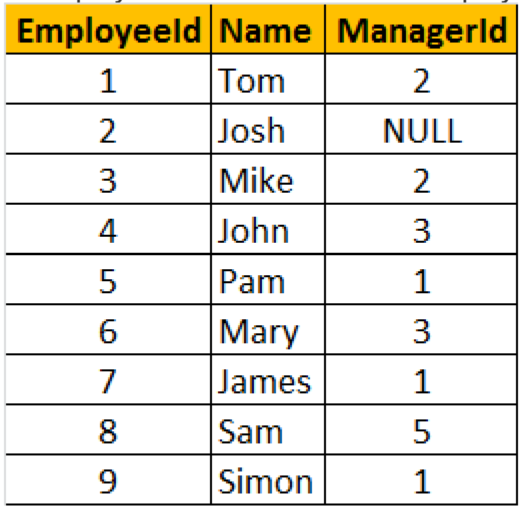
A CTE that references itself is called as recursive CTE. Recursive CTE's can be of great help when displaying hierarchical data. Example, displaying employees in an organization hierarchy. A simple organization chart is shown below.



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

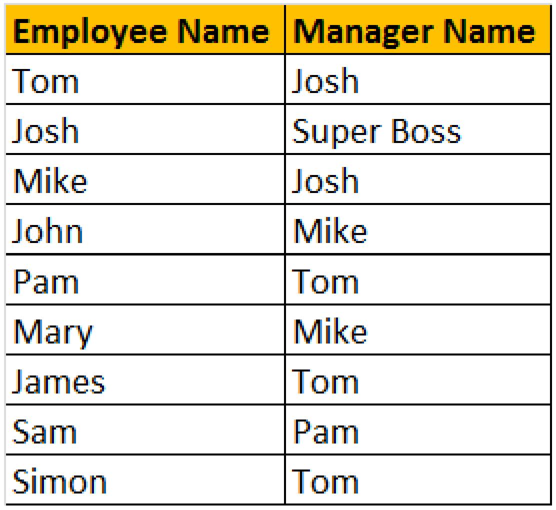
```
Create Table tblEmployee
(
  EmployeeId int Primary key,
  Name nvarchar(20),
  ManagerId int
)

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

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



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

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



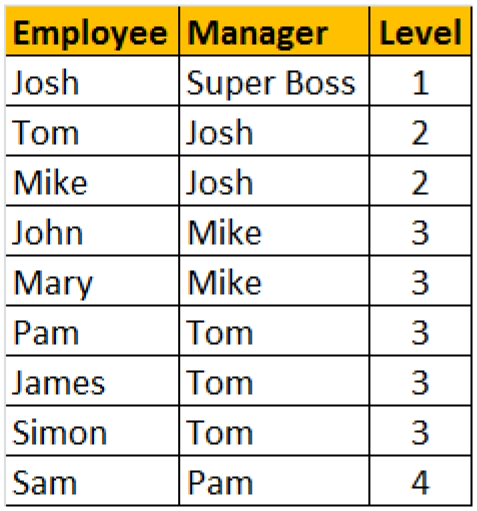
Employee Name	Manager Name
Tom	Josh
Josh	Super Boss
Mike	Josh
John	Mike
Pam	Tom
Mary	Mike
James	Tom
Sam	Pam
Simon	Tom

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

SELF JOIN QUERY:

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

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



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

We can easily achieve this using a self referencing CTE.

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

    union all

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

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

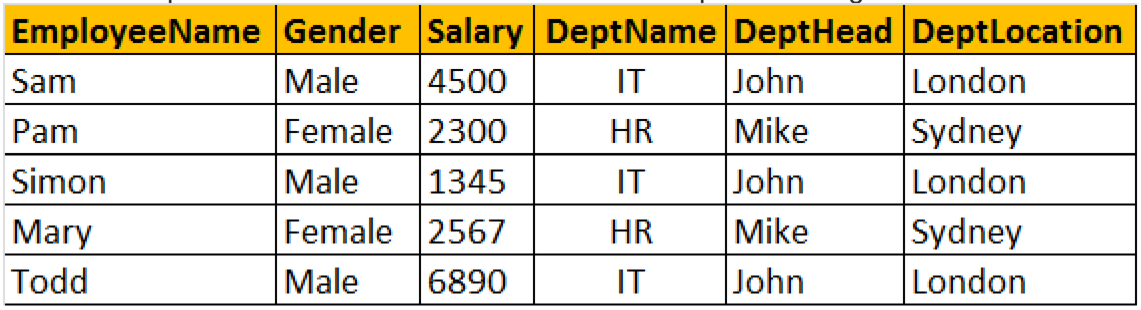
Email This

Database Normalization - Part 52

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

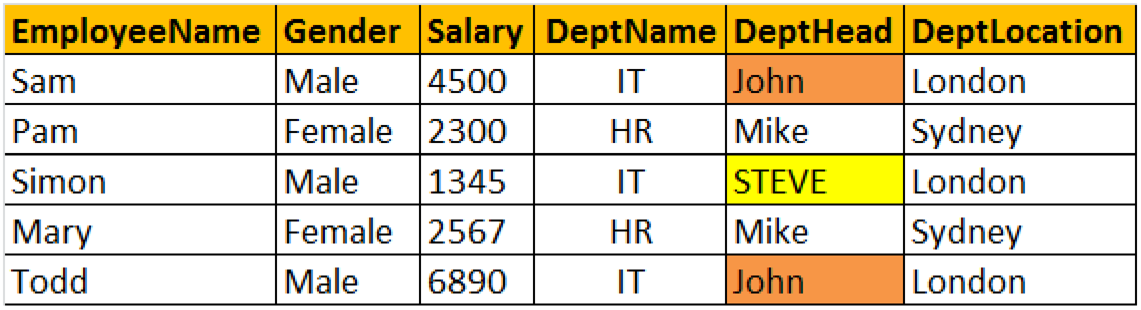
Database normalization is the process of organizing data to minimize data redundancy (data duplication), which in turn ensures data consistency.

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



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

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

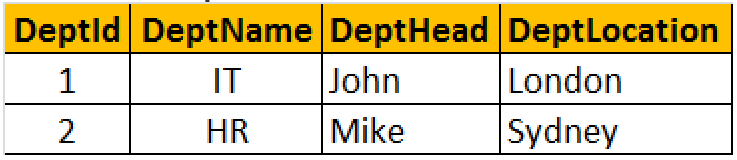


EmployeeName	Gender	Salary	DeptName	DeptHead	DeptLocation
Sam	Male	4500	IT	John	London
Pam	Female	2300	HR	Mike	Sydney
Simon	Male	1345	IT	STEVE	London
Mary	Female	2567	HR	Mike	Sydney
Todd	Male	6890	IT	John	London

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

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

Normalized Departments Table



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

Normalized Employees Table

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

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

Now, let's explore the first normal form (1NF). A table is said to be in 1NF, if

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

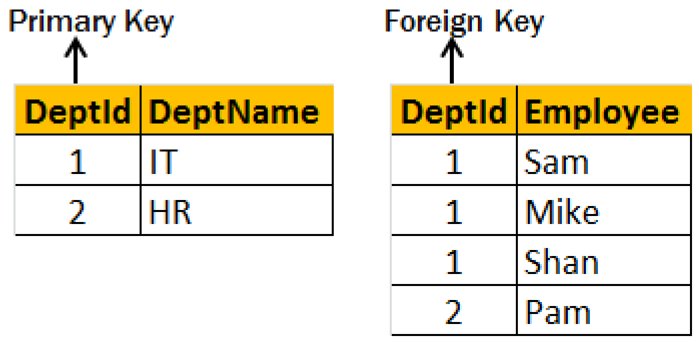
In the table below, data in Employee column is not atomic. It contains multiple employees seperated by comma. From the data you can see that in the IT department, we have 3 employees - Sam, Mike, Shan. Now, let's say I want to change just, SHAN name. **It is not possible, we have to update the entire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

DeptName	Employee
IT	Sam, Mike, Shan
HR	Pam

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

DeptName	Employee1	Employee2	Employee3
IT	Sam	Mike	Shan
HR	Pam		

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



DeptId	DeptName
1	IT
2	HR

DeptId	Employee
1	Sam
1	Mike
1	Shan
2	Pam

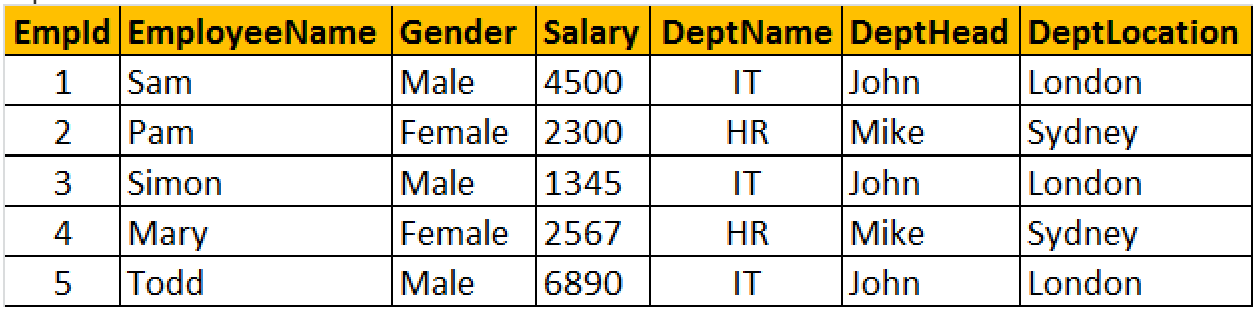
Email This

Second Normal Form and Third Normal Form - Part 53

Suggested sql server videos
Part 52 - Database Normalization & First Normal Form

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

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

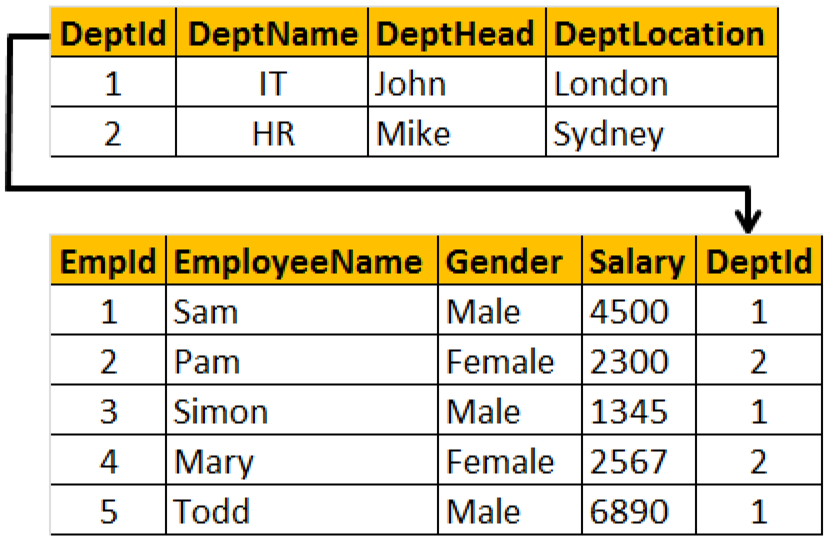


EmpId	EmployeeName	Gender	Salary	DeptName	DeptHead	DeptLocation
1	Sam	Male	4500	IT	John	London
2	Pam	Female	2300	HR	Mike	Sydney
3	Simon	Male	1345	IT	John	London
4	Mary	Female	2567	HR	Mike	Sydney
5	Todd	Male	6890	IT	John	London

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

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

Table design in Second Normal Form (2NF)



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

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

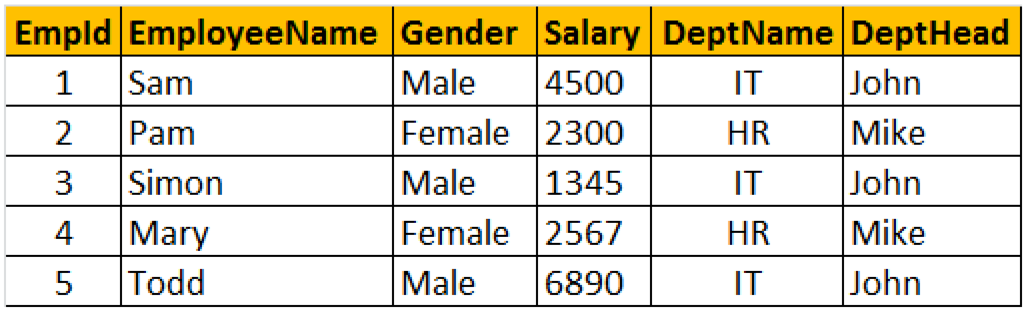
Third Normal Form (3NF):

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

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

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

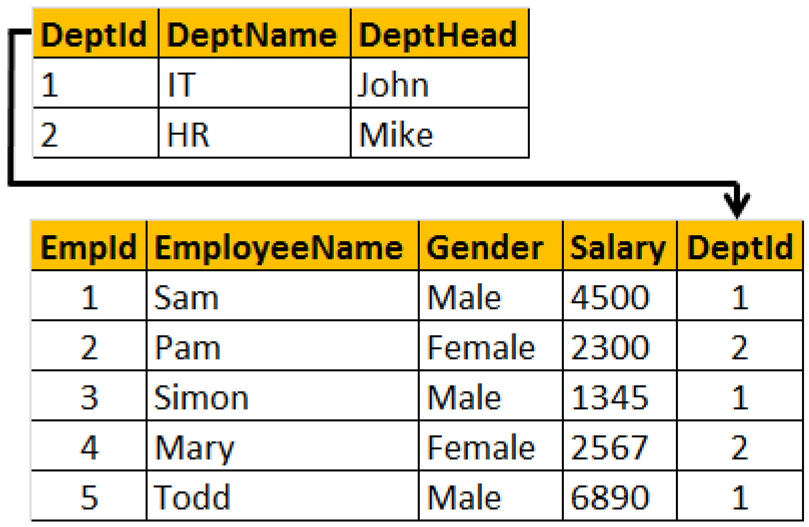
Let's look at another example of Third Normal Form violation. In the table below, **DeptHead** column is not fully dependent on **EmpId** column. **DeptHead** is also dependent on **DeptName**. So, this table is not in **3NF**.



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

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

Table design in Third Normal Form (3NF)



DeptId	DeptName	DeptHead
1	IT	John
2	HR	Mike

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

Email This

Pivot operator in sql server - Part 54

Suggested SQL Server Videos:
Part 11 - Group By
Part 48 - Derived table and CTE in sql server

One of my youtube channel subscribers, has asked me to make a video on **PIVOT** operator. So here we are with another sql server video.

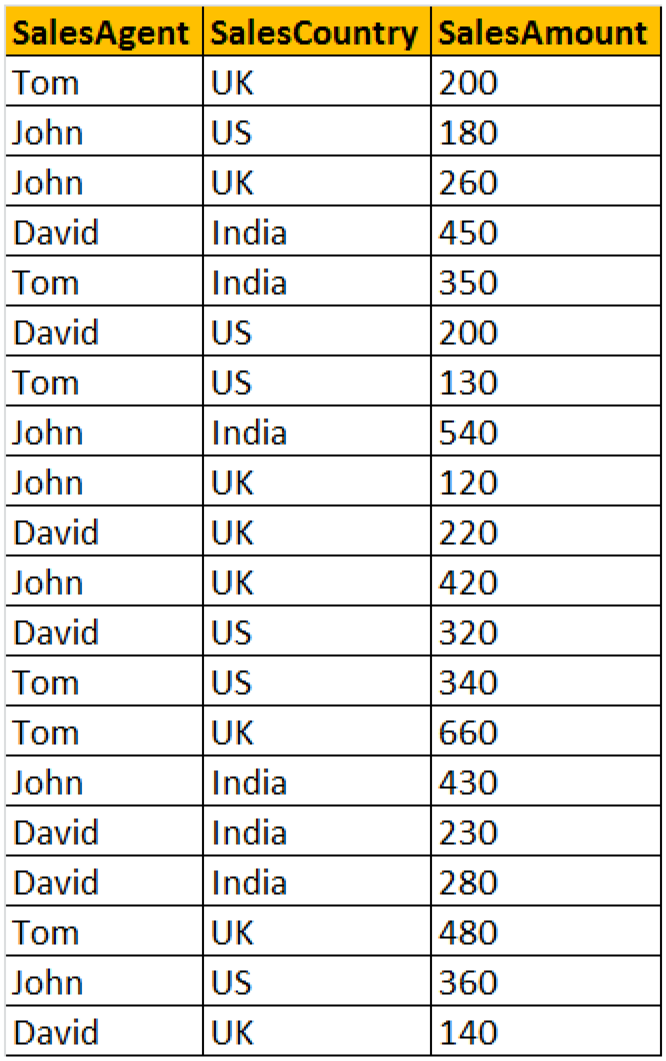
Pivot is a sql server operator that can be used to turn **unique values from one column**, into **multiple columns** in the output, there by effectively **rotating a table**.

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

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

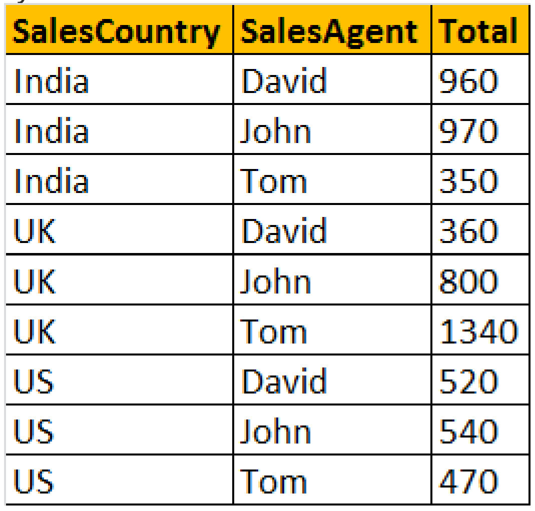
```
Insert into tblProductSales values('Tom', 'UK', 200)
Insert into tblProductSales values('John', 'US', 180)
Insert into tblProductSales values('John', 'UK', 260)
Insert into tblProductSales values('David', 'India', 450)
Insert into tblProductSales values('Tom', 'India', 350)
Insert into tblProductSales values('David', 'US', 200)
Insert into tblProductSales values('Tom', 'US', 130)
Insert into tblProductSales values('John', 'India', 540)
Insert into tblProductSales values('John', 'UK', 120)
Insert into tblProductSales values('David', 'UK', 220)
Insert into tblProductSales values('John', 'UK', 420)
Insert into tblProductSales values('David', 'US', 320)
Insert into tblProductSales values('Tom', 'US', 340)
Insert into tblProductSales values('Tom', 'UK', 660)
Insert into tblProductSales values('John', 'India', 430)
Insert into tblProductSales values('David', 'India', 230)
Insert into tblProductSales values('David', 'India', 280)
Insert into tblProductSales values('Tom', 'UK', 480)
Insert into tblProductSales values('John', 'US', 360)
Insert into tblProductSales values('David', 'UK', 140)
```

Select * from tblProductSales: As you can see, we have 3 sales agents selling in 3 countries



SalesAgent	SalesCountry	SalesAmount
Tom	UK	200
John	US	180
John	UK	260
David	India	450
Tom	India	350
David	US	200
Tom	US	130
John	India	540
John	UK	120
David	UK	220
John	UK	420
David	US	320
Tom	US	340
Tom	UK	660
John	India	430
David	India	230
David	India	280
Tom	UK	480
John	US	360
David	UK	140

Now, let's write a query which returns TOTAL SALES, grouped by **SALESCOUNTRY** and **SALESAGENT**. The output should be as shown below.

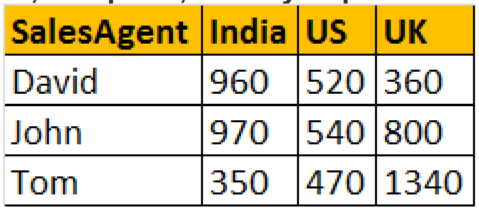


SalesCountry	SalesAgent	Total
India	David	960
India	John	970
India	Tom	350
UK	David	360
UK	John	800
UK	Tom	1340
US	David	520
US	John	540
US	Tom	470

A simple GROUP BY query can produce this output.

