DDL (Data Definition Language)-By default commit.

DR CAT- Drop, Rename, Create, Alter, Truncate

DML (Data Manipulation Language) - By default not commit.

SUDI-Select, Update, Delete, Insert

DQL (Data Query Language)

Select

Difference between Delete, Truncate and drop command.

**Delete:-**

1-Delete is a DML Command.

2-Delete command support the where clause.

3-We can rollback the record.

4-This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.

Ex:-1-DELETE FROM employee WHERE id = 100;

2-DELETE FROM employee;

**Truncate:-**

1-Truncate is DDL command.

2-Truncate did not support where clause.

3-We cannot rollback by default its commit.

4-Truncate is much faster than delete.

Reason: When you type DELETE. All the data get copied into the Rollback Table space first. then delete operation get performed. That’s why when you type ROLLBACK after deleting a table ,you can get back the data(The system get it for you from the Rollback Table space).All this process take time. But when you type TRUNCATE, it removes data directly without copying it into the Rollback Table space. That’s why TRUNCATE is faster. Once you Truncate you can't get back the data.

5-This command is used to delete all the rows from the table and free the space containing the table.

Ex:- TRUNCATE TABLE employee;

**Drop:-**

1-Drop is a DDL command.

2-Drop also did not support the where clause.

3-We cannot rollback by default its commit.

4-The SQL DROP command is used to remove an object from the database. If you drop a table, all the rows in the table are deleted and the table structure is removed from the database. Once a table is dropped we cannot get it back.

Ex:-1-DROP TABLE employee;

2- Drop database <DB name>

**Difference between DROP and TRUNCATE Statement:**

If a table is dropped, all the relationships with other tables will no longer be valid, the integrity constraints will be dropped, grant or access privileges on the table will also be dropped, if you want use the table again it has to be recreated with the integrity constraints, access privileges and the relationships with other tables should be established again. But, if a table is truncated, the table structure remains the same, therefore any of the above problems will not exist.

**Temp Table:-**

These tables can be created at **runtime** and can do the all kinds of operations that one normal table can do. But, based on the table types, the scope is limited. These tables are created inside tempdb database.

Different Types of Temporary Tables

SQL Server provides two types of temp tables based on the behavior and scope of the table. These are:

* Local Temp Table
* Global Temp Table

**1-Local Table:-**

Local temporary table are stored in TempDB. The name of the table is suffixed with lot of underscore and a random number. For this reason we have to use the LIKE operator in the query.

Ex:-

Create table #Example (id int, name varchar (20))--Create the temp table

Insert into #Example values (101,'Ankit');--Insert the record in temp table

Insert into #Example values (102,'Mohit');--Insert the record in temp table

Select \* from #Example;--Retrieve the record from the temp table

Output:-

101 Ankit

102 Mohit

Select name from tempdb..sysobjects where name like'#Example%';--See the temp table

Output:-

#Example\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_000000000005

-Local temp table is available only for the connection that has created the table.

-A local temp table is automatically dropped, when the connection that created the local temp table is closed.

-If the user wants to explicitly drop the local temporary table, he can do so using

Drop table #Example

-If the temporary table is created inside the stored procedure. It’s gets dropped automatically upon the completion of the stored procedure.

Ex:-

Create procedure MyProcedure

As

Begin

Create table #Example (id int, name varchar (20))

Insert into #Example values (101,'Ankit');--Insert the record in temp table

Insert into #Example values (102,'Mohit');--Insert the record in temp table

Select \* from #Example

End

Execute the stored procedure

Exec MyProcedure

Output:-

101 Ankit

102 Mohit

Execute the select query for temp table:-

Select \* from #Example

Output:-

Msg 208, Level 16, State 0, Line 7

Invalid object name '#Example'.

Note:- It is also possible for different connection can create the same a local temp table with the same name. For Example user1 and user2 both can create a local temp table with the same name like:-#Example.

### Global Temp Table

Global Temporary tables name starts with a double hash ("##").

Ex:-Create table ##Example (id int, name varchar (20))

-Global temporary table visible to all the user of the sql server and only destroyed when all user connection are closed.

-If the global temporary table is created inside the stored procedure then procedure and the global temp table both are created and destroyed when all user session are closed.

Difference B/W local and Global temp Table:-

1-Local temp table name start with singe hash (#) where as global temp table name start with (##).

2-Local temp table are destroyed when the local temp table created connection is closed where as global temp table destroyed when all the user connection are closed in SQL Server.

3-Sql Server append some random number at the end of the local temp variable where as this is not done in global temp table.

4-Local temp table are accessible only that connection that created it but global temp table is access by all the user that are available in Sql Server.

**Table Variable:-**

Table variable just like a temporary table .It’s just alternative of temporary table. They are partial store in memory and the disk. We can perform the DML operation on Table Variable but DDL operation is not possible. In table variable the constraint types allowed are PRIMARY KEY, UNIQUE, NULL, and CHECK but they must be incorporated with the creation of the table in the DECLARE statement. FOREIGN KEY constraints are not allowed.

-Table variable are stored in RAM. So after execution of the query all the data assign to table variable are removed.

Syntax: - For create the table variable

Declare @Emp Table (Id int, name varchar (20));

Insert the record inside the table

Insert into @EMP Values (101,’A’)

Insert into @EMP Values (102,’B’)

Select \* from@ EMP

-We have to execute the above 5 query together then only it’s showing the records that we stored inside the table variable. If we try to execute the query one by one it’s showing the error because table variable are stored in RAM so after execute the single statement all the data that is assign to the table variable are removed (For reference see the video(Temp Tables Vs Table Variables) in SQL Server Folder).

-Table variable can be declared in batch, stored procedure and user define function just like a local variable. They cannot be dropped explicitly. Once the batch execution is finished, the Table Variables are dropped automatically.

-The Table Variables are stored in both the memory and the disk in the tempdb database.

-The structure of Table Variables cannot be changed once they are created. Thus, it means that DDL commands cannot be run in Table Variables.

- The Table variable is no longer exists after the batch, stored procedure and function exist.

We cannot use a table variable as an input or an output parameter.

**Functions in Sql Server:-**

Function is a database object in Sql Server. Basically it is a set of sql statements that accepts only input parameters, perform actions and return the result. Function can return only single value or a table. We can’t use function to Insert, Update, and Delete records in the database table.

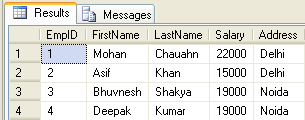
User Defined Function

These functions are created by user in system database or in user defined database. We three types of user defined functions.

1-Scalar Function

User defined scalar function also returns single value as a result of actions perform by function. We return any datatype value from function.

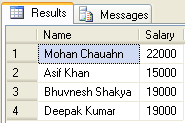
1. **--Create a table**
2. **CREATE TABLE Employee**
3. **(**
4. **EmpID int PRIMARY KEY,**
5. **FirstName varchar(50) NULL,**
6. **LastName varchar(50) NULL,**
7. **Salary int NULL,**
8. **Address varchar(100) NULL,**
9. **)**
10. **--Insert Data**
11. **Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(1,'Mohan','Chauahn',22000,'Delhi');**
12. **Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(2,'Asif','Khan',15000,'Delhi');**
13. **Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(3,'Bhuvnesh','Shakya',19000,'Noida');**
14. **Insert into Employee(EmpID,FirstName,LastName,Salary,Address) Values(4,'Deepak','Kumar',19000,'Noida');**
15. **--See created table**
16. **Select \* from Employee**



1. **--Create function to get emp full name**
2. **Create function fnGetEmpFullName**
3. **(**
4. **@FirstName varchar(50),**
5. **@LastName varchar(50)**
6. **)**
7. **returns varchar(101)**
8. **As**
9. **Begin return (Select @FirstName + ' '+ @LastName);**
10. **end**

http://www.dotnettricks.com/img/sqlserver/success.png

1. **--Calling the above created function**
2. **Select dbo.fnGetEmpFullName (FirstName,LastName) as Name, Salary from Employee**



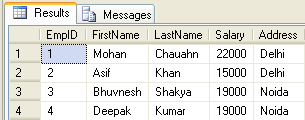
2-Inline Table-Valued Function

User defined inline table-valued function returns a table variable as a result of actions perform by function. The value of table variable should be derived from a single SELECT statement.

1. **--Create function to get employees**
2. **Create function fnGetEmployee()**
3. **returns Table**
4. **As**
5. **return (Select \* from Employee)**

http://www.dotnettricks.com/img/sqlserver/success.png

1. **--Now call the above created function**
2. **Select \* from fnGetEmployee()**



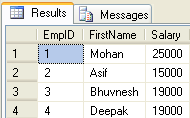
3-Multi-Statement Table-Valued Function

User defined multi-statement table-valued function returns a table variable as a result of actions perform by function. In this a table variable must be explicitly declared and defined whose value can be derived from a multiple sql statements.

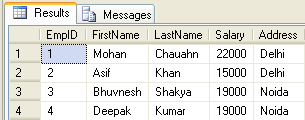
1. **--Create function for EmpID,FirstName and Salary of Employee**
2. **Create function fnGetMulEmployee()**
3. **returns @Emp Table**
4. **(**
5. **EmpID int,**
6. **FirstName varchar(50),**
7. **Salary int**
8. **)**
9. **As**
10. **begin**
11. **Insert into @Emp Select e.EmpID,e.FirstName,e.Salary from Employee e;**
12. **--Now update salary of first employee**
13. **update @Emp set Salary=25000 where EmpID=1;**
14. **--It will update only in @Emp table not in Original Employee table**
15. **return**
16. **end**

http://www.dotnettricks.com/img/sqlserver/success.png

1. **--Now call the above created function**
2. **Select \* from fnGetMulEmployee()**



1. **--Now see the original table. This is not affected by above function update command**
2. **Select \* from Employee**



# SQL Server Stored Procedures

A stored procedure is a precompiled set of one or more SQL statements that is stored on Sql Server. Benefit of Stored Procedures is that they are executed on the server side and perform a set of actions, before returning the results to the client side. This allows a set of actions to be executed with minimum time . Hence stored procedure improves performance to execute sql statements.

Stored procedure can accepts input and output parameters.

Stored procedure can returns multiple values using output parameters.

Using stored procedure, we can Select, Insert, Update and Delete data in database.