```
Select SalesCountry, SalesAgent, SUM(SalesAmount) as Total
from tblProductSales
group by SalesCountry, SalesAgent
order by SalesCountry, SalesAgent
```

At, this point, let's try to present the same data in different format using PIVOT operator.



SalesAgent	India	US	UK
David	960	520	360
John	970	540	800
Tom	350	470	1340

Query using PIVOT operator:

```
Select SalesAgent, India, US, UK
from tblProductSales
Pivot
(
   Sum(SalesAmount) for SalesCountry in ([India],[US],[UK])
) as PivotTable
```

This PIVOT query is converting the unique column values (India, US, UK) in **SALESCOUNTRY** column, **into Columns** in the output, along with performing aggregations on the **SALESAMOUNT** column. The Outer query, simply, selects **SALESAGENT** column from **tblProductSales** table, along with pivoted columns from the PivotTable.

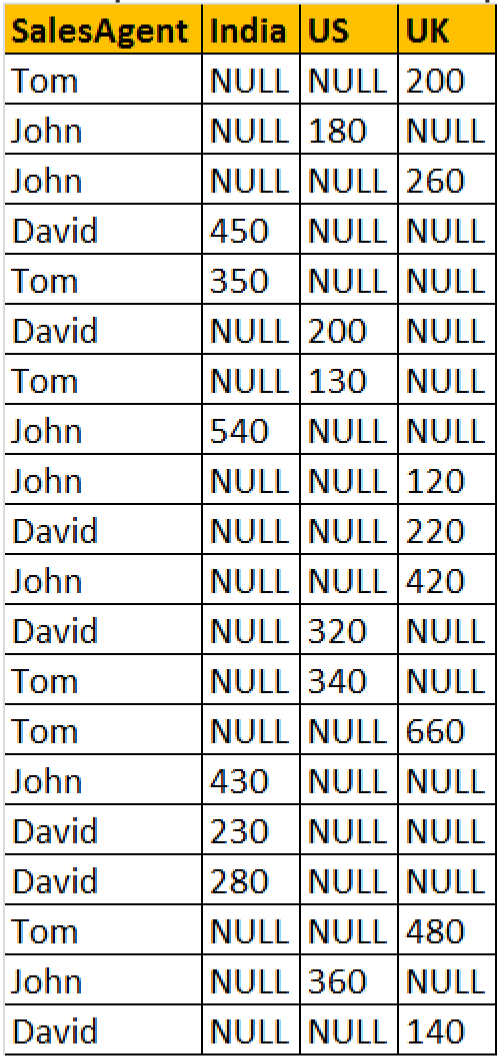
Having understood the basics of PIVOT, let's look at another example. Let's create **tblProductsSale**, a slight variation of **tblProductSales**, that we have already created. The table, that we are creating now, has got an additional **Id** column.

```
Create Table tblProductsSale
(
   Id int primary key,
   SalesAgent nvarchar(50),
   SalesCountry nvarchar(50),
   SalesAmount int
)

Insert into tblProductsSale values(1, 'Tom', 'UK', 200)
Insert into tblProductsSale values(2, 'John', 'US', 180)
Insert into tblProductsSale values(3, 'John', 'UK', 260)
Insert into tblProductsSale values(4, 'David', 'India', 450)
Insert into tblProductsSale values(5, 'Tom', 'India', 350)
Insert into tblProductsSale values(6, 'David', 'US', 200)
Insert into tblProductsSale values(7, 'Tom', 'US', 130)
Insert into tblProductsSale values(8, 'John', 'India', 540)
Insert into tblProductsSale values(9, 'John', 'UK', 120)
Insert into tblProductsSale values(10, 'David', 'UK', 220)
Insert into tblProductsSale values(11, 'John', 'UK', 420)
Insert into tblProductsSale values(12, 'David', 'US', 320)
Insert into tblProductsSale values(13, 'Tom', 'US', 340)
Insert into tblProductsSale values(14, 'Tom', 'UK', 660)
Insert into tblProductsSale values(15, 'John', 'India', 430)
Insert into tblProductsSale values(16, 'David', 'India', 230)
Insert into tblProductsSale values(17, 'David', 'India', 280)
Insert into tblProductsSale values(18, 'Tom', 'UK', 480)
Insert into tblProductsSale values(19, 'John', 'US', 360)
Insert into tblProductsSale values(20, 'David', 'UK', 140)
```

Now, run the same PIVOT query that we have already created, just by changing the name of the table to **tblProductsSale** instead of **tblProductSales**

```
Select SalesAgent, India, US, UK
from tblProductsSale
Pivot
(
   Sum(SalesAmount) for SalesCountry in ([India],[US],[UK])
)
as PivotTable
```

This output is not what we have expected.



SalesAgent	India	US	UK
Tom	NULL	NULL	200
John	NULL	180	NULL
John	NULL	NULL	260
David	450	NULL	NULL
Tom	350	NULL	NULL
David	NULL	200	NULL
Tom	NULL	130	NULL
John	540	NULL	NULL
John	NULL	NULL	120
David	NULL	NULL	220
John	NULL	NULL	420
David	NULL	320	NULL
Tom	NULL	340	NULL
Tom	NULL	NULL	660
John	430	NULL	NULL
David	230	NULL	NULL
David	280	NULL	NULL
Tom	NULL	NULL	480
John	NULL	360	NULL
David	NULL	NULL	140

This is because of the presence of **Id** column in **tblProductsSale**, which is also considered when performing pivoting and group by. To eliminate this from the calculations, we have used derived table, which only selects, **SALESAGENT, SALESCOUNTRY**, and **SALESAMOUNT**. The rest of the query is very similar to what we have already seen.

```
Select SalesAgent, India, US, UK
from
(
  Select SalesAgent, SalesCountry, SalesAmount from tblProductsSale
) as SourceTable
Pivot
(
 Sum(SalesAmount) for SalesCountry in (India, US, UK)
) as PivotTable
```

UNPIVOT performs the opposite operation to **PIVOT** by rotating columns of a table-valued expression into column values.

The syntax of PIVOT operator from MSDN

```
SELECT <non-pivoted column>,
   [first pivoted column] AS <column name>,
   [second pivoted column] AS <column name>,
   ...
   [last pivoted column] AS <column name>
FROM
   (<SELECT query that produces the data>)
   AS <alias for the source query>
PIVOT
(
   <aggregation function>(<column being aggregated>)
FOR
   [<column that contains the values that will become column headers>]
   IN ( [first pivoted column], [second pivoted column], ... [last pivoted column])
)
AS <alias for the pivot table>
<optional ORDER BY clause>;
```

Error handling in sql server 2000 - Part 55

Suggested SQL Server videos

Part 18 - Stored procedures

With the introduction of Try/Catch blocks in SQL Server 2005, error handling in sql server, is now similar to programming languages like C#, and java. Before understanding error handling using try/catch, let's step back and understand how error handling was done in SQL Server 2000, using system function @@**Error**. Sometimes, system functions that begin with two at signs **(@@)**, are called as **global variables**. They are not variables and do not have the same behaviours as variables, instead they are very similar to functions.

Now let's create **tblProduct** and **tblProductSales**, that we will be using for the rest of this demo.

SQL script to create tblProduct

```
Create Table tblProduct
(
 ProductId int NOT NULL primary key,
 Name nvarchar(50),
 UnitPrice int,
 QtyAvailable int
)
```

SQL script to load data into tblProduct

```
Insert into tblProduct values(1, 'Laptops', 2340, 100)
Insert into tblProduct values(2, 'Desktops', 3467, 50)
```

SQL script to create tblProductSales

```
Create Table tblProductSales
(
 ProductSalesId int primary key,
 ProductId int,
 QuantitySold int
)

Create Procedure spSellProduct
@ProductId int,
@QuantityToSell int
as
Begin
 -- Check the stock available, for the product we want to sell
 Declare @StockAvailable int
 Select @StockAvailable = QtyAvailable
 from tblProduct where ProductId = @ProductId

 -- Throw an error to the calling application, if enough stock is not available
 if(@StockAvailable < @QuantityToSell)
   Begin
  Raiserror('Not enough stock available',16,1)
   End
 -- If enough stock available
 Else
   Begin
    Begin Tran
        -- First reduce the quantity available
  Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)
  where ProductId = @ProductId

  Declare @MaxProductSalesId int
  -- Calculate MAX ProductSalesId
  Select @MaxProductSalesId = Case When
```

```
          MAX(ProductSalesId) IS NULL
          Then 0 else MAX(ProductSalesId) end
         from tblProductSales
  -- Increment @MaxProductSalesId by 1, so we don't get a primary key violation
  Set @MaxProductSalesId = @MaxProductSalesId + 1
  Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)
    Commit Tran
   End
End
```

1. **Stored procedure - spSellProduct**, has 2 parameters - @**ProductId** and @**QuantityToSell**. @ProductId specifies the product that we want to sell, and @QuantityToSell specifies, the quantity we would like to sell.

2. Sections of the stored procedure is commented, and is self explanatory.

3. In the procedure, we are using **Raiserror**() function to return an error message back to the calling application, if the stock available is less than the quantity we are trying to sell. We have to pass atleast 3 parameters to the Raiserror() function.
RAISERROR('Error Message', ErrorSeverity, ErrorState)
Severity and State are integers. In most cases, when you are returning custom errors, the severity level is 16, which indicates general errors that can be corrected by the user. In this case, the error can be corrected, by adjusting the **@QuantityToSell**, to be less than or equal to the stock available. ErrorState is also an integer between 1 and 255. RAISERROR only generates errors with state from 1 through 127.

4. The problem with this procedure is that, the **transaction is always committed**. Even, if there is an error somewhere, between updating **tblProduct** and **tblProductSales** table. In fact, the main purpose of wrapping these 2 statments (Update tblProduct Statement & Insert into tblProductSales statement) in a transaction is to ensure that, both of the statements are treated as a single unit. For example, if we have an error when executing the second statement, then the first statement should also be rolledback.

In SQL server 2000, to detect errors, we can use **@@Error** system function. @@Error returns a NON-ZERO value, if there is an error, otherwise ZERO, indicating that the previous sql statement encountered no errors. The stored procedure **spSellProductCorrected**, makes use of @@ERROR system function to detect any errors that may have occurred. If there are errors, roll back the transaction, else commit the transaction. If you comment the line (Set @MaxProductSalesId = @MaxProductSalesId + 1), and then execute the stored procedure there will be a primary key violation error, when trying to insert into **tblProductSales**. As a result of this the entire transaction will be rolled back.

```
Alter Procedure spSellProductCorrected
@ProductId int,
@QuantityToSell int
as
Begin
 -- Check the stock available, for the product we want to sell
 Declare @StockAvailable int
 Select @StockAvailable = QtyAvailable
 from tblProduct where ProductId = @ProductId

 -- Throw an error to the calling application, if enough stock is not available
 if(@StockAvailable < @QuantityToSell)
   Begin
  Raiserror('Not enough stock available',16,1)
   End
 -- If enough stock available
 Else
   Begin
    Begin Tran
         -- First reduce the quantity available
  Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)
  where ProductId = @ProductId

  Declare @MaxProductSalesId int
  -- Calculate MAX ProductSalesId
  Select @MaxProductSalesId = Case When
          MAX(ProductSalesId) IS NULL
          Then 0 else MAX(ProductSalesId) end
         from tblProductSales
  -- Increment @MaxProductSalesId by 1, so we don't get a primary key violation
  Set @MaxProductSalesId = @MaxProductSalesId + 1
  Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)
  if(@@ERROR <> 0)
  Begin
   Rollback Tran
   Print 'Rolled Back Transaction'
  End
  Else
  Begin
   Commit Tran
   Print 'Committed Transaction'
  End
   End
End
```

Note: @@ERROR is cleared and reset on each statement execution. Check it immediately following the statement being verified, or save it to a local variable that can be checked later.

In **tblProduct** table, we already have a record with **ProductId = 2**. So the insert statement causes a primary key violation error. @@ERROR retains the error number, as we are checking for it immediately after the statement that cause the error.

```
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
if(@@ERROR <> 0)
 Print 'Error Occurred'
Else
```

```
 Print 'No Errors'
```

On the other hand, when you execute the code below, you get message **'No Errors'** printed. This is because the @@ERROR is cleared and reset on each statement execution.

```
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
--At this point @@ERROR will have a NON ZERO value
Select * from tblProduct
--At this point @@ERROR gets reset to ZERO, because the
--select statement successfullyexecuted
if(@@ERROR <> 0)
 Print 'Error Occurred'
Else
 Print 'No Errors'
```

In this example, we are storing the value of @@Error function to a local variable, which is then used later.

```
Declare @Error int
Insert into tblProduct values(2, 'Mobile Phone', 1500, 100)
Set @Error = @@ERROR
Select * from tblProduct
if(@Error <> 0)
 Print 'Error Occurred'
Else
 Print 'No Errors'
```

Error handling in sql server 2005, and later versions - Part 56

Suggested SQL Server Videos
Part 18 - Stored Procedures
Part 55 - Error handling in SQL Server 2000

In Part 55, of this video series we have seen Handling errors in SQL Server using **@@Error** system function. In this session we will see, how to achieve the same using Try/Catch blocks.

Syntax:

```
BEGIN TRY
    { Any set of SQL statements }
END TRY
BEGIN CATCH
    [ Optional: Any set of SQL statements ]
END CATCH
[Optional: Any other SQL Statements]
```

Any set of SQL statements, that can possibly throw an exception are wrapped between BEGIN TRY and END TRY blocks. If there is an exception in the TRY block, the control immediately, jumps to the CATCH block. If there is no exception, CATCH block will be skipped, and the statements, after the CATCH block are executed.

Errors trapped by a CATCH block are not returned to the calling application. If any part of the error information must be returned to the application, the code in the CATCH block must do so by using RAISERROR() function.

1. **In procedure spSellProduct**, Begin Transaction and Commit Transaction statements are wrapped between Begin Try and End Try block. If there are no errors in the code that is enclosed in the TRY block, then COMMIT TRANSACTION gets executed and the changes are made permanent. On the other hand, if there is an error, then the control immediately jumps to the CATCH block. In the CATCH block, we are rolling the transaction back. So, it's much easier to handle errors with Try/Catch construct than with @@Error system function.

2. Also notice that, in the scope of the CATCH block, there are several system functions, that are used to retrieve more information about the error that occurred These functions return NULL if they are executed outside the scope of the CATCH block.

3. TRY/CATCH cannot be used in a user-defined functions.

```
Create Procedure spSellProduct
@ProductId int,
@QuantityToSell int
as
Begin
 -- Check the stock available, for the product we want to sell
 Declare @StockAvailable int
 Select @StockAvailable = QtyAvailable
 from tblProduct where ProductId = @ProductId

 -- Throw an error to the calling application, if enough stock is not available
 if(@StockAvailable < @QuantityToSell)
   Begin
  Raiserror('Not enough stock available',16,1)
   End
 -- If enough stock available
 Else
   Begin
    Begin Try
     Begin Transaction
         -- First reduce the quantity available
  Update tblProduct set QtyAvailable = (QtyAvailable - @QuantityToSell)
  where ProductId = @ProductId
  Declare @MaxProductSalesId int
  -- Calculate MAX ProductSalesId
  Select @MaxProductSalesId = Case When
          MAX(ProductSalesId) IS NULL
          Then 0 else MAX(ProductSalesId) end
         from tblProductSales
  --Increment @MaxProductSalesId by 1, so we don't get a primary key violation
  Set @MaxProductSalesId = @MaxProductSalesId + 1
  Insert into tblProductSales values(@MaxProductSalesId, @ProductId, @QuantityToSell)
     Commit Transaction
    End Try
    Begin Catch
  Rollback Transaction
  Select
   ERROR_NUMBER() as ErrorNumber,
   ERROR_MESSAGE() as ErrorMessage,
   ERROR_PROCEDURE() as ErrorProcedure,
   ERROR_STATE() as ErrorState,
   ERROR_SEVERITY() as ErrorSeverity,
   ERROR_LINE() as ErrorLine
    End Catch
   End
End
```

Transactions in SQL Server - Part 57

What is a Transaction?

A transaction is a group of commands that change the data stored in a database. A transaction, is treated as a single unit. A transaction ensures that, either all of the commands succeed, or none of them. If one of the commands in the transaction fails, all of the commands fail, and any data that was modified in the database is rolled back. In this way, transactions maintain the integrity of data in a database.

Transaction processing follows these steps:

```
1. Begin a transaction.
2. Process database commands.
3. Check for errors.
   If errors occurred,
       rollback the transaction,
   else,
       commit the transaction
```

Let's understand transaction processing with an example. For this purpose, let's Create and populate, tblMailingAddress and tblPhysicalAddress tables

```
Create Table tblMailingAddress
(
  AddressId int NOT NULL primary key,
  EmployeeNumber int,
  HouseNumber nvarchar(50),
  StreetAddress nvarchar(50),
  City nvarchar(10),
  PostalCode nvarchar(50)
)

Insert into tblMailingAddress values (1, 101, '#10', 'King Street', 'Londoon', 'CR27DW')



Create Table tblPhysicalAddress
(
 AddressId int NOT NULL primary key,
 EmployeeNumber int,
 HouseNumber nvarchar(50),
 StreetAddress nvarchar(50),
 City nvarchar(10),
 PostalCode nvarchar(50)
)

Insert into tblPhysicalAddress values (1, 101, '#10', 'King Street', 'Londoon', 'CR27DW')
```

An employee with EmployeeNumber 101, has the same address as his physical and mailing address. His city name is mis-spelled as **Londoon** instead of **London**. The following stored procedure **'spUpdateAddress'**, updates the physical and mailing addresses. Both the UPDATE statements are wrapped between **BEGIN TRANSACTION** and **COMMIT TRANSACTION** block, which in turn is wrapped between **BEGIN TRY** and **END TRY** block.

So, if both the UPDATE statements succeed, without any errors, then the transaction is committed. If there are errors, then the control is immediately transferred to the catch block. The **ROLLBACK TRANSACTION** statement, in the CATCH block, rolls back the transaction, and any data that was written to the database by the commands is backed out.

```
Create Procedure spUpdateAddress
as
Begin
 Begin Try
  Begin Transaction
   Update tblMailingAddress set City = 'LONDON'
   where AddressId = 1 and EmployeeNumber = 101

   Update tblPhysicalAddress set City = 'LONDON'
   where AddressId = 1 and EmployeeNumber = 101
  Commit Transaction
 End Try
 Begin Catch
  Rollback Transaction
 End Catch
End
```

Let's now make the second UPDATE statement, fail. CITY column length in tblPhysicalAddress table is 10. The second UPDATE statement fails, because the value for CITY column is more than 10 characters.

```
Alter Procedure spUpdateAddress
as
Begin
 Begin Try
  Begin Transaction
   Update tblMailingAddress set City = 'LONDON12'
   where AddressId = 1 and EmployeeNumber = 101

   Update tblPhysicalAddress set City = 'LONDON LONDON'
   where AddressId = 1 and EmployeeNumber = 101
  Commit Transaction
 End Try
 Begin Catch
  Rollback Transaction
 End Catch
End
```

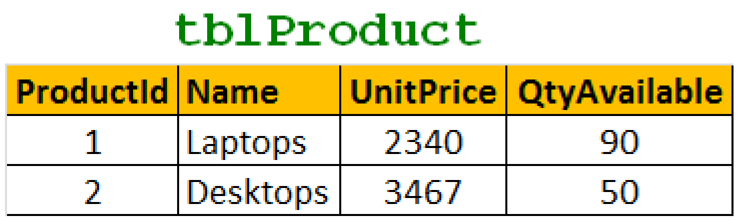
Now, if we execute spUpdateAddress, the first **UPDATE** statements succeeds, but the second **UPDATE** statement fails. As, soon as the second UPDATE statement fails, the control is immediately transferred to the CATCH block. The CATCH block rolls the transaction back. So, the change made by the first UPDATE statement is undone.

Transaction Acid Tests - Part 58

Suggested SQL Server Videos
Part 57 - Transactions in SQL Server

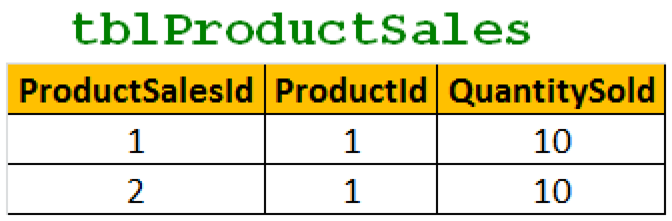
A transaction is a group of database commands that are treated as a single unit. A successful transaction must pass the "ACID" test, that is, it must be
A - Atomic
C - Consistent
I - Isolated
D - Durable

Atomic - All statements in the transaction either completed successfully or they were all rolled back. The task that the set of operations represents is either accomplished or not, but in any case not left half-done. For example, in the **spUpdateInventory_and_Sell** stored procedure, both the UPDATE statements, should succeed. If one UPDATE statement succeeds and the other UPDATE statement fails, the database should undo the change made by the first UPDATE statement, by rolling it back. In short, the transaction should be ATOMIC.



tblProduct

ProductId	Name	UnitPrice	QtyAvailable
1	Laptops	2340	90
2	Desktops	3467	50



tblProductSales

ProductSalesId	ProductId	QuantitySold
1	1	10
2	1	10

```
Create Procedure spUpdateInventory_and_Sell
as
Begin
  Begin Try
    Begin Transaction
      Update tblProduct set QtyAvailable = (QtyAvailable - 10)
      where ProductId = 1

      Insert into tblProductSales values(3, 1, 10)
    Commit Transaction
  End Try
  Begin Catch
    Rollback Transaction
  End Catch
End
```

Consistent - All data touched by the transaction is left in a **logically consistent state**. For example, if stock available numbers are decremented from **tblProductTable**, then, there has to be a related entry in **tblProductSales** table. The inventory can't just disappear.

Isolated - The transaction must affect data without interfering with other concurrent transactions, or being interfered with by them. This prevents transactions from making changes to data based on uncommitted information, for example changes to a record that are subsequently rolled back. **Most databases use locking to maintain transaction isolation**.

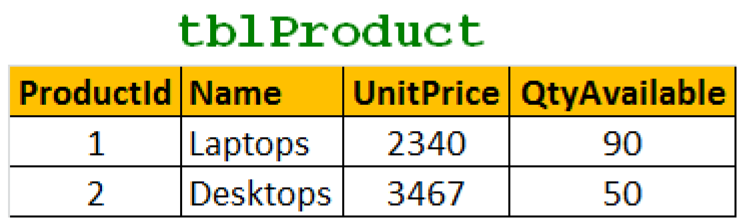
Durable - Once a change is made, it is permanent. If a system error or power failure occurs before a set of commands is complete, those commands are undone and the data is restored to its original state once the system begins running again.
Email This

Transaction Acid Tests - Part 58

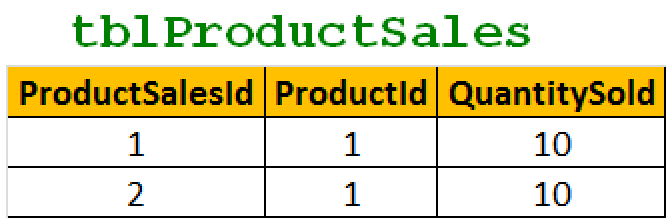
Suggested SQL Server Videos
Part 57 - Transactions in SQL Server

A transaction is a group of database commands that are treated as a single unit. A successful transaction must pass the "ACID" test, that is, it must be
A - Atomic
C - Consistent
I - Isolated
D - Durable

Atomic - All statements in the transaction either completed successfully or they were all rolled back. The task that the set of operations represents is either accomplished or not, but in any case not left half-done. For example, in the **spUpdateInventory_and_Sell** stored procedure, both the UPDATE statements, should succeed. If one UPDATE statement succeeds and the other UPDATE statement fails, the database should undo the change made by the first UPDATE statement, by rolling it back. In short, the transaction should be ATOMIC.


tblProduct

ProductId	Name	UnitPrice	QtyAvailable
1	Laptops	2340	90
2	Desktops	3467	50


tblProductSales

ProductSalesId	ProductId	QuantitySold
1	1	10
2	1	10

```
Create Procedure spUpdateInventory_and_Sell
as
Begin
  Begin Try
    Begin Transaction
      Update tblProduct set QtyAvailable = (QtyAvailable - 10)
      where ProductId = 1

      Insert into tblProductSales values(3, 1, 10)
    Commit Transaction
  End Try
  Begin Catch
    Rollback Transaction
  End Catch
End
```

Consistent - All data touched by the transaction is left in a **logically consistent state**. For example, if stock available numbers are decremented from **tblProductTable**, then, there has to be a related entry in **tblProductSales** table. The inventory can't just disappear.

Isolated - The transaction must affect data without interfering with other concurrent transactions, or being interfered with by them. This prevents transactions from making changes to data based on uncommitted information, for example changes to a record that are subsequently rolled back. **Most databases use locking to maintain transaction isolation**.

Durable - Once a change is made, it is permanent. If a system error or power failure occurs before a set of commands is complete, those commands are undone and the data is restored to its original state once the system begins running again.

Subqueries in sql - Part 59

Suggested Videos
Part 56 - Error handling in sql server 2005, 2008
Part 57 - Transactions
Part 58 - Transaction Acid Tests

In this video we will discuss about subqueries in sql server. Let us understand subqueris with an example. Please create the required tables and insert sample data using the script below.

```
Create Table tblProducts
(
 [Id] int identity primary key,
 [Name] nvarchar(50),
 [Description] nvarchar(250)
)

Create Table tblProductSales
(
 Id int primary key identity,
 ProductId int foreign key references tblProducts(Id),
 UnitPrice int,
 QuantitySold int
)

Insert into tblProducts values ('TV', '52 inch black color LCD TV')
Insert into tblProducts values ('Laptop', 'Very thin black color acer laptop')
Insert into tblProducts values ('Desktop', 'HP high performance desktop')

Insert into tblProductSales values(3, 450, 5)
Insert into tblProductSales values(2, 250, 7)
Insert into tblProductSales values(3, 450, 4)
Insert into tblProductSales values(3, 450, 9)
```

Write a query to retrieve products that are not at all sold?
This can be very easily achieved using subquery as shown below. Select [Id], [Name],

```
[Description]
from tblProducts
where Id not in (Select Distinct ProductId from tblProductSales)
```

Most of the times subqueries can be very easily replaced with joins. The above query is rewritten using joins and produces the same results.

```
Select tblProducts.[Id], [Name],
[Description]
from tblProducts
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
where tblProductSales.ProductId IS NULL
```

In this example, we have seen how to use a subquery in the where clause.

Let us now discuss about using a sub query in the SELECT clause. Write a query to retrieve the NAME and TOTALQUANTITY sold, using a subquery.

```
Select [Name],

(Select SUM(QuantitySold) from tblProductSales where ProductId =
tblProducts.Id) as TotalQuantity
from tblProducts
order by Name
```

Query with an equivalent join that produces the same result.

```
Select [Name], SUM(QuantitySold) as TotalQuantity
from tblProducts
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
group by [Name]
order by Name
```

From these examples, it should be very clear that, a subquery is simply a select statement, that returns a single value and can be nested inside a SELECT, UPDATE, INSERT, or DELETE statement.

It is also possible to nest a subquery inside another subquery.

According to MSDN, subqueries can be nested upto 32 levels.

Subqueries are always encolsed in paranthesis and are also called as inner queries, and the query containing the subquery is called as outer query.

The columns from a table that is present only inside a subquery, cannot be used in the SELECT list of the outer query.

Next Video:
What to choose for performance? Queries that involve a subquery or a join

Correlated subquery in sql - Part 60

Suggested Videos
Part 57 - Transactions
Part 58 - Transaction Acid Tests
Part 59 - Subqueries

In this video we will discuss about Corelated Subqueries

In Part 59, we discussed about 2 examples that uses subqueries. Please watch Part 59, before proceeding with this video. We will be using the same tables and queries from Part 59.

In the example below, sub query is executed first and only once. The sub query results are then used by the outer query. A non-corelated subquery can be executed independently of the outer query.

```
Select [Id], [Name], [Description]
from tblProducts
where Id not in (Select Distinct ProductId from tblProductSales)
```

If the subquery depends on the outer query for its values, then that sub query is called as a correlated subquery. In the where clause of the subquery below, **"ProductId"** column get it's value from **tblProducts** table that is present in the outer query. So, here the subquery is dependent on the outer query for it's value, hence this subquery is a correlated subquery. Correlated subqueries get executed, once for every row that is selected by the outer query. Corelated subquery, cannot be executed independently of the outer query.

```
Select [Name],
(Select SUM(QuantitySold) from tblProductSales where ProductId =
tblProducts.Id) as TotalQuantity
from tblProducts
order by Name
```

Correlated subquery in sql - Part 60

Suggested Videos
Part 57 - Transactions
Part 58 - Transaction Acid Tests
Part 59 - Subqueries

In this video we will discuss about Corelated Subqueries

In Part 59, we discussed about 2 examples that uses subqueries. Please watch Part 59, before proceeding with this video. We will be using the same tables and queries from Part 59.

In the example below, sub query is executed first and only once. The sub query results are then used by the outer query. A non-corelated subquery can be executed independently of the outer query.