How to Write a Stored Procedure in SQL Server

*/\* Getstudentname is the name of the stored procedure\*/*

Create PROCEDURE Getstudentname (

@studentid INT *--Input parameter, Studentid of the student*

)

AS

BEGIN

SELECT Firstname+' '+Lastname FROM tbl\_Students WHERE studentid=@studentid

END

**We can also collect the student name in the output parameter of the Stored Procedure. For example:**

Hide   Copy Code

*/\**

*GetstudentnameInOutputVariable is the name of the stored procedure which*

*uses output variable @Studentname to collect the student name returns by the*

*stored procedure*

*\*/*

Create PROCEDURE GetstudentnameInOutputVariable

(

@studentid INT, *--Input parameter , Studentid of the student*

@studentname VARCHAR(200) OUT *-- Out parameter declared with the help of OUT keyword*

)

AS

BEGIN

SELECT @studentname= Firstname+' '+Lastname FROM tbl\_Students WHERE studentid=@studentid

END

How to Alter a Stored Procedure in a SQL Server

In SQL Server, a stored procedure can be modified with the help of the Alter **keyword**. Now if we want to get student email address through the sameprocedure GetstudentnameInOutputVariable. So we need to modify it by adding one more output parameter " @StudentEmail " which is shown below:

*/\**

*Stored Procedure GetstudentnameInOutputVariable is modified to collect the*

*Email address of the student with the help of the Alert Keyword*

*\*/*

Alter PROCEDURE GetstudentnameInOutputVariable

(

@studentid INT, *--Input parameter , Studentid of the student*

@studentname VARCHAR (200) OUT, *-- Output parameter to collect the student name*

@StudentEmail VARCHAR (200) OUT *-- Output Parameter to collect the student email*

)

AS

BEGIN

SELECT @studentname= Firstname+' '+Lastname,

@StudentEmail=email FROM tbl\_Students WHERE studentid=@studentid

END

**Note**: It is not necessary that a stored procedure will have to return. There can be a case when a stored procedure doesn't returns anything. For example, a stored procedure can be used to Insert, delete or update a SQL statement. For example, the below stored procedure is used to insert value into the table tbl\_students

*/\**

*This Stored procedure is used to insert value into the table tbl\_students.*

*\*/*

Create Procedure InsertStudentrecord

(

@StudentFirstName Varchar (200),

@StudentLastName Varchar (200),

@StudentEmail Varchar (50)

)

As

Begin

Insert into tbl\_Students (Firstname, lastname, Email)

Values (@StudentFirstName, @StudentLastName, @StudentEmail)

End

.

## Execution of the Stored Procedure in SQL Server

### Execution of the Stored Procedure which doesn't have an Output Parameter

A stored procedure is used in the SQL Server with the help of the "Execute" or "Exec" Keyword. For example, if we want to execute the stored procedure "Getstudentname", then we will use the following statement.

Hide   Copy Code

Execute Getstudentname 1

Exec Getstudentname 1

### Execution of the Stored Procedure using the Output Parameter

If we want to execute the Stored procedure "GetstudentnameInOutputVariable" , then we first need to declare the variable to collect the output values. For example:

Declare @Studentname as nvarchar(200) -- Declaring the variable to collect the Studentname

Declare @Studentemail as nvarchar(50) -- Declaring the variable to collect the Studentemail

Execute GetstudentnameInOutputVariable 1 , @Studentname output, @Studentemail outputproc

select @Studentname,@Studentemail -- "Select" Statement is used to show the output from Procedure

Exception handling example in procedure

1. **BEGIN TRY**
2. **DECLARE @num INT, @msg varchar(200)**
3. **---- Divide by zero to generate Error**
4. **SET @num = 5/0**
5. **PRINT 'This will not execute'**
6. **END TRY**
7. **BEGIN CATCH**
8. **PRINT 'Error occured that is'**
9. **set @msg=(SELECT ERROR\_MESSAGE())**
10. **print @msg;**
11. **END CATCH**
12. **GO**

http://www.dotnettricks.com/img/sqlserver/dividebyzero1.png

**Call function using stored procedure SQL server**

*--exec fun\_proc 1,2*

CREATE PROC fun\_proc

@p\_val1 int,

@p\_val2 int

AS

BEGIN

Declare @sum int

Select @sum = dbo.sums(@p\_val1, @p\_val2)*--call function to get the value*

print @sum

END

*--Function*

create function sums

(

@val int,

@val2 int

)

returns int

as

begin

return(@val + @val2)

end

# Difference Between Stored Procedure and User Defined Function in SQL Server

|  |  |  |
| --- | --- | --- |
| **Sr.No.** | **User Defined Function** | **Stored Procedure** |
| 1 | Function must return a value. | Stored Procedure may or not return values. |
| 2 | Will allow only Select statements, it will not allow us to use DML statements. | Can have select statements as well as DML statements such as insert, update, delete and so on |
| 3 | It will allow only input parameters, doesn't support output parameters. | It can have both input and output parameters. |
| 4 | It will not allow us to use try-catch blocks. | For exception handling we can use try catch blocks. |
| 5 | Transactions are not allowed within functions. | Can use transactions within Stored Procedures. |
| 6 | Stored Procedures can't be called from a function. | Stored Procedures can call functions. |

What is a Trigger

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

A trigger is a special kind of a store procedure that executes in response to certain action on the table like insertion, deletion or updation of data. It is a database object which is bound to a table and is executed automatically. You can’t explicitly invoke triggers.

Types of Triggers

1. After Triggers (For Triggers)
2. Instead Of Triggers

(i) After Triggers

These triggers run after an insert, update or delete on a table. They are **not supported for views.**   
AFTER TRIGGERS can be classified further into three types as:

1. AFTER INSERT Trigger.
2. AFTER UPDATE Trigger.
3. AFTER DELETE Trigger.

Let’s create after triggers. First of all, let’s create a table and insert some sample data. Then, on this table, I will be attaching several triggers.

Hide   Copy Code

CREATE TABLE Employee\_Test

(

Emp\_ID INT Identity,

Emp\_name Varchar(100),

Emp\_Sal Decimal (10, 2)

)

INSERT INTO Employee\_Test VALUES ('Anees', 1000);

INSERT INTO Employee\_Test VALUES ('Rick',1200);

INSERT INTO Employee\_Test VALUES ('John', 1100);

INSERT INTO Employee\_Test VALUES ('Stephen', 1300);

INSERT INTO Employee\_Test VALUES ('Maria', 1400);

I will be creating an AFTER INSERT TRIGGER which will insert the rows inserted into the table into another audit table. The main purpose of this audit table is to record the changes in the main table. This can be thought of as a generic audit trigger.

Now, create the audit table as:-

Hide   Copy Code

CREATE TABLE Employee\_Test\_Audit

(

Emp\_ID int,

Emp\_name varchar (100),

Emp\_Sal decimal (10, 2),

Audit\_Action varchar(100),

Audit\_Timestamp datetime

)

(a) After Insert Trigger

This trigger is fired after an INSERT on the table. Let’s create the trigger as:

Create trigger Inserttrigger on Employee\_Test

After Insert

As

declare @Empid int;

declare @empname varchar (100);

declare @empsal decimal (10, 2);

declare @audit\_action varchar (100);

Select @Empid=Emp\_ID from inserted;

select @Empname=Emp\_Name from inserted;

select @empsal=Emp\_Sal from inserted;

set @audit\_action='Inserted Record--After Insert Trigger';

insert into Employee\_Test\_Audit values (@Empid,@empname,@empsal,@audit\_action,getdate());

Go

The CREATE TRIGGER statement is used to create the trigger. The ON clause specifies the table name on which the trigger is to be attached. The FOR INSERT specifies that this is an AFTER INSERT trigger. In place of FOR INSERT, AFTER INSERT can be used. Both of them mean the same.

Inserted table is a special table that is used by trigger and this table is only available in this context of the trigger means within the trigger. Whenever we insert any row in a table like our Employee\_Test table a sql server is maintain a magic table called inserted table that retain the copy of the row that we inserted in the Employee\_Test table.

- To see the newly created trigger in action, let’s insert a row into the main table as:

Insert into Employee\_Test values ('Ankit', 19000);

-After execute the above query one record is inserted into the Employee\_Test table. After the execution of the After Insert trigger below record is also inserted into the Employee\_Test\_Audit table.

6 Ankit 19000.00 Inserted Record -- After Insert Trigger. 2008-04-26 12:00:55.700

(b) AFTER DELETE Trigger

This trigger is fired after a delete on the table. Let’s create the trigger as:

CREATE TRIGGER trgAfterDelete ON [dbo].[Employee\_Test]

AFTER DELETE

AS

declare @empid int;

declare @empname varchar (100);

Declare @empsal decimal (10, 2);

declare @audit\_action varchar(100);

select @empid=Emp\_ID from deleted;

select @empname=Emp\_Name from deleted;

select @empsal=Emp\_Sal from deleted;

set @audit\_action='Deleted -- After Delete Trigger.';

insert into Employee\_Test\_Audit

(Emp\_ID,Emp\_Name,Emp\_Sal,Audit\_Action,Audit\_Timestamp)

values(@empid,@empname,@empsal,@audit\_action,getdate());

PRINT 'AFTER DELETE TRIGGER fired.'

GO

Delete table is same as the insert table means delete table is contain the copy of the row that we deleted

In this trigger, the deleted record’s data is picked from the **logical deleted table** and inserted into the audit table. Let’s fire a delete on the main table. A record has been inserted into the audit table as:

6 Chris 1550.00 Deleted -- After Delete Trigger. 2008-04-26 12:52:13.867

All the triggers can be enabled/disabled on the table using the statement

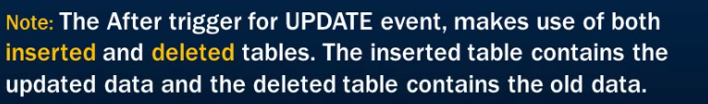
ALTER TABLE Employee\_Test {ENABLE|DISBALE} TRIGGER ALL

Specific Triggers can be enabled or disabled as:

ALTER TABLE Employee\_Test DISABLE TRIGGER trgAfterDelete

This disables the After Delete Trigger named trgAfterDelete on the specified table.

(c) AFTER UPDATE Trigger



CREATE TRIGGER trgAfterUpdate ON [dbo]. [Employee\_Test]

FOR UPDATE

AS

declare @empid int;

declare @empname varchar (100);

declare @empsal decimal (10,2);

declare @audit\_action varchar (100);

select @empid=Emp\_ID from inserted;

select @empname=Emp\_Name from inserted;

select @empsal=Emp\_Sal from inserted;

if update(Emp\_Name)

set @audit\_action='Updated Record -- After Update Trigger.';

if update (Emp\_Sal)

set @audit\_action='Updated Record -- After Update Trigger.';

insert into Employee\_Test\_Audit (Emp\_ID, Emp\_Name, Emp\_Sal, Audit\_Action, Audit\_Timestamp) values (@empid, @empname, @empsal, @audit\_action, getdate ());

PRINT 'AFTER UPDATE Trigger fired.'

GO

The AFTER UPDATE Trigger is created in which the updated record is inserted into the audit table. There is **no logical table updated like the logical table inserted.** We can obtain the updated value of a field from the update(column\_name) function. In our trigger, we have used, if update(Emp\_Name) to check if the column Emp\_Name has been updated. We have similarly checked the column Emp\_Sal for an update.

Let’s update a record column and see what happens.

Update Employee\_Test set Emp\_Sal=1550 where Emp\_ID=6

This inserts the row into the audit table as:

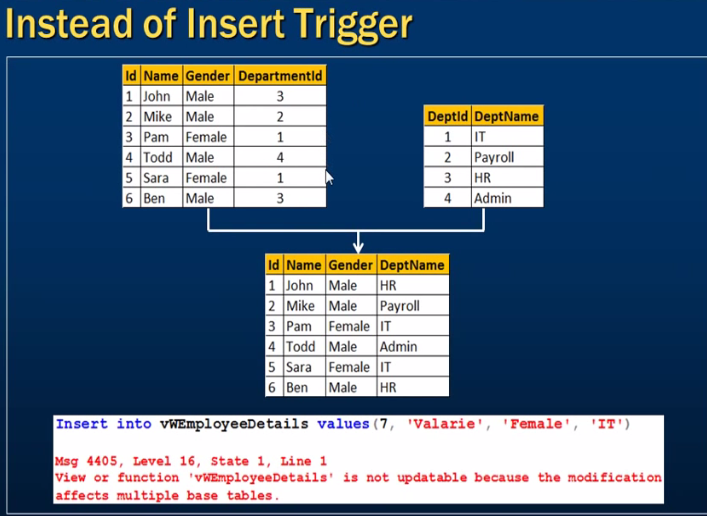
6 Chris 1550.00 Updated Records -- After Update Trigger. 2008-04-26 12:38:11.843

(ii) Instead of Triggers

An INSTEAD OF trigger can be defined to perform error or value checking on one or more columns, and then perform additional actions before inserting the record. The table data will not be affected.