```
Select [Id], [Name], [Description]
from tblProducts
where Id not in (Select Distinct ProductId from tblProductSales)
```

If the subquery depends on the outer query for its values, then that sub query is called as a correlated subquery. In the where clause of the subquery below, **"ProductId"** column get it's value from **tblProducts** table that is present in the outer query. So, here the subquery is dependent on the outer query for it's value, hence this subquery is a correlated subquery. Correlated subqueries get executed, once for every row that is selected by the outer query. Corelated subquery, cannot be executed independently of the outer query.

```
Select [Name],
(Select SUM(QuantitySold) from tblProductSales where ProductId =
tblProducts.Id) as TotalQuantity
from tblProducts
order by Name
```

Creating a large table with random data for performance testing - Part 61

Suggested Videos

Part 58 - Transaction Acid Tests
Part 59 - Subqueries
Part 60 - Correlated subquery

In this video we will discuss about inserting large amount of random data into sql server tables for performance testing.

```
-- If Table exists drop the tables
If (Exists (select *
            from information_schema.tables
            where table_name = 'tblProductSales'))
Begin
 Drop Table tblProductSales
End

If (Exists (select *
            from information_schema.tables
            where table_name = 'tblProducts'))
Begin
 Drop Table tblProducts
End

-- Recreate tables
Create Table tblProducts
(
 [Id] int identity primary key,
 [Name] nvarchar(50),
 [Description] nvarchar(250)
)

Create Table tblProductSales
(
 Id int primary key identity,
 ProductId int foreign key references tblProducts(Id),
 UnitPrice int,
 QuantitySold int
)

--Insert Sample data into tblProducts table
Declare @Id int
Set @Id = 1

While(@Id <= 300000)
Begin
 Insert into tblProducts values('Product - ' + CAST(@Id as nvarchar(20)),
 'Product - ' + CAST(@Id as nvarchar(20)) + ' Description')
 Print @Id
 Set @Id = @Id + 1
End

-- Declare variables to hold a random ProductId,
-- UnitPrice and QuantitySold
declare @RandomProductId int
declare @RandomUnitPrice int
declare @RandomQuantitySold int

-- Declare and set variables to generate a
-- random ProductId between 1 and 100000
declare @UpperLimitForProductId int
declare @LowerLimitForProductId int

set @LowerLimitForProductId = 1
set @UpperLimitForProductId = 100000

-- Declare and set variables to generate a
-- random UnitPrice between 1 and 100
declare @UpperLimitForUnitPrice int
declare @LowerLimitForUnitPrice int

set @LowerLimitForUnitPrice = 1
set @UpperLimitForUnitPrice = 100

-- Declare and set variables to generate a
-- random QuantitySold between 1 and 10
declare @UpperLimitForQuantitySold int
declare @LowerLimitForQuantitySold int

set @LowerLimitForQuantitySold = 1
set @UpperLimitForQuantitySold = 10

--Insert Sample data into tblProductSales table
Declare @Counter int
Set @Counter = 1

While(@Counter <= 450000)
Begin
 select @RandomProductId = Round(((@UpperLimitForProductId - @LowerLimitForProductId)
* Rand() + @LowerLimitForProductId), 0)
 select @RandomUnitPrice = Round(((@UpperLimitForUnitPrice - @LowerLimitForUnitPrice)
* Rand() + @LowerLimitForUnitPrice), 0)

 select @RandomQuantitySold = Round(((@UpperLimitForQuantitySold -
@LowerLimitForQuantitySold) * Rand() + @LowerLimitForQuantitySold), 0)

 Insert into tblProductsales
 values(@RandomProductId, @RandomUnitPrice, @RandomQuantitySold)
 Print @Counter
 Set @Counter = @Counter + 1
End
```

Finally, check the data in the tables using a simple SELECT query to make sure the data has been inserted as expected.

```
Select * from tblProducts
Select * from tblProductSales
```

In our next video, we will be using these tables, for performance testing of queries that uses subqueries and joins.

What to choose for performance - SubQueries or Joins - Part 62

Suggested Videos
Part 59 - Subqueries
Part 60 - Correlated subquery
Part 61 - Creating a large table with random data for performance testing

According to MSDN, in sql server, in most cases, there is usually no performance difference between queries that uses sub-queries and equivalent queries using joins. For example, on my machine I have
400,000 records in tblProducts table
600,000 records in tblProductSales tables

The following query, returns, the list of products that we have sold atleast once. This query is formed using sub-queries. When I execute this query I get 306,199 rows in 6 seconds

```
Select Id, Name, Description
from tblProducts
where ID IN
(
 Select ProductId from tblProductSales
)
```

At this stage please clean the query and execution plan cache using the following T-SQL command.

```
CHECKPOINT;
GO
DBCC DROPCLEANBUFFERS; -- Clears query cache
Go
DBCC FREEPROCCACHE; -- Clears execution plan cache
GO
```

Now, run the query that is formed using joins. Notice that I get the exact same 306,199 rows in 6 seconds.

```
Select distinct tblProducts.Id, Name, Description
from tblProducts
inner join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
```

Please Note: I have used automated sql script to insert huge amounts of this random data. Please watch Part 61 of SQL Server tutorial, in which we have discussed about this automated script.

According to MSDN, in some cases where existence must be checked, a join produces better performance. Otherwise, the nested query must be processed for each result of the outer query. In such cases, a join approach would yield better results.

The following query returns the products that we have not sold at least once. This query is formed using sub-queries. When I execute this query I get 93,801 rows in 3 seconds

```
Select Id, Name, [Description]
from tblProducts
where Not Exists(Select * from tblProductSales where ProductId = tblProducts.Id)
```

When I execute the below equivalent query, that uses joins, I get the exact same 93,801 rows in 3 seconds.

```
Select tblProducts.Id, Name, [Description]
from tblProducts
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
where tblProductSales.ProductId IS NULL
```

In general joins work faster than sub-queries, but in reality it all depends on the execution plan that is generated by SQL Server. It does not matter how we have written the query, SQL Server will always transform it on an execution plan. If sql server generates the same plan from both queries, we will get the same result.

I would say, rather than going by theory, turn on client statistics and execution plan to see the performance of each option, and then make a decision.

In a later video session we will discuss about client statistics and execution plans in detail.

Cursors in sql server - Part 63

Suggested Videos

Part 60 - Correlated subquery
Part 61 - Creating a large table with random data for performance testing
Part 62 - What to choose for performance - SubQuery or Joins

Relational Database Management Systems, including sql server are very good at handling data in SETS. For example, the following "UPDATE" query, updates a set of rows that matches the condition in the "WHERE" clause at the same time.

```
Update tblProductSales Set UnitPrice = 50 where ProductId = 101
```

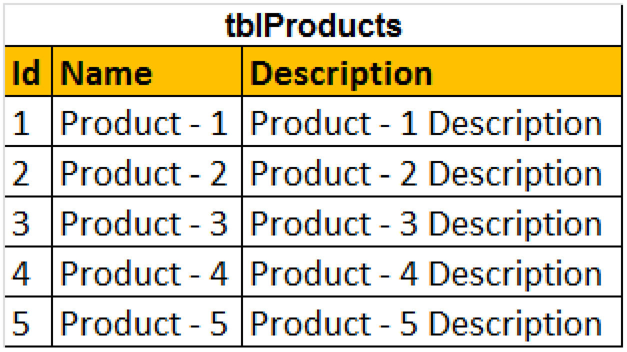
However, if there is ever a need to process the rows, on a row-by-row basis, then cursors are your choice. Cursors are very bad for performance, and should be avoided always. Most of the time, cursors can be very easily replaced using joins.

There are different types of cursors in sql server as listed below. We will talk about the differences between these cursor types in a later video session.

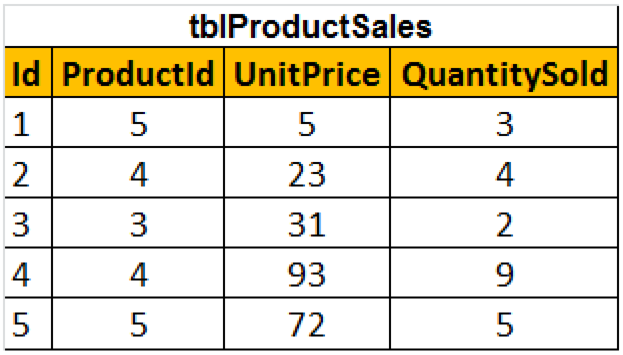
1. Forward-Only

2. Static
3. Keyset
4. Dynamic

Let us now look at a simple example of using sql server cursor to process one row at time. We will be using tblProducts and tblProductSales tables, for this example. The tables here show only 5 rows from each table. However, on my machine, there are 400,000 records in tblProducts and 600,000 records in tblProductSales tables. If you want to learn about generating huge amounts of random test data, **please watch Part - 61 in sql server video tutorial.**



tblProducts		
Id	**Name**	**Description**
1	Product - 1	Product - 1 Description
2	Product - 2	Product - 2 Description
3	Product - 3	Product - 3 Description
4	Product - 4	Product - 4 Description
5	Product - 5	Product - 5 Description



tblProductSales			
Id	**ProductId**	**UnitPrice**	**QuantitySold**
1	5	5	3
2	4	23	4
3	3	31	2
4	4	93	9
5	5	72	5

Cursor Example: Let us say, I want to update the UNITPRICE column in tblProductSales table, based on the following criteria
1. If the ProductName = 'Product - 55', Set Unit Price to 55

2. If the ProductName = 'Product - 65', Set Unit Price to 65
3. If the ProductName is like 'Product - 100%', Set Unit Price to 1000

```
Declare @ProductId int

-- Declare the cursor using the declare keyword
Declare ProductIdCursor CURSOR FOR
Select ProductId from tblProductSales

-- Open statement, executes the SELECT statment
-- and populates the result set
Open ProductIdCursor

-- Fetch the row from the result set into the variable
```

```
Fetch Next from ProductIdCursor into @ProductId

-- If the result set still has rows, @@FETCH_STATUS will be ZERO
While(@@FETCH_STATUS = 0)
Begin
 Declare @ProductName nvarchar(50)
 Select @ProductName = Name from tblProducts where Id = @ProductId

 if(@ProductName = 'Product - 55')
 Begin
  Update tblProductSales set UnitPrice = 55 where ProductId = @ProductId
 End
 else if(@ProductName = 'Product - 65')
 Begin
  Update tblProductSales set UnitPrice = 65 where ProductId = @ProductId
 End
 else if(@ProductName like 'Product - 100%')
 Begin
  Update tblProductSales set UnitPrice = 1000 where ProductId = @ProductId
 End

 Fetch Next from ProductIdCursor into @ProductId
End

-- Release the row set
CLOSE ProductIdCursor
-- Deallocate, the resources associated with the cursor
DEALLOCATE ProductIdCursor
```

The cursor will loop thru each row in tblProductSales table. As there are 600,000 rows, to be processed on a row-by-row basis, it takes around 40 to 45 seconds on my machine. We can achieve this very easily using a join, and this will significantly increase the performance. We will discuss about this in our next video session.

To check if the rows have been correctly updated, please use the following query.

```
Select  Name, UnitPrice
from tblProducts join
tblProductSales on tblProducts.Id = tblProductSales.ProductId
where (Name='Product - 55' or Name='Product - 65' or Name like 'Product - 100%')
```

Replacing cursors using joins in sql server - Part 64

Suggested Videos

Part 61 - Creating a large table with random data for performance testing
Part 62 - What to choose for performance - SubQuery or Joins
Part 63 - Cursors in sql server

In Part 63, we have discussed about cursors. The example, in Part 63, took around 45 seconds on my machine. Please watch Part 63, before proceeding with this video. In this video we will re-write the example, using a join.

```
Update tblProductSales
set UnitPrice =
 Case
  When Name = 'Product - 55' Then 155
  When Name = 'Product - 65' Then 165
  When Name like 'Product - 100%' Then 10001
 End
from tblProductSales
join tblProducts
on tblProducts.Id = tblProductSales.ProductId
Where Name = 'Product - 55' or Name = 'Product - 65' or
Name like 'Product - 100%'
```

When I executed this query, on my machine it took less than a second. Where as the same thing using a cursor took 45 seconds. Just imagine the amount of impact cursors have on performance. Cursors should be used as your last option. Most of the time cursors can be very easily replaced using joins.

To check the result of the UPDATE statement, use the following query.

```
Select  Name, UnitPrice from
tblProducts join
tblProductSales on tblProducts.Id = tblProductSales.ProductId
where (Name='Product - 55' or Name='Product - 65' or
Name like 'Product - 100%')
```

Part 65 - List all tables in a sql server database using a query

Suggested Videos

Part 62 - What to choose for performance - SubQuery or Joins
Part 63 - Cursors in sql server
Part 64 - Replacing cursors using joins in sql server

In this video we will discuss, writing a **transact sql query to list all the tables in a sql server database**. This is a very common sql server interview question.

Object explorer with in sql server management studio can be used to get the list of tables in a specific database. However, if we have to write a query to achieve the same, there are 3 system views that we can use.

1. SYSOBJECTS - Supported in SQL Server version 2000, 2005 & 2008
2. SYS.TABLES - Supported in SQL Server version 2005 & 2008
3. INFORMATION_SCHEMA.TABLES - Supported in SQL Server version 2005 & 2008

```
-- Gets the list of tables only
Select * from SYSOBJECTS where XTYPE='U'
-- Gets the list of tables only
Select * from  SYS.TABLES
-- Gets the list of tables and views
Select * from INFORMATION_SCHEMA.TABLES
```

To get the list of different object types (XTYPE) in a database

```
Select Distinct XTYPE from SYSOBJECTS
```

Executing the above query on my SAMPLE database returned the following values for XTYPE column from SYSOBJECTS

IT - Internal table
P - Stored procedure
PK - PRIMARY KEY constraint
S - System table
SQ - Service queue
U - User table
V - View

Please check the following MSDN link for all possible XTYPE column values and what they represent.
http://msdn.microsoft.com/en-us/library/ms177596.aspx

Writing re-runnable sql server scripts - Part 66

Suggested Videos
Part 63 - Cursors in sql server
Part 64 - Replacing cursors using joins in sql server
Part 65 - List all tables in a sql server database using a query

What is a re-runnable sql script?
A re-runnable script is a script, that, when run more than, once will not throw errors.

Let's understand **writing re-runnable sql scripts** with an example. To create a table **tblEmployee** in **Sample** database, we will write the following **CREATE TABLE** sql script.

```
USE [Sample]
Create table tblEmployee
(
 ID int identity primary key,
 Name nvarchar(100),
 Gender nvarchar(10),
 DateOfBirth DateTime
)
```

When you run this script once, the table **tblEmployee** gets created without any errors. If you run the script again, you will get an error - There is already an object named 'tblEmployee' in the database.

To make this script re-runnable
1. Check for the existence of the table
2. Create the table if it does not exist
3. Else print a message stating, the table already exists

```
Use [Sample]
If not exists (select * from information_schema.tables where table_name = 'tblEmployee')
Begin
 Create table tblEmployee
 (
  ID int identity primary key,
  Name nvarchar(100),
  Gender nvarchar(10),
  DateOfBirth DateTime
 )
 Print 'Table tblEmployee successfully created'
End
Else
Begin
 Print 'Table tblEmployee already exists'
End
```

The above **script is re-runnable**, and can be run any number of times. If the table is not already created, the script will create the table, else you will get a message stating - **The table already exists.** You will never get a sql script error.

Sql server built-in function OBJECT_ID(), can also be used to check for the existence of the table

```
IF OBJECT_ID('tblEmployee') IS NULL
Begin
   -- Create Table Script
   Print 'Table tblEmployee created'
End
Else
Begin
   Print 'Table tblEmployee already exists'
End
```

Depending on what we are trying to achieve, sometime we may need **to drop (if the table already exists) and re-create it**. The sql script below, does exactly the same thing.

```
Use [Sample]
IF OBJECT_ID('tblEmployee') IS NOT NULL
Begin
 Drop Table tblEmployee
End
Create table tblEmployee
(
 ID int identity primary key,
 Name nvarchar(100),
 Gender nvarchar(10),
 DateOfBirth DateTime
)
```

Let's look at another example. The following sql script adds column **"EmailAddress"** to table **tblEmployee**. This script is not re-runnable because, if the column exists we get a script error.