We can have an INSTEAD OF insert/update/delete trigger on a table that successfully executed but does not include the actual insert/update/delete to the table.

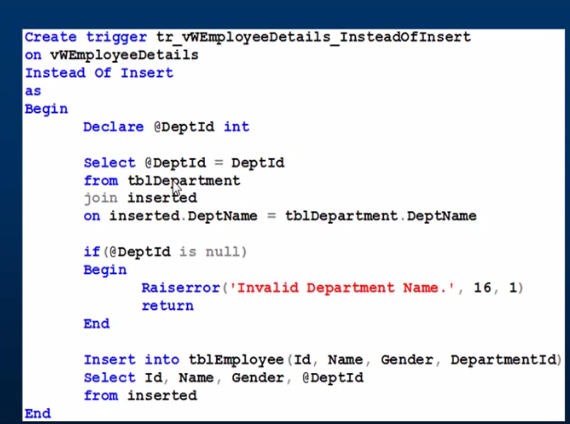
**Example :** If you insert record/row in a table then the trigger associated with the insert event on this table will fire before the row passes all the checks, such as primary key, rules, and constraints. If the record/row insertion fails, SQL Server will fire the Instead of Trigger.



In the above screen shot one is Employee table and other one is department table and create one view in using both of the parents table.

-After creating the view if we try to insert the record in the view then it’s showing the above mention error but the above mention data is insert into the special table which is called inserted table. Because that view is related to two base table. So for that we use the instead of trigger.

Ex:-



**View in SQL Server**

View is nothing but a virtual table that just looks likes a real table.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

A view can be used for the following purposes:

* To focus, simplify, and customize the perception each user has of the database.
* As a security mechanism by allowing users to access data through the view, without granting the users permissions to directly access the underlying base tables.
* We make views for security purpose since it restricts the user to view some columns/fields of the table(s).

### CREATE VIEW Syntax

CREATE VIEW view\_name AS  
SELECT column1, column2 ...  
FROM table\_name  
WHERE condition;

### Example

Let's look at an example of how to use the CREATE VIEW statement in SQL Server (Transact-SQL).

For example:

CREATE VIEW prod\_inv AS

SELECT products.product\_id, products.product\_name, inventory.quantity

FROM products

INNER JOIN inventory

ON products.product\_id = inventory.product\_id

WHERE products.product\_id >= 1000;

This SQL Server CREATE VIEW example would create a virtual table based on the result set of the SELECT statement. The view would be called prod\_inv.

You can now query the SQL Server VIEW as follows:

SELECT \* FROM prod\_inv;

**Update VIEW**

You can modify the definition of a VIEW in SQL Server without dropping it by using the ALTER VIEW Statement.

### Example

Here is an example of how you would use the ALTER VIEW Statement in SQL Server (Transact-SQL):

ALTER VIEW prod\_inv AS

SELECT products.product\_name, inventory.quantity

FROM products

INNER JOIN inventory

ON products.product\_id = inventory.product\_id

WHERE products.product\_id >= 500

AND products.product\_id <= 1000;

This ALTER VIEW example would update the definition of the VIEW called prod\_inv without dropping it in SQL Server. The VIEW must exist for you to be able to execute an ALTER VIEW command.

## Drop VIEW

Once a VIEW has been created in SQL Server, you can drop it with the DROP VIEW Statement.

### Syntax

The syntax for the DROP VIEW statement in SQL Server (Transact-SQL) is:

DROP VIEW view\_name;

User Defined Views

These types of view are defined by users. We have two types of user defined views.

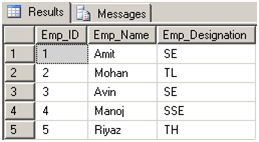
Simple View

When we create a view on a single table, it is called simple view.

1. **--Now Insert data to table Employee\_Test**
2. **Insert into Employee\_Test values ('Amit','PHP',12000,'SE');**
3. **Insert into Employee\_Test values ('Mohan','ASP.NET',15000,'TL');**
4. **Insert into Employee\_Test values ('Avin','C#',14000,'SE');**
5. **Insert into Employee\_Test values ('Manoj','JAVA',22000,'SSE');**
6. **Insert into Employee\_Test values ('Riyaz','VB',18000,'TH');**
7. **-- Now create view on single table Employee\_Test**
8. **create VIEW vw\_Employee\_Test**
9. **AS**
10. **Select Emp\_ID ,Emp\_Name ,Emp\_Designation**
11. **From Employee\_Test**

http://www.dotnettricks.com/img/sqlserver/success.png

1. **-- Query view like as table**
2. **Select \* from vw\_Employee\_Test**



**In simple view we can insert, update, delete data.**

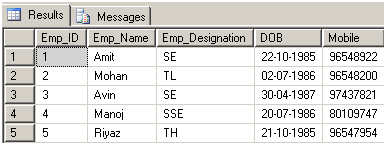
Complex View

When we create a view on more than one table, it is called complex view.

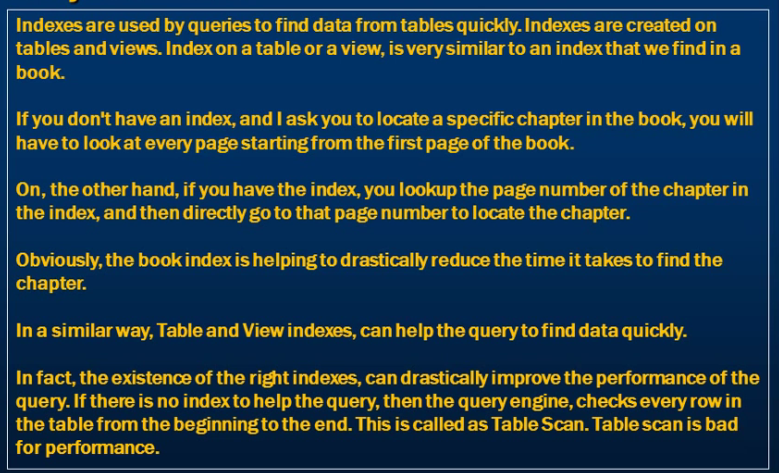
1. **Select e.Emp\_ID, e.Emp\_Name,e.Emp\_Designation,p.DOB,p.Mobile**
2. **From Employee\_Test e INNER JOIN Personal\_Info p**
3. **On e.Emp\_Name = p. Emp\_Name**

http://www.dotnettricks.com/img/sqlserver/success.png

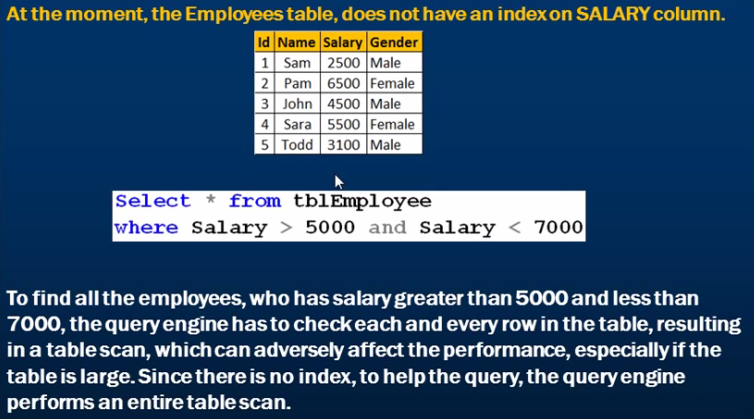
1. **-- Now Query view like as table**
2. **Select \* from vw\_Employee\_Personal\_Info**



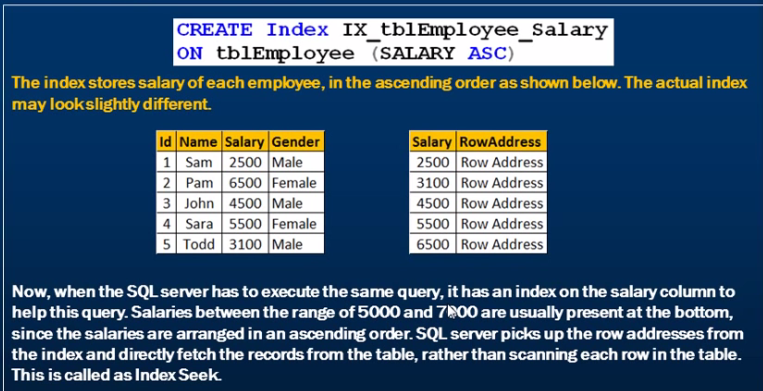
**Indexes in Sql Server:-**

****

**Problem if we did not use index:-**

****

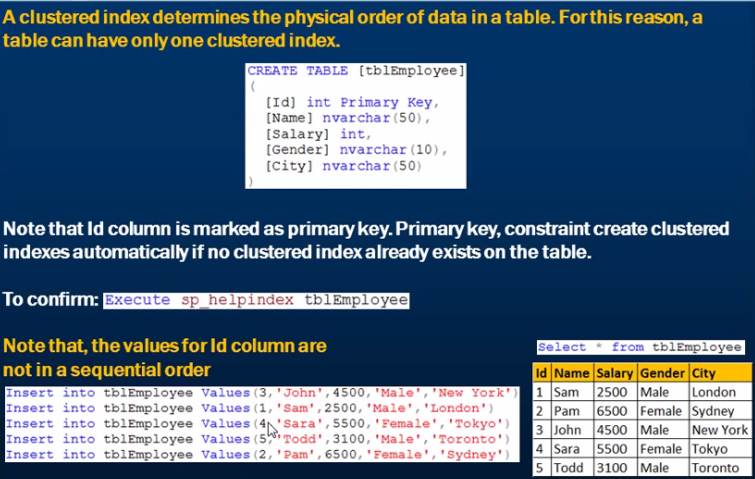
**Creating An Index:-**

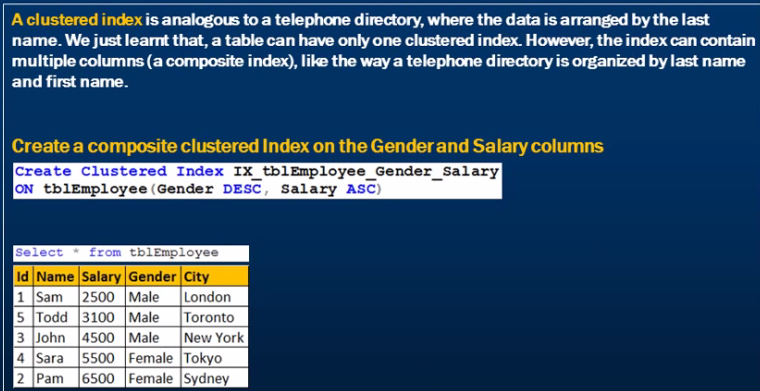
****

**Types of Indexes in Sql Server:-**

****

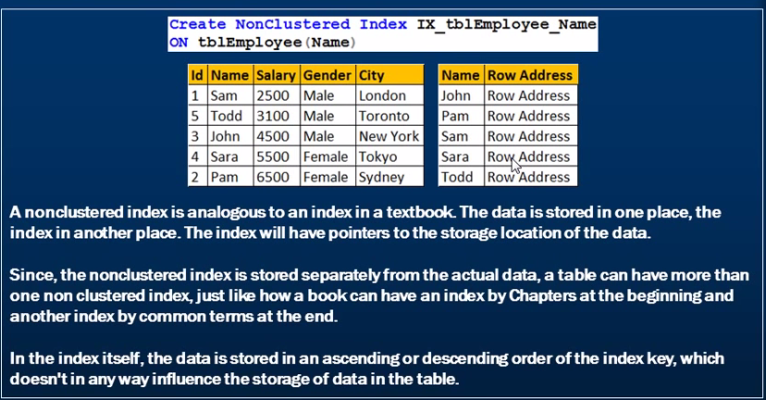
**1-Clustered Index**

****

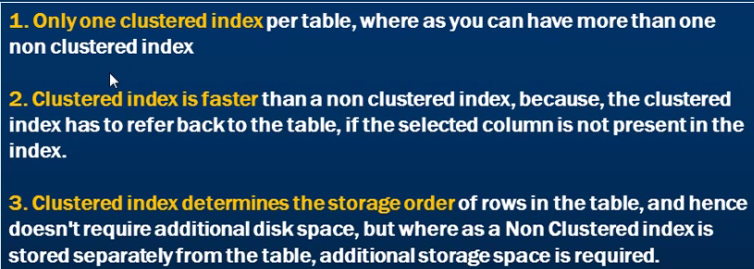
****

**Note:-**Problem with multiple composite clustered index is that it’s mixed up the Id of the table so that is the reason a table can have only one clustered index.

**2-Non-Clustered Index:-**

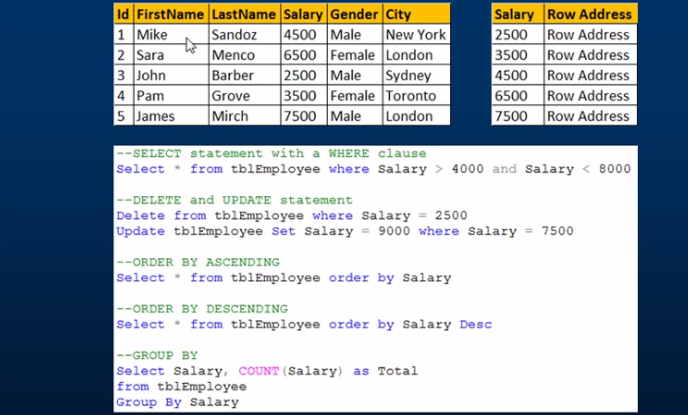
****

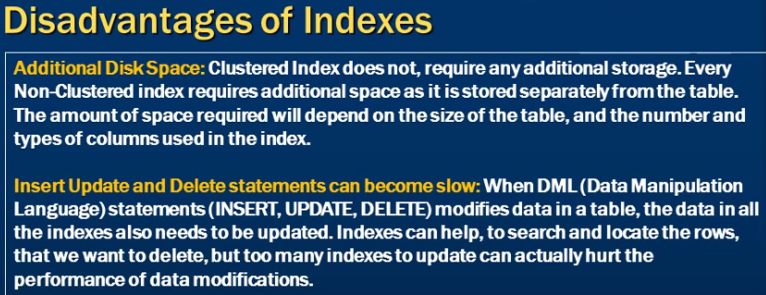
**Difference between Clustered and Non-Clustered Index:-**

****

**Adv. And Dis-Adv of Indexes:-**

**1-** Advantages of indexes of faster retrieval of data for the below query.





**Covering Query:-**



**Difference between Primary Key and Indexes:-**

A **PRIMARY KEY** is just a **constraint**that makes sure while inserting records you have UNIQUE and NOT NULL values in a particular or a set of columns, in a table.

An **Index**is a separate object in a database that help **quick retrieval** of records.

# SQL - SELF JOINS:-

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax

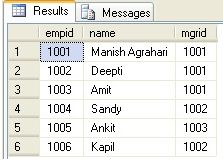
The basic syntax of SELF JOIN is as follows −

SELECT a.column\_name, b.column\_name...

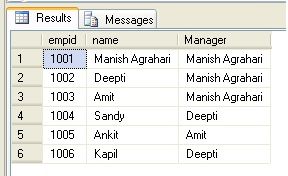
FROM table1 a, table1 b

WHERE a.common\_field = b.common\_field;

Here, the WHERE clause could be any given expression based on your requirement.

Ex:-  
  

SELECT e.empid, e.name, m.name  "Manager" FROM Emp e, Emp m WHERE e.mgrid=m.empid;

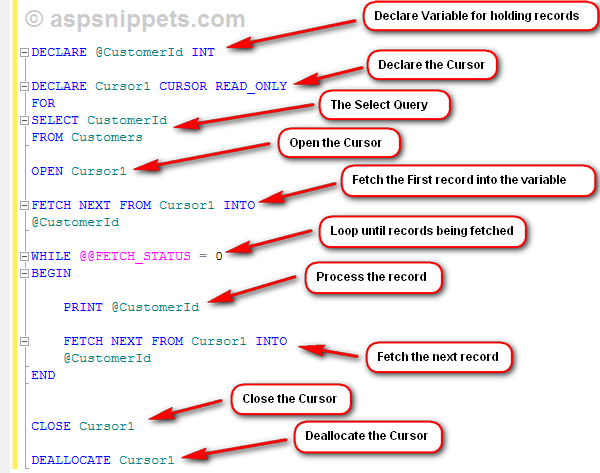


**Difference between cursor and trigger:-**

1-A trigger is a procedure (code segment) that is executed automatically when some specific events occur in a table/view of a database, while a cursor is a control structure used in databases to go through the database records. A cursor can be declared and used within a trigger. Cursors allow row-by-row prcessing of the resultsets.

A Cursor is always associated with a Select Query and it will process each row returned by the Select Query one by one.

**What is the syntax for writing Cursors in SQL Server:-**



 Below is the syntax for writing a Cursor. The very first thing is to declare some variables based on the columns you are fetching in your Select Query.

**Note**: In this article I am considering only READ\_ONLY Cursors.

Then you need to declare the Cursor by giving it a name and setting its type as READ\_ONLY and along with the FOR keyword you need to write the Select Query which will return the records you need to process.

Once the Cursor is setup, we need to open it using the OPEN command and then the first record is fetched and saved into the variable.

Whenever a record is fetched the @@FETCH\_STATUS has value 0 and as soon as all the records returned by the Select Query are fetched, its value changes to -1.

A Cursor is associated with a WHILE LOOP which executes until the @@FETCH\_STATUS has value 0.

Inside the WHILE LOOP, the processing is done for the current record and then again the next record is fetched and this process continues until @@FETCH\_STATUS is 0.

Finally the Cursor is closed and deallocated using CLOSE and DEALLOCATE commands respectively.

**Can we used DML operation inside function:-**

**Answer is No.**

Only READ-ONLY database access in functions. If DML operations would be allowed in functions, then functions would be pretty similar to stored procedures.

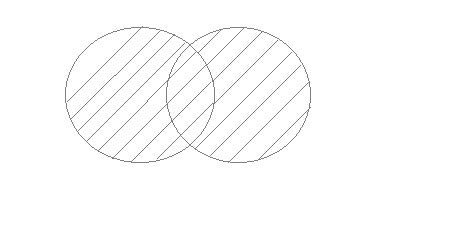
Functions cannot affect the state of the database which means we cannot perform insert, delete, update and create operations on the database. Stored Procedures can affect the state of the database by using insert, delete, update and create operations.

 Set Operation in SQL

SQL supports few Set operations to be performed on table data. These are used to get meaningful results from data, under different special conditions.

#### Union

UNION is used to combine the results of two or more Select statements. However it will eliminate duplicate rows from its result set. In case of union, number of columns and datatype must be same in both the tables.



#### Example of UNION

The **First** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | Abhi |
| 2 | Adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **Name** |
| 2 | Adam |
| 3 | Chester |

Union SQL query will be,

select \* from First

**UNION**

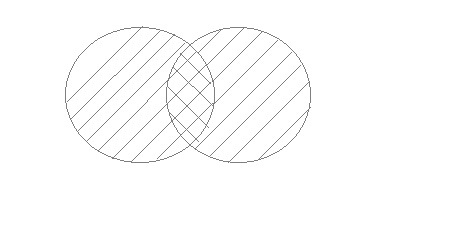
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 3 | Chester |

#### Union All

This operation is similar to Union. But it also shows the duplicate rows.



#### Example of Union All

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | Adam |
| 3 | Chester |

Union All query will be like,

select \* from First

**UNION ALL**

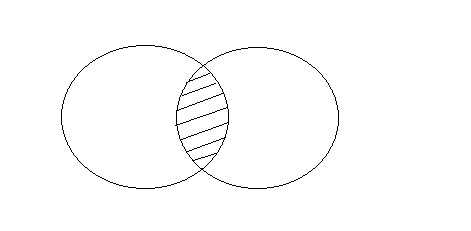
select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |
| 2 | Adam |
| 3 | Chester |

#### Intersect

Intersect operation is used to combine two SELECT statements, but it only retuns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same. MySQL does not support INTERSECT operator.



#### Example of Intersect

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | Adam |
| 3 | Chester |

Intersect query will be,

select \* from First

**INTERSECT**

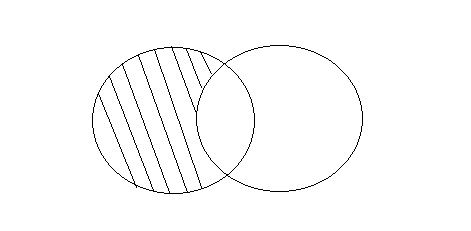
select \* from second

The result table will look like

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | Adam |

#### Minus

Minus operation combines result of two Select statements and return only those result which belongs to first set of result. MySQL does not support INTERSECT operator.



#### Example of Minus

The **First** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |
| 2 | Adam |

The **Second** table,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 2 | Adam |
| 3 | Chester |

Minus query will be,

select \* from First

**MINUS**

select \* from second

The result table will look like,

|  |  |
| --- | --- |
| **ID** | **NAME** |
| 1 | Abhi |

# EXCEPT vs NOT IN in SQL Server

"EXCEPT" operator was introduced in SQL SERVER 2005. This operator used to achieve Distinct and Not In queries. EXCEPT operator returns all distinct rows from left hand side table which does not exist in right hand side table.

On the other hand, "NOT IN" will return all rows from left hand side table which are not present in right hand side table but it will not remove duplicate rows from the result.