```
Use [Sample]
ALTER TABLE tblEmployee
ADD EmailAddress nvarchar(50)
```

To make this script re-runnable, check for the column existence

```
Use [Sample]
if not
exists(Select * from INFORMATION_SCHEMA.COLUMNS where COLUMN_NAME='EmailAddr
ess' and TABLE_NAME = 'tblEmployee' and TABLE_SCHEMA='dbo')
Begin
```

```
 ALTER TABLE tblEmployee
 ADD EmailAddress nvarchar(50)
End
Else
BEgin
 Print 'Column EmailAddress already exists'
End
```

Col_length() function can also be used to check for the existence of a column

```
If col_length('tblEmployee','EmailAddress') is not null
Begin
 Print 'Column already exists'
End
Else
Begin
 Print 'Column does not exist'
End
```

Part 67 - Alter database table columns without dropping table

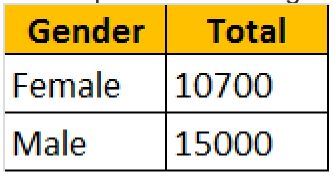
Suggested Videos

Part 64 - Replacing cursors using joins in sql server
Part 65 - List all tables in a sql server database using a query
Part 66 - Writing re-runnable sql server scripts

In this video, we will discuss, **altering a database table column without having the need to drop the table.** Let's understand this with an example.
We will be using table **tblEmployee** for this demo. Use the sql script below, to create and populate this table with some sample data.

```
Create table tblEmployee
(
 ID int primary key identity,
 Name nvarchar(50),
 Gender nvarchar(50),
 Salary nvarchar(50)
)
Insert into tblEmployee values('Sara Nani','Female','4500')
Insert into tblEmployee values('James Histo','Male','5300')
Insert into tblEmployee values('Mary Jane','Female','6200')
Insert into tblEmployee values('Paul Sensit','Male','4200')
Insert into tblEmployee values('Mike Jen','Male','5500')
```

The requirement is to group the salaries by gender. The output should be as shown below.



Gender	Total
Female	10700
Male	15000

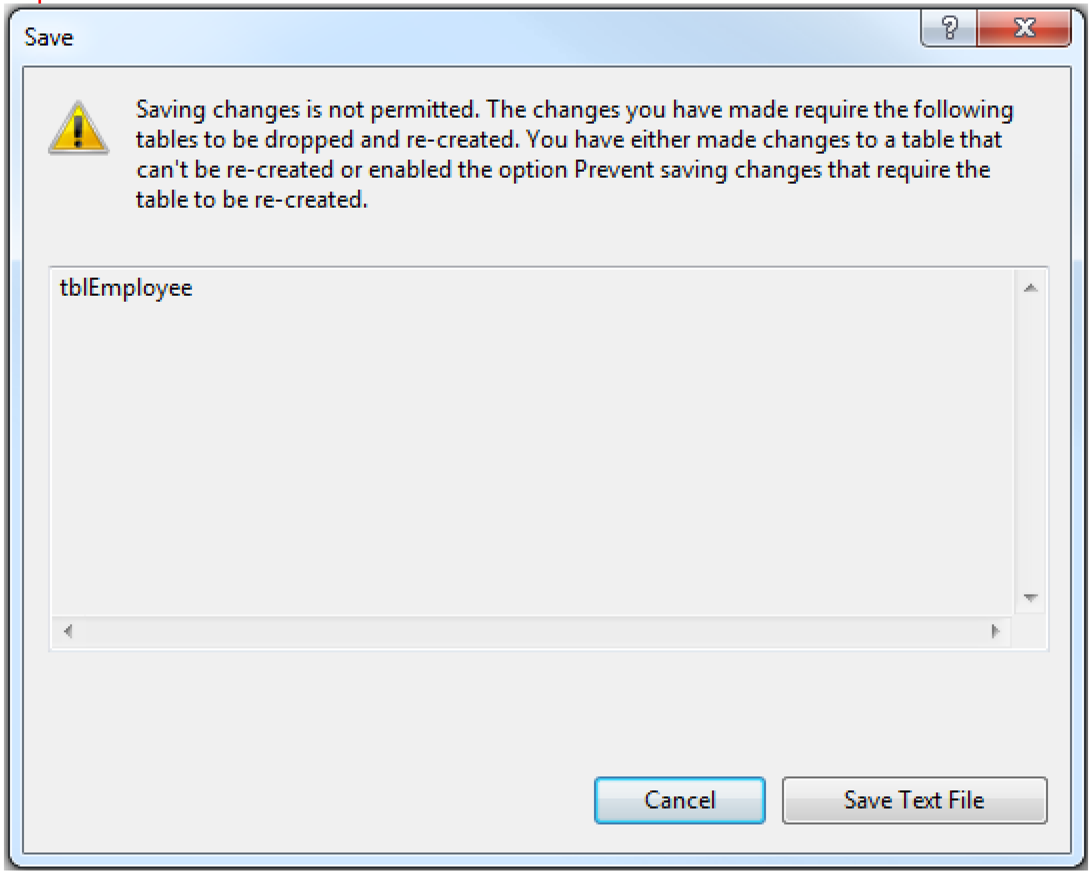
To achieve this we would write a sql query using GROUP BY as shown below.

```
Select Gender, Sum(Salary) as Total
from tblEmployee
```

Group by Gender

When you execute this query, we will get an error - Operand data type nvarchar is invalid for sum operator. This is because, when we created **tblEmployee** table, the **"Salary"** column was created using **nvarchar** datatype. SQL server **Sum**() aggregate function can only be applied on numeric columns. So, let's try to modify **"Salary"** column to use **int** datatype. Let's do it using the designer.
1. Right click on "tblEmployee" table in "Object Explorer" window, and select "Design"
2. Change the datatype from nvarchar(50) to int
3. Save the table

At this point, you will get an error message - Saving changes is not permitted. The changes you have made require the following tables to be dropped and re-created. You have either made changes to a table that can't be re-created or enabled the option Prevent saving changes that require the table to be re-created.



Saving changes is not permitted. The changes you have made require the following tables to be dropped and re-created. You have either made changes to a table that can't be re-created or enabled the option Prevent saving changes that require the table to be re-created.

tblEmployee

So, the obvious next question is, **how to alter the database table definition without the need to drop, re-create and again populate the table with data?**
There are 2 options

Option 1: Use a sql query to alter the column as shown below.
Alter table tblEmployee
Alter column Salary int

Option 2: Disable **"Prevent saving changes that require table re-creation"** option in sql

server 2008
1. Open Microsoft SQL Server Management Studio 2008
2. Click Tools, select Options
3. Expand Designers, and select "Table and Database Designers"
4. On the right hand side window, uncheck, Prevent saving changes that require table re-creation
5. Click OK

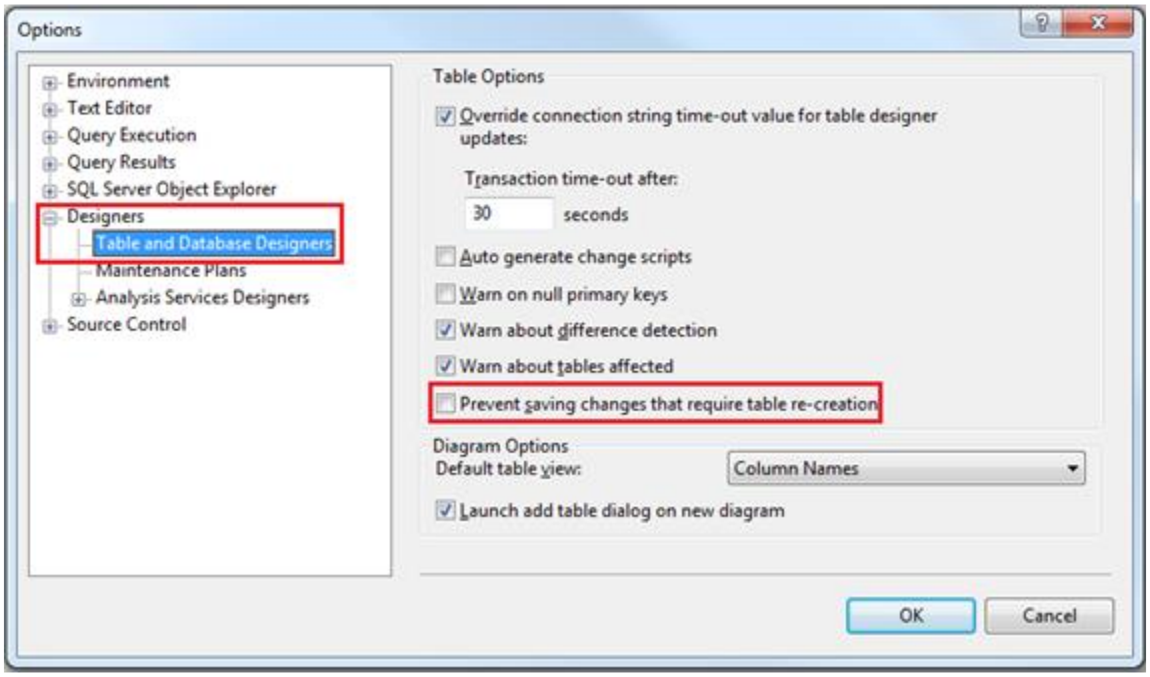


Table Options
- [✓] Override connection string time-out value for table designer updates:
- Transaction time-out after: 30 seconds
- [ ] Auto generate change scripts
- [ ] Warn on null primary keys
- [✓] Warn about difference detection
- [✓] Warn about tables affected
- [ ] Prevent saving changes that require table re-creation

Diagram Options
- Default table view: Column Names
- [✓] Launch add table dialog on new diagram

Part 68 - Optional parameters in sql server stored procedures

Suggested Videos
Part 65 - List all tables in a sql server database using a query
Part 66 - Writing re-runnable sql server scripts
Part 67 - Alter database table columns without dropping table

Parameters of a sql server stored procedure can be made optional by specifying default values.

We wil be using table tblEmployee for this Demo.

```
CREATE TABLE tblEmployee
(
 Id int IDENTITY PRIMARY KEY,
 Name nvarchar(50),
 Email nvarchar(50),
 Age int,
 Gender nvarchar(50),
 HireDate date,
)
Insert into tblEmployee values
('Sara Nan','Sara.Nan@test.com',35,'Female','1999-04-04')
```

```
Insert into tblEmployee values
('James Histo','James.Histo@test.com',33,'Male','2008-07-13')
Insert into tblEmployee values
('Mary Jane','Mary.Jane@test.com',28,'Female','2005-11-11')
Insert into tblEmployee values
('Paul Sensit','Paul.Sensit@test.com',29,'Male','2007-10-23')
```

Name, Email, Age and Gender parameters of spSearchEmployees stored procedure are optional. Notice that, we have set defaults for all the parameters, and in the "WHERE" clause we are checking if the respective parameter IS NULL.

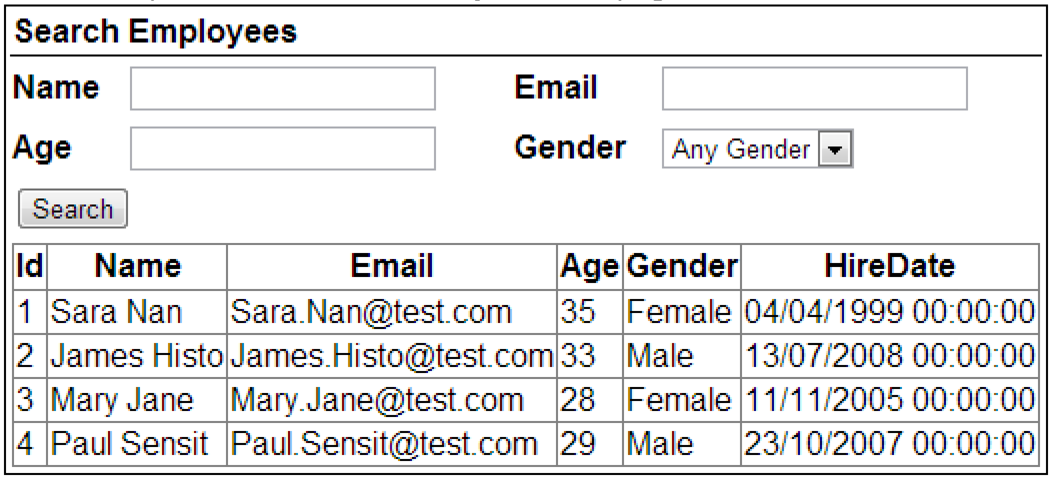
```
Create Proc spSearchEmployees
@Name nvarchar(50) = NULL,
@Email nvarchar(50) = NULL,
@Age int = NULL,
@Gender nvarchar(50) = NULL
as
Begin
 Select * from tblEmployee where
 (Name = @Name OR @Name IS NULL) AND
 (Email = @Email OR @Email IS NULL) AND
 (Age = @Age OR @Age IS NULL) AND
 (Gender = @Gender OR @Gender IS NULL)
End
```

Testing the stored procedure

1. Execute spSearchEmployees -- This command will return all the rows
2. Execute spSearchEmployees @Gender = 'Male' -- Retruns only Male employees

3. Execute spSearchEmployees @Gender = 'Male', @Age = 29 -- Retruns Male employees whose age is 29

This stored procedure can be used by a search page that looks as shown below.



Id	Name	Email	Age	Gender	HireDate
1	Sara Nan	Sara.Nan@test.com	35	Female	04/04/1999 00:00:00
2	James Histo	James.Histo@test.com	33	Male	13/07/2008 00:00:00
3	Mary Jane	Mary.Jane@test.com	28	Female	11/11/2005 00:00:00
4	Paul Sensit	Paul.Sensit@test.com	29	Male	23/10/2007 00:00:00

WebForm1.aspx:

```
<table style="font-family:Arial; border:1px solid black">
    <tr>
        <td colspan="4" style="border-bottom: 1px solid black">
            <b>Search Employees</b>
        </td>
    </tr>
    <tr>
        <td>
            <b>Name</b>
        </td>
        <td>
            <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
        </td>
        <td>
            <b>Email</b>
        </td>
        <td>
            <asp:TextBox ID="txtEmail" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            <b>Age</b>
        </td>
        <td>
            <asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
        </td>
        <td>
            <b>Gender</b>
        </td>
        <td>
            <asp:DropDownList ID="ddlGender" runat="server">
                <asp:ListItem Text="Any Gender" Value="-1"></asp:ListItem>
                <asp:ListItem Text="Male" Value="Male"></asp:ListItem>

                <asp:ListItem Text="Female" Value="Female"></asp:ListItem>
            </asp:DropDownList>
        </td>
    </tr>
    <tr>
        <td colspan="4">
            <asp:Button ID="btnSerach" runat="server" Text="Search"
                onclick="btnSerach_Click" />
        </td>
    </tr>
    <tr>
        <td colspan="4">
            <asp:GridView ID="gvEmployees" runat="server">
            </asp:GridView>
        </td>
    </tr>
</table>
```

WebForm1.aspx.cs:

```
public partial class WebForm1 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            GetData();
        }
    }

    protected void btnSerach_Click(object sender, EventArgs e)
    {
        GetData();
    }

    private void GetData()
    {
        string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
        using (SqlConnection con = new SqlConnection(cs))
        {
            SqlCommand cmd = new SqlCommand("spSearchEmployees", con);
            cmd.CommandType = CommandType.StoredProcedure;

            AttachParameter(cmd, "@Name", txtName);
            AttachParameter(cmd, "@Email", txtEmail);
            AttachParameter(cmd, "@Age", txtAge);
            AttachParameter(cmd, "@Gender", ddlGender);

            con.Open();
            gvEmployees.DataSource = cmd.ExecuteReader();
            gvEmployees.DataBind();
        }
    }

    private
void AttachParameter(SqlCommand command, string parameterName, Control control)
    {
        if (control is TextBox && ((TextBox)control).Text != string.Empty)
        {
            SqlParameter parameter = new SqlParameter(parameterName,
((TextBox)control).Text);
            command.Parameters.Add(parameter);
        }
        else if (control is DropDownList && ((DropDownList)control).SelectedValue != "-1")
        {
            SqlParameter parameter = new SqlParameter parameterName,
((DropDownList)control).SelectedValue);
            command.Parameters.Add(parameter);
        }
    }
}
```

Make sure you have the following using statements in your code-behind page

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
```

Part 69 - Merge in SQL Server

Suggested Videos

Part 66 - Writing re-runnable sql server scripts
Part 67 - Alter database table columns without dropping table
Part 68 - Optional parameters in sql server stored procedures

What is the use of MERGE statement in SQL Server

Merge statement introduced in SQL Server 2008 allows us to perform Inserts, Updates and Deletes in one statement. This means we no longer have to use multiple statements for performing Insert, Update and Delete.