Example of EXCEPT vs NOT IN in SQL SERVER:

create table #tblsample (ProductId tinyint)

create table #tblsample2 (ProductId tinyint)

*--insert values to tbl 1*

insert into #tblsample values (1)

insert into #tblsample values (2)

insert into #tblsample values (3)

insert into #tblsample values (4)

*--insert values to tbl 2*

insert into #tblsample2 values (1)

*--SELECT*

select \* from #tblsample

select \* from #tblsample2

*--USE EXCEPT*

select \* from #tblsample

except

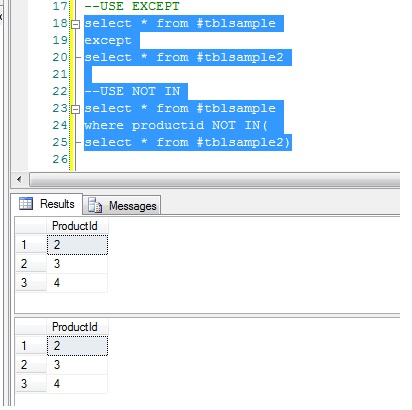
select \* from #tblsample2

*--USE NOT IN*

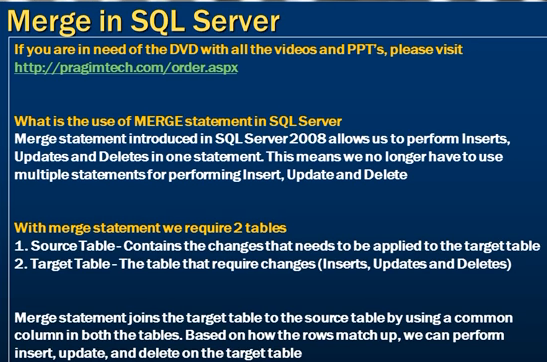
select \* from #tblsample

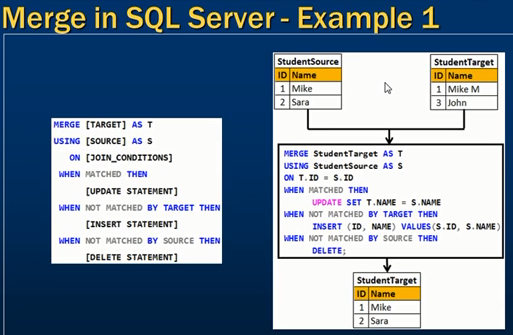
where productid NOT IN(

select \* from #tblsample2)



# The MERGE Statement in SQL Server 2008





# OUT and INOUT parameter

**OUT parameter**— You can only receive values from the procedure. Your cannot pass values to OUT parameters in a stored procedure call. After successful execution of the stored procedure, all the OUT parameters values will be assigned to their respective bound variables.

1. **IN parameter**— You can only send values to the stored procedure. In the stored procedure body, you can execute queries based on input value. This is the default parameter, and keyword IN is optional.
2. **INOUT parameter**— You can send and receive values from the procedure. This provides the benefits of both OUT and IN parameters using the single bound variable.

# [Why can we not execute a stored procedure inside a function in SQL Server](https://stackoverflow.com/questions/2896445/why-can-we-not-execute-a-stored-procedure-inside-a-function-in-sql-server)

We cannot execute a stored procedure inside a function, because a function is not allowed to modify database state, and stored procedures are allowed to modify database state.

Functions exist to simply calculate something, a value or a table result, nothing more than that. These functions can be called within a SELECT query for instance, e.g.

SELECT calculate\_something(a) FROM some\_table;

User defined functions aren’t allowed to execute any DML statements **except** SELECT; whereas stored procedures do have the ability to execute DML statements.

**why to use stored procedure**

Multiple users and client programs can perform operations on underlying database objects through a procedure, even if the users and programs do not have direct permissions on those underlying objects. The procedure controls what processes and activities are performed and protects the underlying database objects. This eliminates the requirement to grant permissions at the individual object level and simplifies the security layers.

--// Logged in as 'sa'

USE AdventureWorks;

GRANT SELECT ON Person.Address (AddressID, AddressLine1) to Matt;

GRANT UPDATE ON Person.Address (AddressLine1) to Matt;

--// Logged in as 'Matt'

SELECT \* from Person.Address; --// Fail

SELECT AddressID, AddressLine1 from Person.Address; --// Succeed

UPDATE Person.Address SET AddressLine1 = '#\_\_\_\_ 2700 Production Way'

WHERE AddressID = 497; --// Succeed

**What is sql transaction**

A transaction is a single unit of work. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.

What Is a Transaction?

A transaction is a set of operations performed so all operations are guaranteed to succeed or fail as one unit.

This involves two operations:

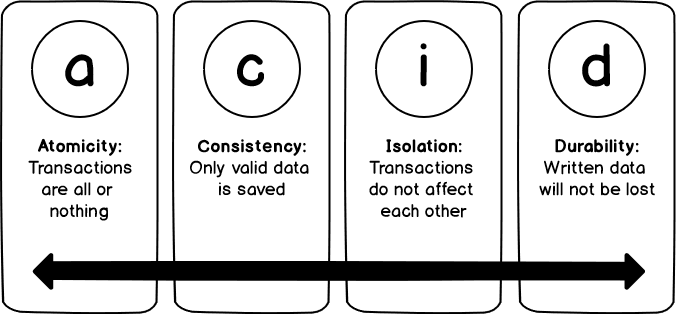
1. Deducting money from the checking account and  
     
   **Note**: in the USA a checking account is like a current account in India
2. Adding it to the savings account.

Both must succeed together and the changes must be committed to the accounts, or both must fail together and rolled back so that the accounts are maintained in a consistent state. Under no circumstances should money be deducted from the checking account but not added to the savings account (or vice versa), you would at least not want this to happen with the transactions occurring with your bank accounts.

By using a transaction concept, both the operations, namely debit and credit, can be guaranteed to succeed or fail together. So both accounts remain in a consistent state all the time.

ACID

After understanding the concept of the SQL Server Transaction, we can describe the Transaction using the four **ACID** properties. ACID is the acronym for **Atomicity**, **Consistency**, **Isolation** and **Durability**.



**Atomicity** means that the transaction will succeed, as one unit, if all the separate tasks succeed with no issue. On the other hand, the failure of any single task within this transaction leads to the overall transaction failure and rollback. In other words, Atomicity guarantees that the transaction is all or none.

**Consistency** guarantees that the transaction will not affect the database consistency and will leave it in a valid state by complying with all database rules, such as foreign keys and constraints, defined on the columns.

**Isolation** means that, each transaction has it is own boundary, that is separated from the other concurrently executing transactions, and will not be affected by these transactions’ operations.

**Durability** means that the result of the committed transaction that is written permanently to the database will not be lost even in the case of any abnormal system failure or termination, as there should be mechanism of recovering this data, that we will see later in the next articles of this series.

The following commands are used to control transactions.

* **COMMIT** − to save the changes.
* **ROLLBACK** − to roll back the changes.
* **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK.
* **SET TRANSACTION** − Places a name on a transaction.

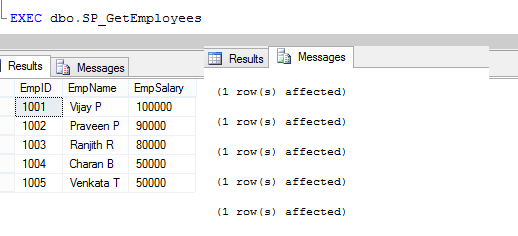
**How to improve Stored Procedure performance**

There are some below tips which will help to improve the SP performance.

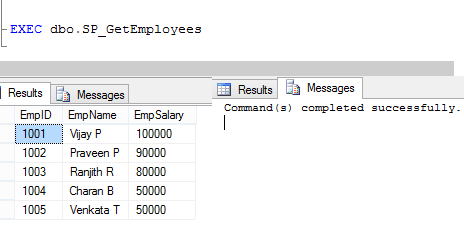
Use SET NOCOUNT ON

This statement is used to stop the message, which shows the number of rows affected by SQL statement like INSERT, UPDATE and DELETE.

Ex.



It will remove this extra overhead from the network.



Use EXISTS () instead of COUNT ()

This SQL optimization technique concerns the use of EXISTS (). If you want to check if a record exists, use EXISTS () instead of COUNT (). While COUNT () scans the entire table. This will give you better performance.

Ex. SELECT Count(1) FROM dbo.Employee

Ex. IF ( EXISTS (SELECT 1 FROM db.Employees))

BEGIN

END

Specify column names instead of using \* in SELECT statement

Try to avoid \*

1. **SELECT** \* **FROM** dbo.Employee **WITH**(NOLOCK)
2. **WHERE** Convert(**varchar**(12),ip.CreatedDate,101)=Convert(**varchar**(12),GETDATE(),101)

Best practice is to specify the column name.

1. **SELECT** EmpID,EmpName,EmpSalary
2. **FROM** dbo.Employee **WITH**(NOLOCK)
3. **WHERE** Convert(**varchar**(12),ip.CreatedDate,101)=Convert(**varchar**(12),GETDATE(),101)

Use Join query instead of sub-query and co-related subquery

Using JOIN is better for the performance than using subqueries or nested queries

Also, use minimum JOINS (as required) and check whether proper JOIN is used or not.

Create Proper Index

Proper indexing will improve the speed of the operations in the database.

Avoid temp temporary table

Temporary tables usually increase a query’s complexity. It’s suggested to avoid the temporary tables.

Use NO LOCK

Use NOLOCK will improve the performance of the select query

1. **SELECT** EmpID,EmpName,EmpSalary
2. **FROM** dbo.Employee **WITH**(NOLOCK)
3. **WHERE** Convert(**varchar**(12),ip.CreatedDate,101)=Convert(**varchar**(12),GETDATE(),101)

Don’t use functions in the WHERE clause

While writing select query, if you use the function in where condition, it will reduce the performance of your query. Try to avoid the function in where clause.

**What is SQL Server Profiler**

SQL Server Profiler is a tracing tool provided by Microsoft since SQL Server 2000 version. It is used to trace activities and operations executed on a specific SQL Server database engine or Analysis Service to be analyzed later.

SQL Server is responsible for two main operations:

1. **Tracing**: It can monitor all operations executed over an instance
2. **Replay**: It can rerun all operations logged in a trace later

Profiler is a standalone tool installed with SQL Server. You can access it from Windows Explorer or SQL Server Management Studio.

Under Windows 10, you can write SQL Server Profiler from the Start menu, and the application will show in the results:

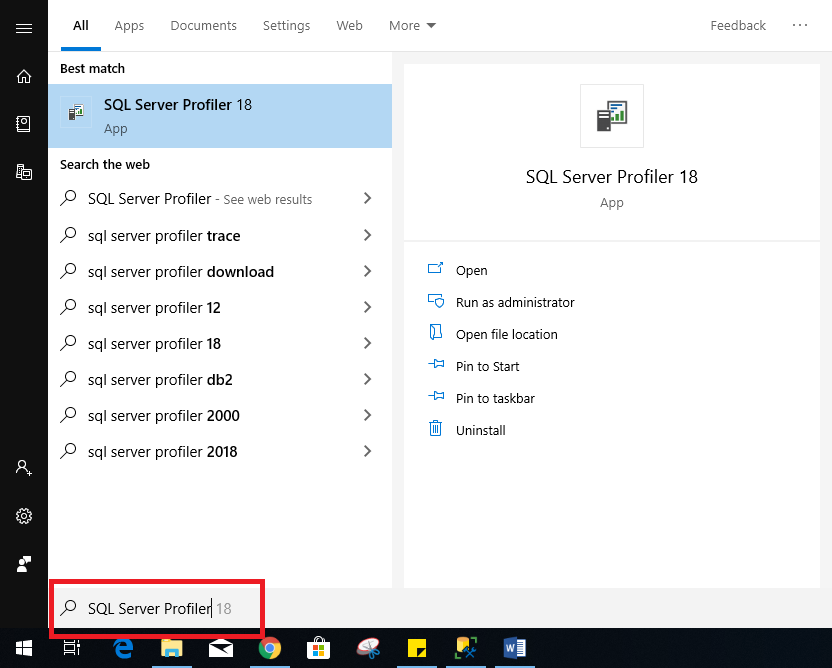


Figure 1 – Accessing Profiler from the Start menu

Or you can simply find a shortcut of this tool under the **Tools** menu inside the SQL Server Management Studio as shown in the image below:

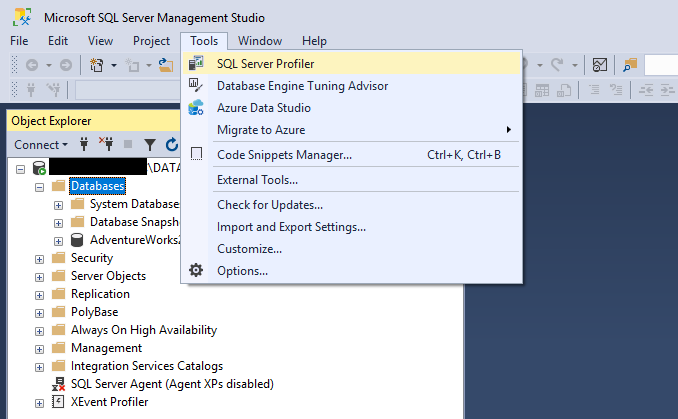
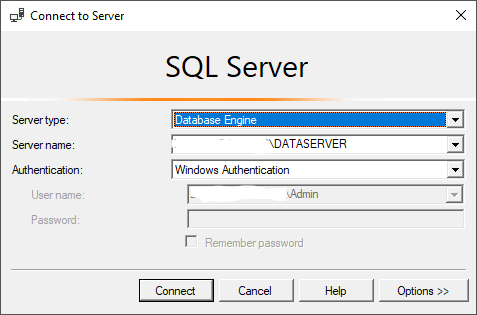
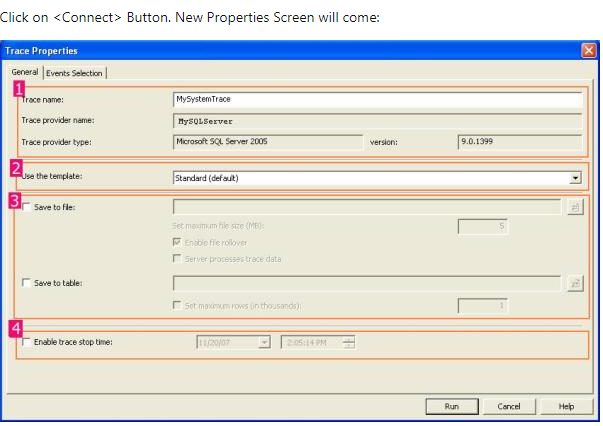


Figure 2 – Accessing SQL Server Profiler from Management Studio

## How to use Profiler?

When you open the Profiler, the authentication form is shown. You have to select whether you need to connect to an Analysis Service instance or a Database Engine. Then you should enter the instance name, the authentication type, and the credentials:





It has two selection tabs:

* **General:**It is used for general setting for Trace Database Engine.
* **Event:** It is used to add or remove some selected event for monitor.

In General Section (as given in Figure 1.1), it is divided into four sections.

**Section 1:**In this section, you have to just specify the name of your trace, Trace provider name and server name are predefined and based upon your SQL Server.

And it is not editable.

**Section 2:**It is the template section. You can choose different type of Templates based upon your requirements. It is the configuration for trace. By default, it is "Standard (Default)" templates. Others templates are T-SQL, T-SQL Duration, T-SQL Reply, T-SQL SPs, etc. You can create your own custom Templates by selecting different Events and Event Class. It is saved as ".tdf" Extension.

**Section 3:**This section is related to save your trace. Either as File (.trc) or in a database. as table. While clicking on Save to file check box, File save dialog box should open and you can save that file (with .trc extension).

If you check the "Save to Table", it will connect with your server and ask you to which database you want to save that trace table information.



**Section 4:**You can stop your trace on a particular time. Check the "Enable trace stop time" checkbox and give the time at which you want to stop track, SQL Server will automatically stop trace on that time.

Now Move To "Event Section" Tab.

Now we need to know some definition with respect to SQL Server Profiler.

### What is an Event?

An Event is an action or operation that is performed in your SQL Server 2005 Database Engine.

Some examples of Events are:

* + Transact-SQL SELECT, INSERT, UPDATE, and DELETE statements.
  + User login and logout
  + Execution of Stored procedures
  + Operation with cursor

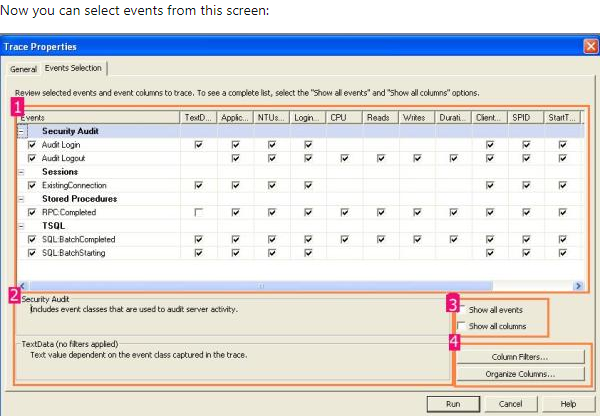
SQL Server profiler is able to trace all of these events and all these events are categories on a particular Event class.

### What is an Event Class?

Event class is a type of event that can be traced.

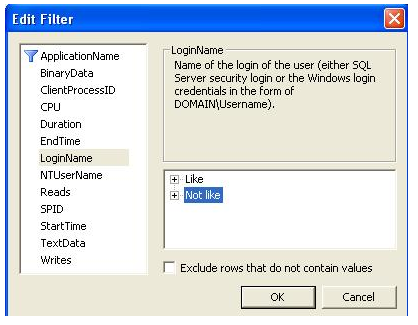
Some examples are:

* SQL: BatchCompleted
* SQL: Batch Starting
* Audit Login
* Audit Logout
* Lock: Acquired
* Lock: Released



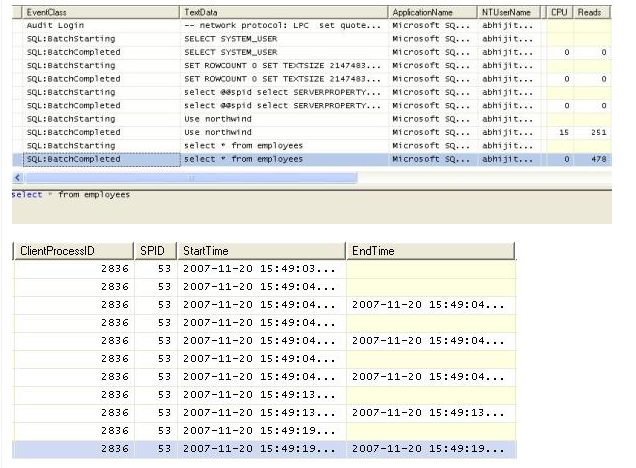
In section 1, we can select the proper check box based upon our requirement, section 2 will display the details of Selected events and Events class. If you check in the check box of section 3, you will get all the list of Events and Columns in Section 1.

Section 4 is something like customization. Just click on the "Column Filter Button". In this section, you can specify some condition (like or Not like).



By clicking on "Organize Column" button, you can change the sequence of order of selected events.

Now Click on the "Run" Button, then Trace window will come:



Using these windows, you will get the detailed time duration of a query and all other events information that you have selected.

You can save this result and use it in future. Or you can extract a particular query from the trace, just right click and click on "Extract Event Data". And save this as a SQL Script.

### Reply in SQL Server Profiler

SQL Server profiler has a Reply facility which has the ability to save a trace and replay it later.

Replay is useful to troubleshoot an application. Trace replay supports debugging by using Toggle Breakpoint and the Run to Cursor options on the SQL Server Profiler Replay menu.

Anything changed in SQL Server Management Studio will be traced by the SQL Profiler. So it can basically be used for database performance check. We also have "SQL Server Performance Monitor" to monitor the System and Server performance too.

**Rank function in Sqlserver**

Ranking functions provide a very good feature of assigning numbering to the records in the result set in SQL. Row\_Number in SQL is one of these functions available in that allows us to assign rankings or numbering to the rows of the result set data. Different values are assigned to different rows, based on the type of ranking function used. They are basically of the following types:

1. Row\_Number
2. Rank
3. Dense\_Rank
4. NTile

To understand this concept of ranking, we need to first understand the syntax of it. The syntax is :

1. Rank() Over ( {Partition\_By\_Clause} Order\_By\_Clause)

First of all, Rank() is the function to assign the numbering or ranking to the result set data. Next, the Over keyword is used along with the conditions specified in the '(' brackets (in other words the Partition and Order by clause), to decide how the ranking or numbering is started and implemented on the result set data. Here, these two keywords are also very important to be understood, before we proceed.

**Order\_By\_Clause :**This clause decides how the ranking or numbering of the data starts. For example, if we specify Order By EmployeeID desc, it will result in ordering the data in descending order by EmployeeID and then assign the rank to the records, starting from 1 to n, based on the type of ranking function we use (in other words rank, dense rank and so on).

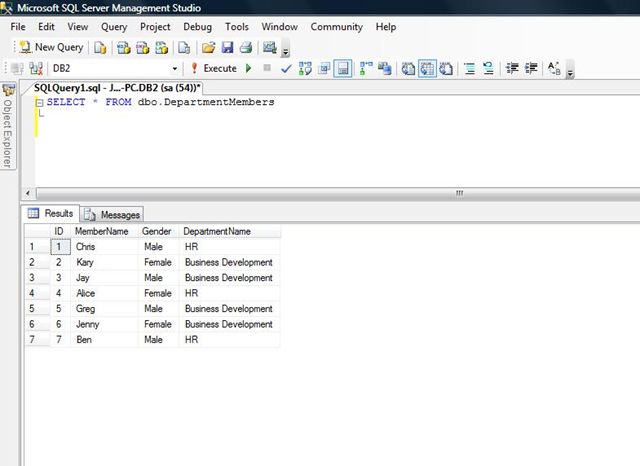
**Partition\_By\_Clause :** This clause basically provides the grouping functionality on the result set data. When used in combination with the order by clause, it results in grouping the data and then the ranking of the rows is done, based on the type of ranking function we use (in other words rank, dense rank and so on).

So its the over keyword, along with these two clauses, that handle the ranking process of the records, when using the ranking functions.