With merge statement we require 2 tables

1. Source Table - Contains the changes that needs to be applied to the target table
2. Target Table - The table that require changes (Inserts, Updates and Deletes)

The merge statement joins the target table to the source table by using a common column in both the tables. Based on how the rows match up as a result of the join, we can then perform insert, update, and delete on the target table.

Merge statement syntax

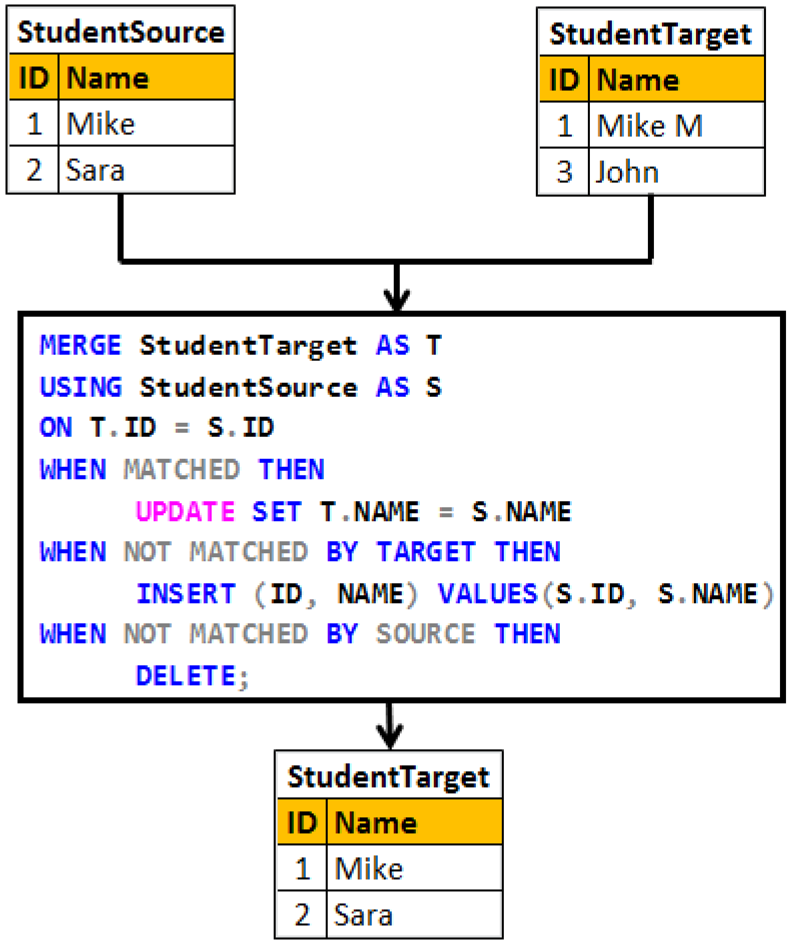
```
MERGE [TARGET] AS T
USING [SOURCE] AS S
   ON [JOIN_CONDITIONS]
 WHEN MATCHED THEN
      [UPDATE STATEMENT]
 WHEN NOT MATCHED BY TARGET THEN
      [INSERT STATEMENT]
 WHEN NOT MATCHED BY SOURCE THEN
      [DELETE STATEMENT]
```

Example 1 : In the example below, INSERT, UPDATE and DELETE are all performed in one statement

1. When matching rows are found, StudentTarget table is UPDATED (i.e WHEN MATCHED)

2. When the rows are present in StudentSource table but not in StudentTarget table those rows are INSERTED into StudentTarget table (i.e WHEN NOT MATCHED BY TARGET)

3. When the rows are present in StudentTarget table but not in StudentSource table those rows are DELETED from StudentTarget table (i.e WHEN NOT MATCHED BY SOURCE)



StudentSource	
ID	Name
1	Mike
2	Sara

StudentTarget	
ID	Name
1	Mike M
3	John

```
MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
      UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
      INSERT (ID, NAME) VALUES(S.ID, S.NAME)
WHEN NOT MATCHED BY SOURCE THEN
      DELETE;
```

StudentTarget	
ID	Name
1	Mike
2	Sara

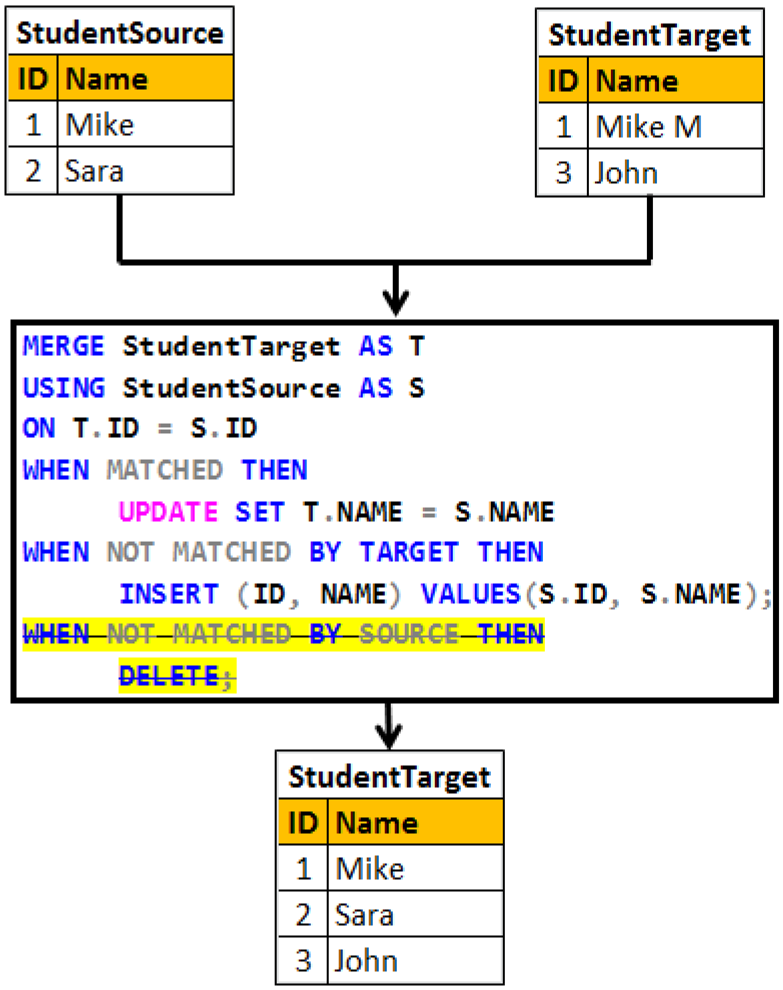
```
Create table StudentSource
(
    ID int primary key,
    Name nvarchar(20)
)
GO

Insert into StudentSource values (1, 'Mike')
Insert into StudentSource values (2, 'Sara')
GO

Create table StudentTarget
(
    ID int primary key,
    Name nvarchar(20)
)
GO
```

```
Insert into StudentTarget values (1, 'Mike M')
Insert into StudentTarget values (3, 'John')
GO

MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, NAME) VALUES(S.ID, S.NAME)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Please Note : Merge statement should end with a semicolon, otherwise you would get an error stating - A MERGE statement must be terminated by a semi-colon (;)

In real time we mostly perform INSERTS and UPDATES. The rows that are present in target table but not in source table are usually not deleted from the target table.

Example 2 : In the example below, only INSERT and UPDATE is performed. We are not deleting the rows that are present in the target table but not in the source table.



StudentSource

ID	Name
1	Mike
2	Sara

StudentTarget

ID	Name
1	Mike M
3	John

```
MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
      UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
      INSERT (ID, NAME) VALUES(S.ID, S.NAME);
~~WHEN NOT MATCHED BY SOURCE THEN~~
      ~~DELETE;~~
```

StudentTarget

ID	Name
1	Mike
2	Sara
3	John

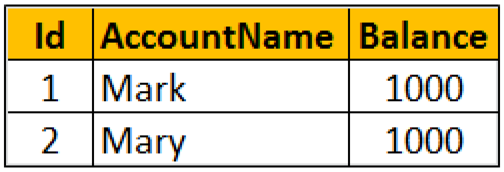
```
Truncate table StudentSource
Truncate table StudentTarget
GO

Insert into StudentSource values (1, 'Mike')
Insert into StudentSource values (2, 'Sara')
GO

Insert into StudentTarget values (1, 'Mike M')
Insert into StudentTarget values (3, 'John')
GO

MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, NAME) VALUES(S.ID, S.NAME);
```

sql server concurrent transactions

Suggested Videos
Part 67 - Alter database table columns without dropping table
Part 68 - Optional parameters in sql server stored procedures
Part 69 - Merge in SQL Server

In this video we will discuss
1. What a transaction is
2. The problems that might arise when tarnsactions are run concurrently
3. The different transaction isolation levels provided by SQL Server to address concurrency side effects

What is a transaction
A transaction is a group of commands that change the data stored in a database. A transaction, is treated as a single unit of work. A transaction ensures that, either all of the commands succeed, or none of them. If one of the commands in the transaction fails, all of the commands fail, and any data that was modified in the database is rolled back. In this way, transactions maintain the integrity of data in a database.



Id	AccountName	Balance
1	Mark	1000
2	Mary	1000

Example : The following transaction ensures that both the UPDATE statements succeed or both of them fail if there is a problem with one UPDATE statement.

```
-- Transfer $100 from Mark to Mary Account
BEGIN TRY
   BEGIN TRANSACTION
       UPDATE Accounts SET Balance = Balance - 100 WHERE Id = 1
       UPDATE Accounts SET Balance = Balance + 100 WHERE Id = 2
   COMMIT TRANSACTION
   PRINT 'Transaction Committed'
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
   PRINT 'Transaction Rolled back'
END CATCH
```

Databases are powerful systems and are potentially used by many users or applications at the same time. Allowing concurrent transactions is essential for performance but may introduce concurrency issues when two or more transactions are working with the same data at the same time.

Some of the common concurrency problems

- Dirty Reads
- Lost Updates
- Nonrepeatable Reads
- Phantom Reads

We will discuss what these problems are in detail with examples in our upcomning videos

One way to solve all these concurrency problems is by allowing only one user to execute, only one transaction at any point in time. Imagine what could happen if you have a large database with several users who want to execute several transactions. All the transactions get queued and they may have to wait a long time before they could get a chance to execute their transactions. So you are getting poor performance and the whole purpose of having a powerful database system is defeated if you serialize access this way.

At this point you might be thinking, for best performance let us allow all transactions to execute concurrently. The problem with this approach is that it may cause all sorts of concurrency problems (i.e Dirty Reads, Lost Updates, Nonrepeatable Reads, Phantom Reads) if two or more transactions work with the same data at the same time.

SQL Server provides different **transaction isolation levels**, to balance concurrency problems and performance depending on our application needs.

- Read Uncommitted
- Read Committed
- Repeatable Read
- Snapshot
- Serializable

The isolation level that you choose for your transaction, defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Depending on the isolation level you have chosen you get varying degrees of performance and concurrency problems. The table here has the list of isoltaion levels along with concurrency side effects.

Isolation Level	**Dirty Reads**	**Lost Update**	**Nonrepeatable Reads**	**Phantom Reads**
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

If you choose the lowest isolation level (i.e Read Uncommitted), it increases the number of concurrent transactions that can be executed at the same time, but the down side is you have all sorts of concurrency issues. On the other hand if you choose the highest isolation level (i.e Serializable), you will have no concurrency side effects, but the downside is that, this will reduce the number of concurrent transactions that can be executed at the same time if those transactions work with same data.

In our upcoming videos we will discuss the concurrency problems in detail with examples

sql server dirty read example

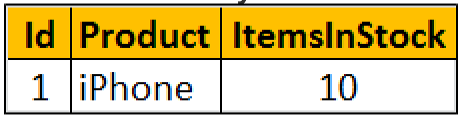
Suggested Videos
Part 68 - Optional parameters in sql server stored procedures
Part 69 - Merge in SQL Server
Part 70 - sql server concurrent transactions

In this video we will discuss, **dirty read concurrency problem** with an example. This is continuation to Part 70. Please watch Part 70 from SQL Server tutorial for beginners.

A dirty read happens when one transaction is permitted to read data that has been modified by another transaction that has not yet been committed. In most cases this would not cause a problem. However, if the first transaction is rolled back after the second reads the data, the second transaction has dirty data that does not exist anymore.

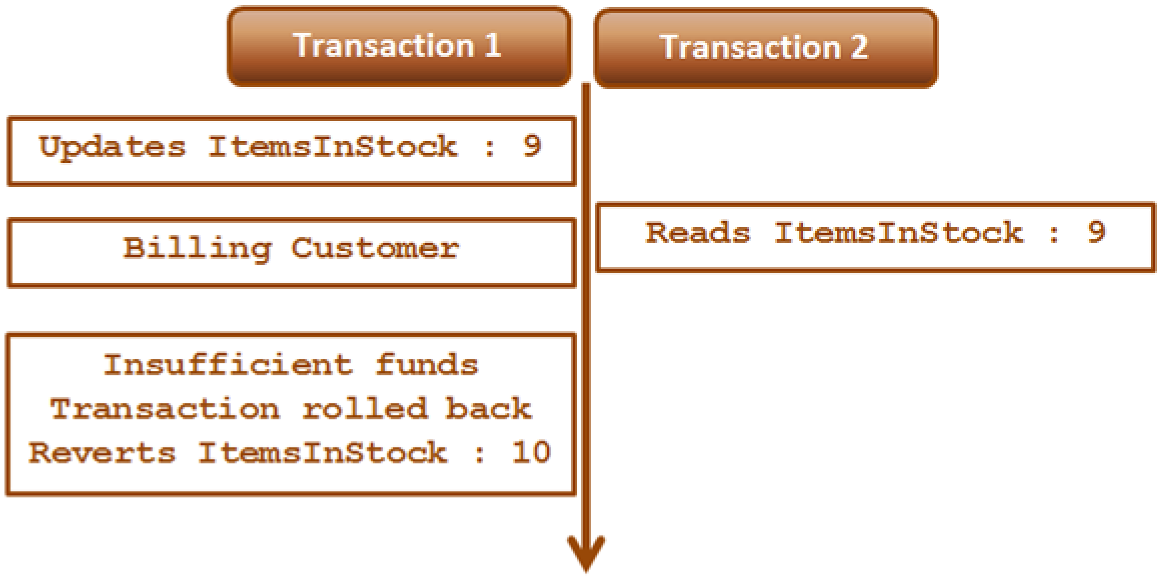
SQL script to create table tblInventory

```
Create table tblInventory
(
   Id int identity primary key,
   Product nvarchar(100),
   ItemsInStock int
)
Go

Insert into tblInventory values ('iPhone', 10)
```

Table tblInventory


Id	Product	ItemsInStock
1	iPhone	10

Dirty Read Example : In the example below, Transaction 1, updates the value of ItemsInStock to 9. Then it starts to bill the customer. While Transaction 1 is still in progress, Transaction 2 starts and reads ItemsInStock value which is 9 at the moment. At this point, Transaction 1 fails because of insufficient funds and is rolled back. The ItemsInStock is reverted to the original value of 10, but Transaction 2 is working with a different value (i.e 10).



Transaction 1 :

```
Begin Tran
Update tblInventory set ItemsInStock = 9 where Id=1

-- Billing the customer
Waitfor Delay '00:00:15'
-- Insufficient Funds. Rollback transaction

Rollback Transaction
```

Transaction 2 :

```
Set Transaction Isolation Level Read Uncommitted
Select * from tblInventory where Id=1
```

Read Uncommitted transaction isolation level is the only isolation level that has dirty read side effect. This is the least restrictive of all the isolation levels. When this transaction isolation level is set, it is possible to read uncommitted or dirty data. Another option to read dirty data is by using NOLOCK table hint. The query below is equivalent to the query in Transaction 2.

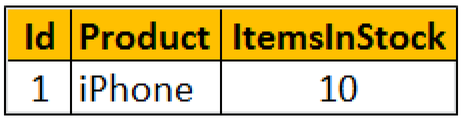
```
Select * from tblInventory (NOLOCK) where Id=1
```

sql server lost update problem

Suggested Videos
Part 69 - Merge in SQL Server
Part 70 - sql server concurrent transactions
Part 71 - sql server dirty read example

In this video we will discuss, **lost update problem in sql server** with an example.

Lost update problem happens when 2 transactions read and update the same data. Let's understand this with an example. We will use the following table **tblInventory** for this example.

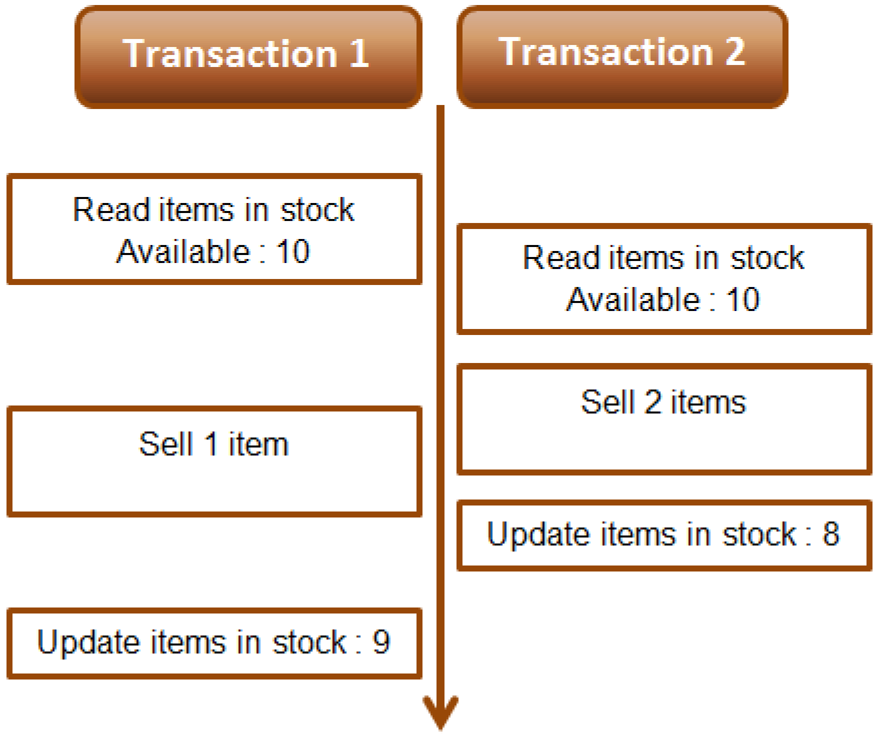


Id	Product	ItemsInStock
1	iPhone	10

As you can see in the diagram below there are 2 transactions - Transaction 1 and Transaction 2. Transaction 1 starts first, and it is processing an order for 1 iPhone. It sees ItemsInStock as 10.

At this time Transaction 2 is processing another order for 2 iPhones. It also sees ItemsInStock as 10. Transaction 2 makes the sale first and updates ItemsInStock with a value of 8.

At this point Transaction 1 completes the sale and silently overwrites the update of Transaction 2. As Transaction 1 sold 1 iPhone it has updated ItemsInStock to 9, while it actually should have updated it to 7.

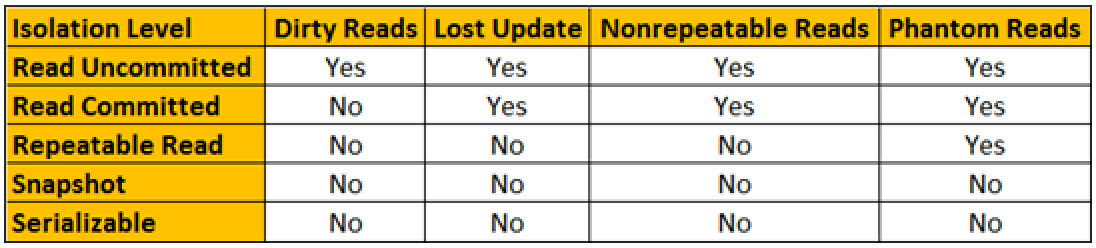

Transaction 1
Transaction 2
Read items in stock Available : 10
Read items in stock Available : 10
Sell 2 items
Sell 1 item
Update items in stock : 8
Update items in stock : 9

Example : The lost update problem example. Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code. Transaction 1 is processing an order for 1 iPhone, while Transaction 2 is processing an order for 2 iPhones. At the end of both the transactions ItemsInStock must be 7, but we have a value of 9. This is because Transaction 1 silently overwrites the update of Transaction 2. This is called the **lost update problem**.

```
-- Transaction 1
Begin Tran
Declare @ItemsInStock int

Select @ItemsInStock = ItemsInStock
```

```
from tblInventory where Id=1

-- Transaction takes 10 seconds
Waitfor Delay '00:00:10'
Set @ItemsInStock = @ItemsInStock - 1

Update tblInventory
Set ItemsInStock = @ItemsInStock where Id=1

Print @ItemsInStock

Commit Transaction


-- Transaction 2
Begin Tran
Declare @ItemsInStock int

Select @ItemsInStock = ItemsInStock
from tblInventory where Id=1

-- Transaction takes 1 second
Waitfor Delay '00:00:1'
Set @ItemsInStock = @ItemsInStock - 2

Update tblInventory
Set ItemsInStock = @ItemsInStock where Id=1

Print @ItemsInStock

Commit Transaction
```

Both Read Uncommitted and Read Committed transaction isolation levels have the lost update side effect. Repeatable Read, Snapshot, and Serializable isolation levels does not have this side effect. If you run the above Transactions using any of the higher isolation levels (Repeatable Read, Snapshot, or Serializable) you will not have lost update problem. The repeatable read isolation level uses additional locking on rows that are read by the current transaction, and prevents them from being updated or deleted elsewhere. This solves the lost update problem.



Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

For both the above transactions, set Repeatable Read Isolation Level. Run Transaction 1 first and then a few seconds later run Transaction 2. Transaction 1 completes successfully, but

Transaction 2 competes with the following error.
Transaction was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

Once you rerun Transaction 2, ItemsInStock will be updated correctly as expected.

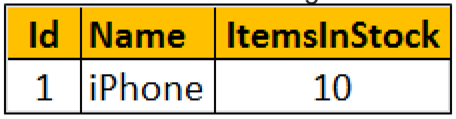
Non repeatable read example in sql server

Suggested Videos
Part 70 - sql server concurrent transactions
Part 71 - sql server dirty read example
Part 72 - sql server lost update problem

In this video we will discuss **non repeatable read concurrency problem with an example**.

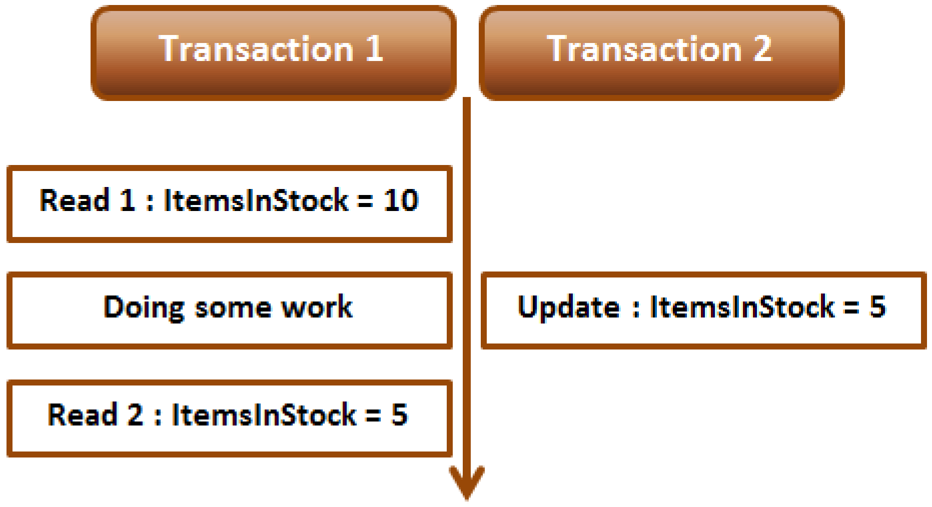
Non repeatable read problem happens when one transaction reads the same data twice and another transaction updates that data in between the first and second read of transaction one.

We will use the following table **tblInventory** in this demo



Id	Name	ItemsInStock
1	iPhone	10

The following diagram explains the problem : Transaction 1 starts first. Reads ItemsInStock. Gets a value of 10 for first read. Transaction 1 is doing some work and at this point Transaction 2 starts and UpdatesItemsInStock to 5. Transaction 1 then makes a second read. At this point Transaction 1 gets a value of 5, reulting in non-repeatable read problem.

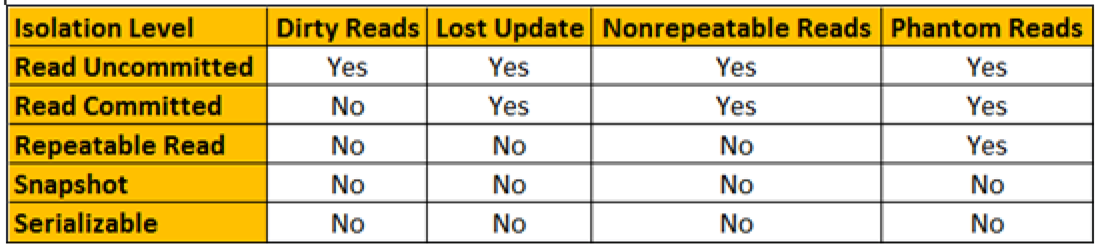

Transaction 1
Transaction 2
Read 1 : ItemsInStock = 10
Doing some work
Update : ItemsInStock = 5
Read 2 : ItemsInStock = 5

Non-repeatable read example : Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code. Notice that when Transaction 1 completes, it gets different values for read 1 and read 2, resulting in non-repeatable read.

```
-- Transaction 1
Begin Transaction
Select ItemsInStock from tblInventory where Id = 1
```

```
-- Do Some work
waitfor delay '00:00:10'

Select ItemsInStock from tblInventory where Id = 1
Commit Transaction

-- Transaction 2
Update tblInventory set ItemsInStock = 5 where Id = 1
```

Repeatable read or any other higher isolation level should solve the non-repeatable read problem.



Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

Fixing non repeatable read concurrency problem : To fix the non-repeatable read problem, set transaction isolation level of Transaction 1 to repeatable read. This will ensure that the data that Transaction 1 has read, will be prevented from being updated or deleted elsewhere. This solves the non-repeatable read problem.

When you execute Transaction 1 and 2 from 2 different instances of SQL Server management studio, Transaction 2 is blocked until Transaction 1 completes and at the end of Transaction 1, both the reads get the same value for ItemsInStock.

```
-- Transaction 1
Set transaction isolation level repeatable read
Begin Transaction
Select ItemsInStock from tblInventory where Id = 1

-- Do Some work
waitfor delay '00:00:10'

Select ItemsInStock from tblInventory where Id = 1
Commit Transaction

-- Transaction 2
Update tblInventory set ItemsInStock = 5 where Id = 1
```

Phantom reads example in sql server

Suggested Videos

Part 71 - sql server dirty read example
Part 72 - sql server lost update problem
Part 73 - Non repeatable read example in sql server

Phantom read happens when one transaction executes a query twice and it gets a different number of rows in the result set each time. This happens when a second transaction inserts a new row that matches the WHERE clause of the query executed by the first transaction.

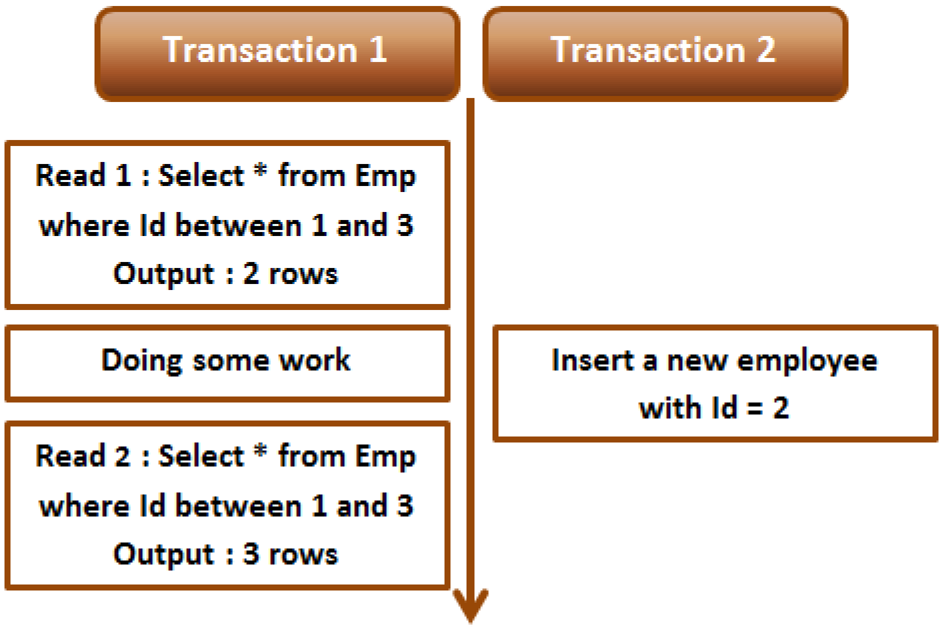
We will use the following table tblEmployees in this demo

Id	Name
1	Mark
3	Sara
100	Mary

Scrip to create the table tblEmployees

```
Create table tblEmployees
(
   Id int primary key,
   Name nvarchar(50)
)
Go

Insert into tblEmployees values(1,'Mark')
Insert into tblEmployees values(3, 'Sara')
Insert into tblEmployees values(100, 'Mary')
```

The following diagram explains the problem : Transaction 1 starts first. Reads from Emp table where Id between 1 and 3. 2 rows retrieved for first read. Transaction 1 is doing some work and at this point Transaction 2 starts and inserts a new employee with Id = 2. Transaction 1 then makes a second read. 3 rows retrieved for second read, reulting in phantom read problem.

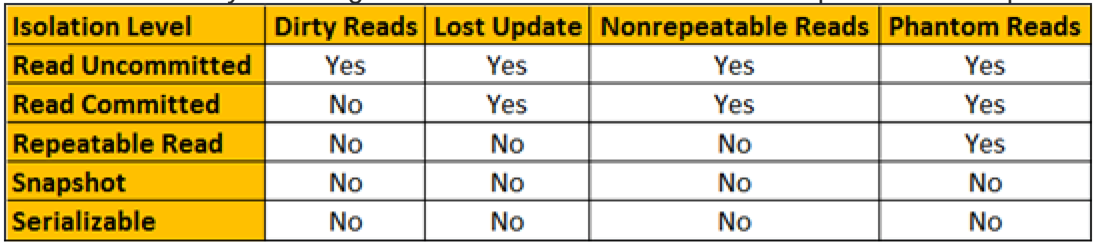

Transaction 1
Transaction 2
Read 1 : Select * from Emp where Id between 1 and 3 Output : 2 rows
Doing some work
Insert a new employee with Id = 2
Read 2 : Select * from Emp where Id between 1 and 3 Output : 3 rows

Phantom read example : Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code.

Notice that when Transaction 1 completes, it gets different number of rows for read 1 and read 2, resulting in phantom read.

```
-- Transaction 1
Begin Transaction
Select * from tblEmployees where Id between 1 and 3
-- Do Some work
waitfor delay '00:00:10'
Select * from tblEmployees where Id between 1 and 3
Commit Transaction

-- Transaction 2
Insert into tblEmployees values(2, 'Marcus')
```

Serializable or any other higher isolation level should solve the phantom read problem.



Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

Fixing phantom read concurrency problem : To fix the phantom read problem, set transaction isolation level of Transaction 1 to serializable. This will place a range lock on the rows between 1 and 3, which prevents any other transaction from inserting new rows with in that range. This solves the phantom read problem.

When you execute Transaction 1 and 2 from 2 different instances of SQL Server management studio, Transaction 2 is blocked until Transaction 1 completes and at the end of Transaction 1, both the reads get the same number of rows.