The Row\_Number function

This function works by assigning continuous ranking to the records, without skipping any number in the result set, whether it is partitioned or not. At the end of the discussion, we will see what we mean by continuous ranking and not skipping any record.

For this, we will create a table named DepartmentMembers, that stores names of the members, along with their DepartmentName. I know this is not a perfect database structure, but just a sample table with some records inserted into it, to discuss the concept. So our setup table will be like the following:



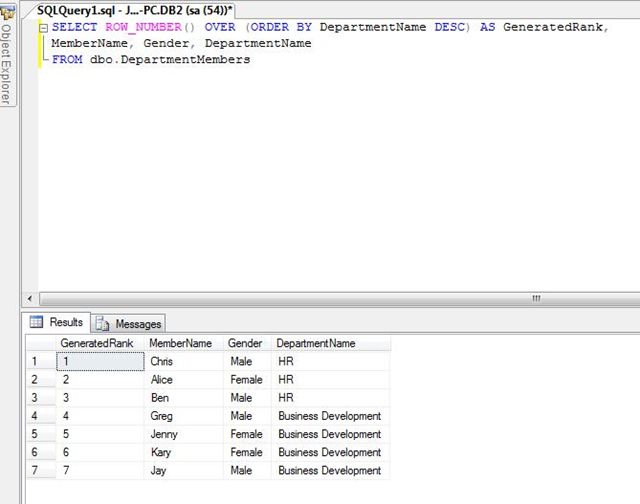
Next, we will be applying the Row\_Number function with each two cases and will discuss them one by one. In the first case, we will apply only the order by clause and in the second case, we will add the partition by clause, along with the order by clause.

The main point here is that the order by clause is a **must** for applying the ranking functions. So let's start.

**Case 1: Ranking entire result set using Order By clause**

Based on the syntax above and data created in the sample table, we will apply the Row\_Number function now. So our query becomes:

Now execute this query and see the results.

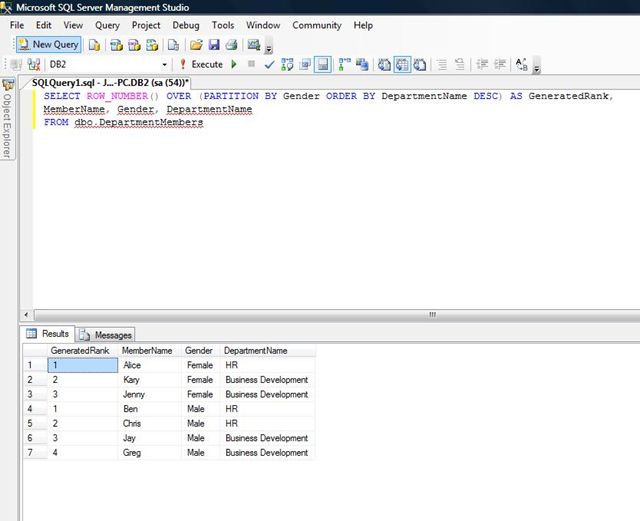


What this query has done is that, it selects the results from the table and orders them by DepartmentName in descending order. Further, it generates the rank or numbering for the records as we can see in the GeneratedRank column. So, simple numbering was generated for the records.

**Case 2: Ranking partitioned result set using Partition By Clause**

Next, we will add the Partition By clause to it. So the query now changes to:

Now run the query and see the results.



This time, the partition by clause grouped the results based on the Gender, ordered the data of the groups by DepartmentName in descending order and then finally applied the numbering or rank on the results, based on the ranking function used.

# Differences among RANK, DENSE\_RANK and ROW\_NUMBER Functions

**Preparing Dummy Data**

Execute the following script to create a database called ShowRoom and containing a table called Cars (that contains 15 random records of cars):

CREATE Database ShowRoom;

GO

USE ShowRoom;

CREATE TABLE Cars

(

id INT,

name VARCHAR(50) NOT NULL,

company VARCHAR(50) NOT NULL,

power INT NOT NULL

)

USE ShowRoom

INSERT INTO Cars

VALUES

(1, 'Corrolla', 'Toyota', 1800),

(2, 'City', 'Honda', 1500),

(3, 'C200', 'Mercedez', 2000),

(4, 'Vitz', 'Toyota', 1300),

(5, 'Baleno', 'Suzuki', 1500),

(6, 'C500', 'Mercedez', 5000),

(7, '800', 'BMW', 8000),

(8, 'Mustang', 'Ford', 5000),

(9, '208', 'Peugeot', 5400),

(10, 'Prius', 'Toyota', 3200),

(11, 'Atlas', 'Volkswagen', 5000),

(12, '110', 'Bugatti', 8000),

(13, 'Landcruiser', 'Toyota', 3000),

(14, 'Civic', 'Honda', 1800),

(15, 'Accord', 'Honda', 200

**RANK Function**

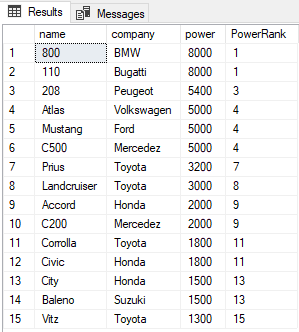
The RANK function is used to retrieve ranked rows based on the condition of the ORDER BY clause. For example, if you want to find the name of the car with third highest power, you can use RANK Function.  
Let’s see RANK Function in action:

SELECT name,company, power,

RANK() OVER(ORDER BY power DESC) AS PowerRank

FROM Cars

The script above finds and ranks all the records in the Cars table and orders them in order of descending power. The output looks like this:



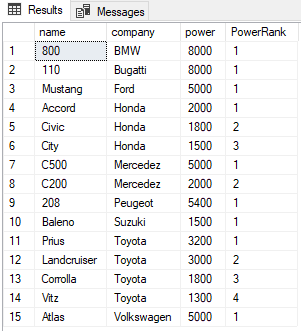
The PowerRank column in the above table contains the RANK of the cars ordered by descending order of their power. An interesting thing about the RANK function is that if there is a tie between N previous records for the value in the ORDER BY column, the RANK functions skips the next N-1 positions before incrementing the counter. For instance, in the above result, there is a tie for the values in the power column between the 1st and 2nd rows, therefore the RANK function skips the next (2-1 = 1) one record and jumps directly to the 3rd row.  
The RANK function can be used in combination with the PARTITION BY clause. In that case, the rank will be reset for each new partition. Take a look at the following script:

SELECT name,company, power,

RANK() OVER(PARTITION BY company ORDER BY power DESC) AS PowerRank

FROM Cars

In the script above, we partition the results by company column. Now for each company, the RANK will be reset to 1 as shown below:



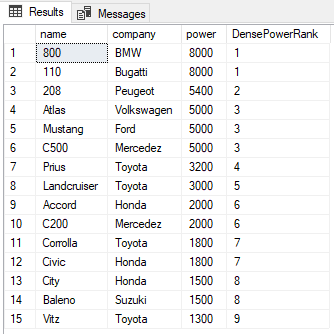
**DENSE\_RANK Function**

The DENSE\_RANK function is similar to RANK function however the DENSE\_RANK function does not skip any ranks if there is a tie between the ranks of the preceding records. Take a look at the following script.

SELECT name,company, power,

RANK() OVER(PARTITION BY company ORDER BY power DESC) AS PowerRank

FROM Cars

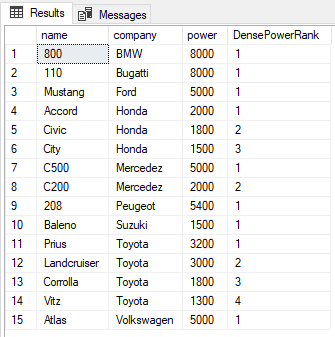


You can see from the output that despite there being a tie between the ranks of the first two rows, the next rank is not skipped and has been assigned a value of 2 instead of 3. As with the RANK function, the PARTITION BY clause can also be used with the DENSE\_RANK function as shown below:

SELECT name,company, power,

DENSE\_RANK() OVER(PARTITION BY company ORDER BY power DESC) AS DensePowerRank

FROM Cars



**Similarities between RANK, DENSE\_RANK, and ROW\_NUMBER Functions**

* 1. All of them require an order by clause.
  2. All of them return an increasing integer with a base value of 1.
  3. When combined with a PARTITION BY clause, all of these functions reset the returned integer value to 1 as we have seen.

SQL Server Triggers on Views

There are two kinds of DML triggers the FOR (or AFTER) trigger and the INSTEAD OF trigger, but the only one you can use with views are INSTEAD OF triggers.

"AFTER" triggers are executed after the triggering events like INSERT, UPDATE or DELETE events, where as the "INSTEAD OF" triggers are fired in place of the triggering events. INSTEAD OF triggers are usually used to correctly update views that are based on multiple tables.

**Steps to be followed to use INSTEAD OF INSERT trigger**So, first, let's create two tables named Student & Section.

**Table Student SQL script**

1. **CREATE** **TABLE** Student
2. (
3. Id **int** **Primary** **Key**,
4. **Name** nvarchar(30),
5. Gender nvarchar(10),
6. SectionId **int**
7. )

**Table Section SQL script**

1. **CREATE** **TABLE** **Section**
2. (
3. SecId **int** **Primary** **Key**,
4. SecName nvarchar(20)
5. )

Then, insert some dummy records in both the tables.

1. **Insert** **into** **Section** **values** (1,'Sec1')
2. **Insert** **into** **Section** **values** (2,'Sec2')
3. **Insert** **into** **Section** **values** (3,'Sec3')
4. **Insert** **into** **Section** **values** (4,'Sec4')
5. **Insert** **into** Student **values** (1,'Satya1', 'Male', 3)
6. **Insert** **into** Student **values** (2,'Satya2', 'Male', 2)
7. **Insert** **into** Student **values** (3,'Satya3', 'Female', 1)
8. **Insert** **into** Student **values** (4,'Satya4', 'Male', 4)
9. **Insert** **into** Student **values** (5,'Satya5', 'Female', 1)
10. **Insert** **into** Student **values** (6,'Satya6', 'Male', 3)

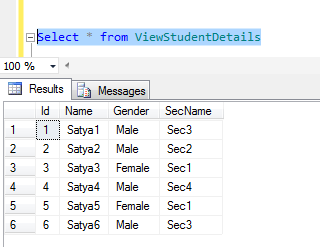
Let's create a View based on the above mentioned tables.  
  
The View should return Student Id, Name, Gender, SecName columns. So, it is obviously based on multiple tables.

**Script to create the View**

1. **Create** **view** ViewStudentDetails
2. **as**
3. **Select** Id, **Name**, Gender, SecName
4. **from** Student
5. join **Section**
6. **on** Student.SectionId = **Section**.SecId

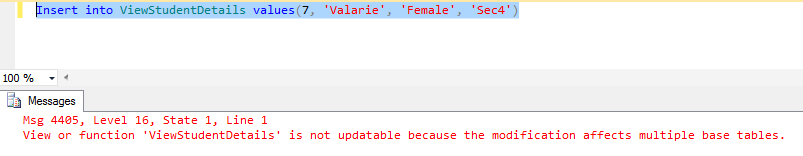
Then, executethe below SQL query. The output will be shown like this.

1. **Select** \* **from** ViewStudentDetails



Let's try to insert a row into the View, ViewStudentDetails, by executing the following query. An error will be raised showing "View or function ViewStudentDetails is not updatable because the modification affects multiple base tables".