```
-- Transaction 1
Set transaction isolation level serializable
Begin Transaction
Select * from tblEmployees where Id between 1 and 3
-- Do Some work
waitfor delay '00:00:10'
Select * from tblEmployees where Id between 1 and 3
Commit Transaction

-- Transaction 2

Insert into tblEmployees values(2, 'Marcus')
```

Difference between repeatable read and serializable
Repeatable read prevents only non-repeatable read. Repeatable read isolation level ensures that the data that one transaction has read, will be prevented from being updated or deleted by any other transaction, but it doe not prevent new rows from being inserted by other transactions resulting in phantom read concurrency problem.

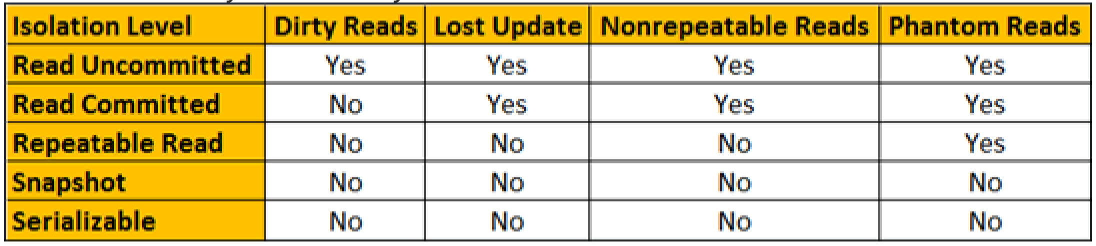
Serializable prevents both non-repeatable read and phantom read problems. Serializable isolation level ensures that the data that one transaction has read, will be prevented from being updated or deleted by any other transaction. It also prevents new rows from being inserted by other transactions, so this isolation level prevents both non-repeatable read and phantom read problems.

Snapshot isolation level in sql server

Suggested Videos

Part 72 - sql server lost update problem
Part 73 - Non repeatable read example in sql server
Part 74 - Phantom reads example in sql server

As you can see from the table below, just like serializable isolation level, snapshot isolation level does not have any concurrency side effects.

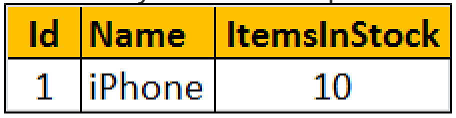


Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

What is the difference between serializable and snapshot isolation levels
Serializable isolation is implemented by acquiring locks which means the resources are locked for the duration of the current transaction. This isolation level does not have any concurrency side effects but at the cost of significant reduction in concurrency.

Snapshot isolation doesn't acquire locks, it maintains versioning in Tempdb. Since, snapshot isolation does not lock resources, it can significantly increase the number of concurrent transactions while providing the same level of data consistency as serializable isolation does.

Let us understand Snapshot isolation with an example. We will be using the following table tblInventory for this example.



Id	Name	ItemsInStock
1	iPhone	10

Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed.

```
--Transaction 1
Set transaction isolation level serializable
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
waitfor delay '00:00:10'
Commit Transaction

-- Transaction 2
```

```
Set transaction isolation level serializable
Select ItemsInStock from tblInventory where Id = 1
```

Now change the isolation level of Transaction 2 to snapshot. To set snapshot isolation level, it must first be enabled at the database level, and then set the transaction isolation level to snapshot.

```
-- Transaction 2
-- Enable snapshot isloation for the database
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
-- Set the transaction isolation level to snapshot
Set transaction isolation level snapshot
Select ItemsInStock from tblInventory where Id = 1
```

From the first window execute Transaction 1 code and from the second window, execute Transaction 2 code. Notice that Transaction 2 is not blocked and returns the data from the database as it was before Transaction 1 has started.

Modifying data with snapshot isolation level : Now let's look at an example of what happens when a transaction that is using snapshot isolation tries to update the same data that another transaction is updating at the same time.

In the following example, Transaction 1 starts first and it is updating ItemsInStock to 5. At the same time, Transaction 2 that is using snapshot isolation level is also updating the same data. Notice that Transaction 2 is blocked until Transaction 1 completes. When Transaction 1 completes, Transaction 2 fails with the following error to prevent concurrency side effect - Lost update. If Transaction 2 was allowed to continue, it would have changed the ItemsInStock value to 8 and when Transaction 1 completes it overwrites ItemsInStock to 5, which means we have lost an update. To complete the work that Transaction 2 is doing we will have to rerun the transaction.

```
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot
isolation to access table 'dbo.tblInventory' directly or indirectly in database 'SampleDB' to
update, delete, or insert the row that has been modified or deleted by another transaction. Retry
the transaction or change the isolation level for the update/delete statement.
```

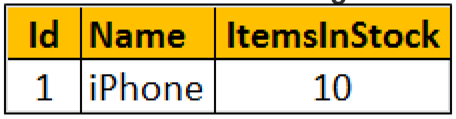
```
--Transaction 1
Set transaction isolation level serializable
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
waitfor delay '00:00:10'
Commit Transaction
```

```
-- Transaction 2
-- Enable snapshot isloation for the database
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
-- Set the transaction isolation level to snapshot
Set transaction isolation level snapshot
Update tblInventory set ItemsInStock = 8 where Id = 1
```

Read committed snapshot isolation level in sql server

Suggested Videos
Part 73 - Non repeatable read example in sql server
Part 74 - Phantom reads example in sql server
Part 75 - Snapshot isolation level in sql server

We will use the following table tblInventory in this demo



Id	Name	ItemsInStock
1	iPhone	10

Read committed snapshot isolation level is not a different isolation level. It is a different way of implementing Read committed isolation level. One problem we have with Read Committed isloation level is that, it blocks the transaction if it is trying to read the data, that another transaction is updating at the same time.

The following example demonstrates the above point. Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed.

```
--Transaction 1
Set transaction isolation level Read Committed
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
waitfor delay '00:00:10'
Commit Transaction
```

```
-- Transaction 2
Set transaction isolation level read committed
Begin Transaction
Select ItemsInStock from tblInventory where Id = 1
Commit Transaction
```

We can make Transaction 2 to use row versioning technique instead of locks by enabling Read committed snapshot isolation at the database level. Use the following command to enable READ_COMMITTED_SNAPSHOT isolation

```
Alter database SampleDB SET READ_COMMITTED_SNAPSHOT ON
```

Please note : For the above statement to execute successfully all the other database connections should be closed.

After enabling READ_COMMITTED_SNAPSHOT, execute Transaction 1 first and then Transaction 2 simultaneously. Notice that the Transaction 2 is not blocked. It immediately returns the committed data that is in the database before Transaction 1 started. This is because Transaction 2 is now using Read committed snapshot isolation level.

Let's see if we can achieve the same thing using snapshot isolation level instead of read committed snapshot isolation level.

Step 1 : Turn off READ_COMMITTED_SNAPSHOT

```
Alter database SampleDB SET READ_COMMITTED_SNAPSHOT OFF
```

Step 2 : Enable snapshot isolation level at the database level

```
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
```

Step 3 : Execute Transaction 1 first and then Transaction 2 simultaneously. Just like in the previous example, notice that the Transaction 2 is not blocked. It immediately returns the committed data that is in the database before Transaction 1 started.

```
--Transaction 1
Set transaction isolation level Read Committed
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
waitfor delay '00:00:10'
Commit Transaction

-- Transaction 2
Set transaction isolation level snapshot
Begin Transaction
Select ItemsInStock from tblInventory where Id = 1
Commit Transaction
```

So what is the point in using read committed snapshot isolation level over snapshot isolation level?
There are some differences between read committed snapshot isolation level and snapshot isolation level. We will discuss these in our next video.

Difference between snapshot isolation and read committed snapshot

Suggested Videos

Part 74 - Phantom reads example in sql server

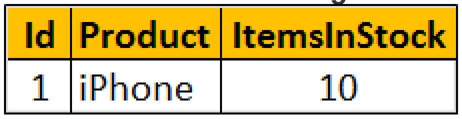
Part 75 - Snapshot isolation level in sql server

Part 76 - Read committed snapshot isolation level in sql server

Read Committed Snapshot Isolation	Snapshot Isolation
No update conflicts	Vulnerable to update conflicts
Works with existing applications without requiring any change to the application	Application change may be required to use with an existing application
Can be used with distributed transactions	Cannot be used with distributed transactions
Provides statement-level read consistency	Provides transaction-level read consistency

Update conflicts : Snapshot isolation is vulnerable to update conflicts where as Read Committed Snapshot Isolation is not. When a transaction running under snapshot isolation triess to update data that an another transaction is already updating at the sametime, an update conflict occurs and the transaction terminates and rolls back with an error.

We will use the following table tblInventory in this demo



Id	Product	ItemsInStock
1	iPhone	10

Enable Snapshot Isolation for the SampleDB database using the following command

```
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION ON
```

Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed. When Transaction 1 completes, Transaction 2 raises an update conflict and the transaction terminates and rolls back with an error.

```
--Transaction 1
Set transaction isolation level snapshot
Begin Transaction
Update tblInventory set ItemsInStock = 8 where Id = 1
waitfor delay '00:00:10'
Commit Transaction
```

```
-- Transaction 2
Set transaction isolation level snapshot
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
Commit Transaction
```

Now let's try the same thing using **Read Committed Sanpshot Isolation**

Step 1 : Disable Snapshot Isolation for the SampleDB database using the following command

```
Alter database SampleDB SET ALLOW_SNAPSHOT_ISOLATION OFF
```

Step 2 : Enable Read Committed Sanpshot Isolation at the database level using the following command

```
Alter database SampleDB SET READ_COMMITTED_SNAPSHOT ON
```

Step 3 : Open 2 instances of SQL Server Management studio. From the first window execute Transaction 1 code and from the second window execute Transaction 2 code. Notice that Transaction 2 is blocked until Transaction 1 is completed. When Transaction 1 completes, Transaction 2 also completes successfully without any update conflict.

```
--Transaction 1
Set transaction isolation level read committed
Begin Transaction
Update tblInventory set ItemsInStock = 8 where Id = 1
waitfor delay '00:00:10'
Commit Transaction
```

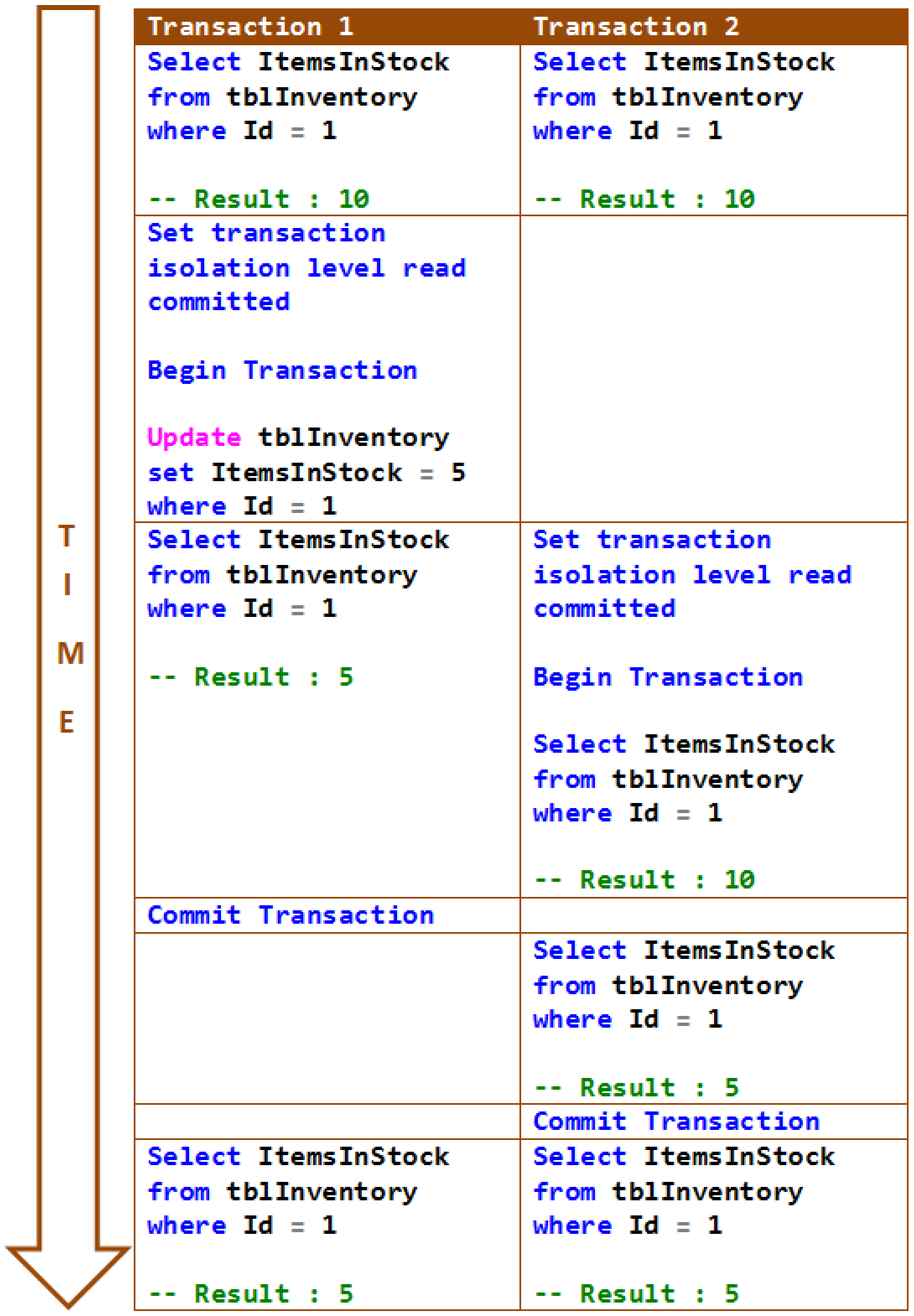
```
-- Transaction 2
Set transaction isolation level read committed
Begin Transaction
Update tblInventory set ItemsInStock = 5 where Id = 1
Commit Transaction
```

Existing application : If your application is using the default Read Committed isolation level, you can very easily make the application to use Read Committed Snapshot Isolation without requiring any change to the application at all. All you need to do is turn on READ_COMMITTED_SNAPSHOT option in the database, which will change read committed isolation to use row versioning when reading the committed data.

Distributed transactions : Read Committed Snapshot Isolation works with distributed transactions, whereas snapshot isolation does not.

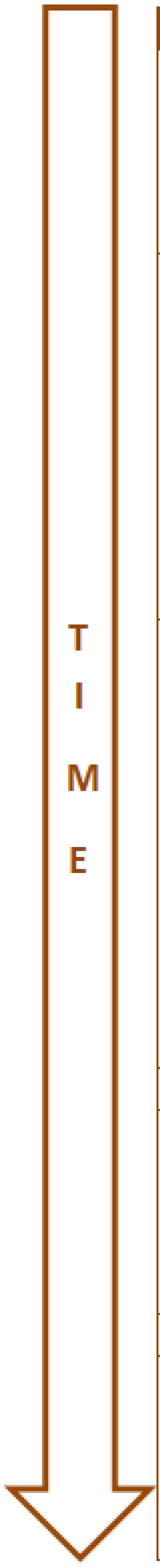
Read consistency : Read Committed Snapshot Isolation provides statement-level read consistency where as Snapshot Isolation provides transaction-level read consistency. The following diagrams explain this.

Transaction 2 has 2 select statements. Notice that both of these select statements return different data. This is because Read Committed Snapshot Isolation returns the last committed data before the select statement began and not the last committed data before the transaction began.



Transaction 1	Transaction 2
Select ItemsInStock from tblInventory where Id = 1 -- Result : 10	Select ItemsInStock from tblInventory where Id = 1 -- Result : 10
Set transaction isolation level read committed Begin Transaction Update tblInventory set ItemsInStock = 5 where Id = 1	
Select ItemsInStock from tblInventory where Id = 1 -- Result : 5	Set transaction isolation level read committed Begin Transaction Select ItemsInStock from tblInventory where Id = 1 -- Result : 10
Commit Transaction	
	Select ItemsInStock from tblInventory where Id = 1 -- Result : 5
	Commit Transaction
Select ItemsInStock from tblInventory where Id = 1 -- Result : 5	Select ItemsInStock from tblInventory where Id = 1 -- Result : 5

In the following example, both the select statements of Transaction 2 return same data. This is because Snapshot Isolation returns the last committed data before the transaction began and not the last committed data before the select statement began.



Transaction 1	Transaction 2

Select ItemsInStock
from tblInventory where
Id = 1
 	
-- Result : 10
	Select ItemsInStock
	from tblInventory where
	Id = 1
	
	-- Result : 10

Set transaction isolation level snapshot Begin Transaction Update tblInventory set ItemsInStock = 5 where Id = 1	

Select ItemsInStock
from tblInventory where
Id = 1
 	
-- Result : 5
	Set transaction
	isolation level
	snapshot
	
	Begin Transaction
	
	Select ItemsInStock
	from tblInventory where
	Id = 1
	
	-- Result : 10

Commit Transaction	

	Select ItemsInStock
	from tblInventory where
	Id = 1
	
	-- Result : 10

	Commit Transaction

Select ItemsInStock
from tblInventory where
Id = 1
 	
-- Result : 5
	Select ItemsInStock
	from tblInventory where
	Id = 1
	
	-- Result : 5