1. **Insert** **into** ViewStudentDetails **values**(7, 'Valarie', 'Female', 'Sec4')

  
So, inserting a  row into a View that is based on multipe tables, raises an error by default. Let's create an INSTEAD OF INSERT trigger on the ViewStudentDetails view.

**Script to create INSTEAD OF INSERT trigger**

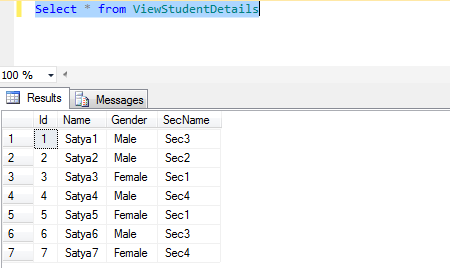
1. **Create** **trigger** TR\_ViewStudentDetails\_InsteadOfInsert
2. **on** ViewStudentDetails
3. **Instead** **Of** **Insert**
4. **as**
5. **Begin**
6. **Declare** @SecId **int**
8. **Select** @SecId = SecId
9. **from** **Section**
10. join inserted
11. **on** inserted.SecName = **Section**.SecName
13. if(@SecId **is** null)
14. **Begin**
15. Raiserror('Invalid Section Name. Statement Terminated OK', 16, 1)
16. **return**
17. **End**
19. **Insert** **into** Student(Id, **Name**, Gender, SectionId)
20. **Select** Id, **Name**, Gender, @SecId
21. **from** inserted
22. **End**

Now, let's execute the INSERT query using existing section name. The data can be successfully inserted.

1. **Insert** **into** ViewStudentDetails **values**(7, 'Satya7', 'Female', 'sec4')

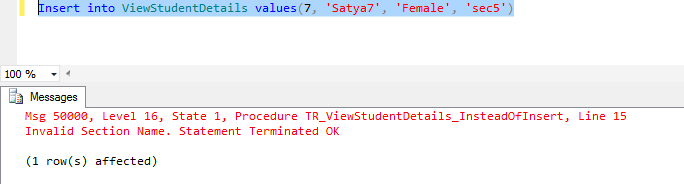
To get the new data, execute the following query.

1. **Select** \* **from** ViewStudentDetails



Now, let's execute the insert query using new section name. We can see that the data can't be successfully inserted.

1. **Insert** **into** ViewStudentDetails **values**(7, 'Satya7', 'Female', 'sec5')



Here, we are getting one error message as we used in Raiserror() function using Instead of insert trigger.

**What Is Raiserror() function In Sql server.**  
The Raiserror() function is used to raise a custom error  when the SectionName provided in the insert query does not exist or is not found, like "sec5" in my previous query. 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.

**SSIS**

stands for Sql Server Integration Services. The key power of SSIS is its data transformation and migration capability. When building a data warehouse or a data mart, the data needs to be extracted out of the various transactional systems and flat files, transformed and loaded to where it can then be analyzed and reported on. The data extraction, transformation and loading are known as ETL and is a common term in data migration and Business Intelligence.

**SSRS**

stands for Sql Server Reporting Services. Once data is in its final state, either in the native transactional system or transformed into a datamart or datawarehouse, SSRS provides the tools necessary to create reports to better understand your data. These three tools are often used together to support your data analysis needs and come part of Microsoft Sql Server and the Microsoft Business Intelligence package.

**Can we use where and having together**

Of course you can since they have different purposes. Where clause is applied on columns, whereas, a Having clause is applied on group functions.

For example, for the given table find all department numbers where average salary of all the CLERKs is more than 5000.

Emp(empno, ename, designation, salary, deptno)

Answer:

Select deptno From emp

Where designation = 'CLERK'

Group by deptno Having avg(Sal) > 5000;

Yes you can use WHERE and HAVING in the same SQL statement but with one precondition. - WHERE should be placed before GROUP BY and HAVING clause.

**What Is Candidate Key**

 A **Candidate Key** can be any column or a combination of columns that can qualify as unique **key in** database. There can be multiple **Candidate Keys in** one table.

For example in table we have primary key which we will use for uniquely fetch the data from the database same way in same table we have other column is there which we will use to identify the data like mobile No., Email Id by defining the that column as unique constrain.

**What is Composite Key**

A composite key is a combination of two or more columns in a table that can be used to uniquely identify each row in the table when the columns are combined uniqueness is guaranteed, but when it taken individually it does not guarantee uniqueness.

Sometimes more than one attributes are needed to uniquely identify an entity. A primary key that is made by the combination of more than one attribute is known as a composite key.

Note for the primary key we can create the composite key by using the unique constrain also.

**Example(Composite key by using unique constrain)**

CREATE TABLE accounts (

acc\_num INTEGER,

acc\_type INTEGER,

acc\_descr CHAR(20),

Unique(acc\_num, acc\_type));

**Example(Composite key by using primary constrain and foreign key)**

CREATE TABLE accounts (

acc\_num INTEGER,

acc\_type INTEGER,

acc\_descr CHAR(20),

PRIMARY KEY (acc\_num, acc\_type));

CREATE TABLE sub\_accounts (

sub\_acc INTEGER PRIMARY KEY,

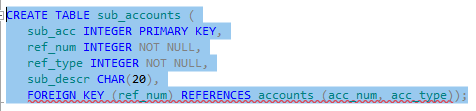
ref\_num INTEGER NOT NULL,

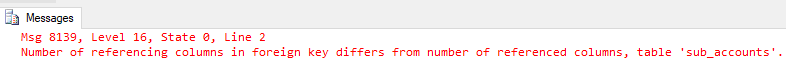
ref\_type INTEGER NOT NULL,

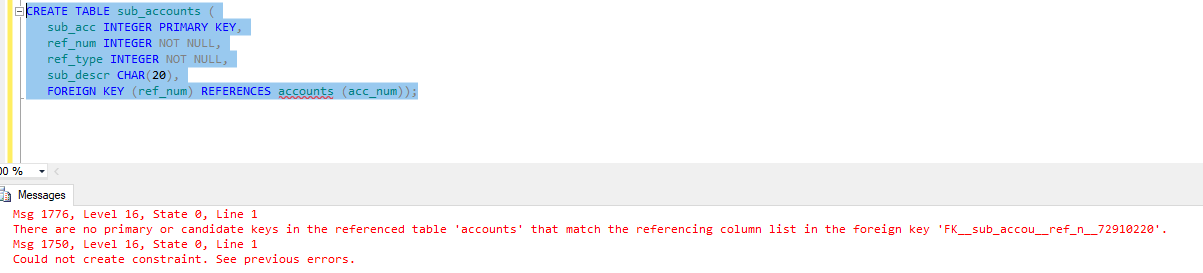
sub\_descr CHAR(20),

FOREIGN KEY (ref\_num, ref\_type) REFERENCES accounts (acc\_num, acc\_type));

**Note-** If we define the primary key as composite key then in child table we should use the all the composite key column for defining foreign key other wise we will get error. Below query will give the error.







# How to Find the Last Inserted Record in SQL Server

While we work with the table in SQL Server database, we set identity column that act as an auto increment column in table to increase column ID value whenever new record is inserted. Suppose we want to insert a name of the employee in the table ‘Employees’, we will do that using the below command:

INSERT INTO Employees (FirstName) VALUES (‘Mellisa’)

Now, in order to get the lasted inserted record ID, we can use the following options:

**SELECT @@IDENTITY**

* It returns the last IDENTITY value produced on a connection, regardless of the table that produced the value and of the scope of the statement that produced the value.
* It is used to give the last identity value generated by the statement.
* If the statement did not affect any tables with identity columns, this command returns NULL
* If the table contains multiple rows generating multiple identity values, @@IDENTITY returns the last identity value generated.
* Though @@IDENTITY is limited to current session, it is not limited to current scope. Even if trigger on the table caused identity to be created, you will get identity that was last created, even if it is a trigger.

Ex- SELECT @@IDENTITY

1. **SELECT SCOPE\_IDENTITY()**
   * As the name suggests, it will return the last identity produced on a connection and by statement in same scope, regardless of the table that produced the value.
   * It is limited to the current scope and in current session as well.
   * It will return the last identity that was explicitly created, rather than any identity created by trigger or user-defined function.

Ex- SELECT SCOPE\_IDENTITY()

1. **SELECT IDENT\_CURRENT(‘TableName’)**
   * It returns the last identity value produced in a table, regardless of the connection and the scope of the statement that created the value.
   * It is not limited by scope and session but is limited to a specified table.
   * It will return the identity value generated for the specific table in any session or any scope.

Ex-SELECT IDENT\_CURRENT('Tmp\_Names')

**Find nth highest salary in sql server | sql query to find 2nd, 3rd highest salary**

**Step 1**

Select employeesalary from tableemployee

Order by employeesalary desc

**Step 2**

Select top 2 employeesalary from tableemployee

Order by employeesalary desc

**Step 3 2nd hightest**

Select top 1 \* from (Select top 2 employeesalary from tableemployee

Order by employeesalary desc) as innerquery

Order by employeesalary asc

Employeesalary

3000 ,4000.1000

This is a very common SQL Server Interview Question. There are several ways of finding the nth highest salary.

By the end of this video, we will be able to answer all the following questions as well.

How to find nth highest salary in SQL Server using a Sub-Query

How to find nth highest salary in SQL Server using a CTE

How to find the 2nd, 3rd or 15th highest salary

Let's use the following Employees table for this demo

Use the following script to create Employees table

Create table Employees

(

ID int primary key identity,

FirstName nvarchar(50),

LastName nvarchar(50),

Gender nvarchar(50),

Salary int

)

GO

Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000)

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000)

Insert into Employees values ('Steve', 'Pound', 'Male', 45000)

Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000)

Insert into Employees values ('Philip', 'Hastings', 'Male', 45000)

Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000)

Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000)

Insert into Employees values ('John', 'Stanmore', 'Male', 80000)

GO

To find the highest salary it is straight forward. We can simply use the Max() function as shown below.

Select Max(Salary) from Employees

To get the second highest salary use a sub query along with Max() function as shown below.

Select Max(Salary) from Employees where Salary [ (Select Max(Salary) from Employees)

To find nth highest salary using Sub-Query

SELECT TOP 1 SALARY

FROM (

SELECT DISTINCT TOP N SALARY

FROM EMPLOYEES

ORDER BY SALARY DESC

) RESULT

ORDER BY SALARY

To find nth highest salary using CTE

WITH RESULT AS

(

SELECT SALARY,

DENSE\_RANK() OVER (ORDER BY SALARY DESC) AS DENSERANK

FROM EMPLOYEES

)

SELECT TOP 1 SALARY

FROM RESULT

WHERE DENSERANK = N

To find 2nd highest salary we can use any of the above queries. Simple replace N with 2.

Similarly, to find 3rd highest salary, simple replace N with 3.

Please Note: On many of the websites, you may have seen that, the following query can be used to get the nth highest salary. The below query will only work if there are no duplicates.

WITH RESULT AS

(

SELECT SALARY,ss

ROW\_NUMBER() OVER (ORDER BY SALARY DESC) AS ROWNUMBER

FROM EMPLOYEES

)

SELECT SALARY

FROM RESULT

WHERE ROWNUMBER = 3

For paramterrised :

