**Advanced Aggregation Features**

**Ranking:**

* Ranking is done in conjunction with an order by specification.
* Suppose we are given a relation   
   *student\_grades(ID, GPA)*giving the grade-point average of each student
* Find the rank of each student.

**select** *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s\_rank*  
 **from** *student\_grades*

* An extra **order by** clause is needed to get them in sorted order

**select** *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s\_rank*  
 **from** *student\_grades* **order by** *s\_rank*

* Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  + **dense\_rank** does not leave gaps, so next dense rank would be 2
* Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

**select** *ID*, (1 + (**select count**(\*) **from** *student\_grades B* **where** *B*.*GPA* > *A*.*GPA*)) **as** *s\_rank* **from** *student\_grades A* **order by** *s\_rank*;

**Ranking can be done within partition of the data.**

* “Find the rank of students within each department.”

**select** *ID*, *dept\_name*,  
 **rank** () **over** (**partition by** *dept\_name* **order by** *GPA* **desc**)   
 **as** *dept\_rank* **from** *dept\_grades* **order by** *dept\_name*, *dept\_rank*;

* Multiple **rank** clauses can occur in a single **select** clause.
* Ranking is done *after* applying **group by** clause/aggregation
* Can be used to find top-n results
  + More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

**Other ranking functions:**

* + **percent\_rank** (within partition, if partitioning is done)
  + **cume\_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  + **row\_number** (non-deterministic in presence of duplicates)
* SQL:1999 permits the user to specify **nulls first** or **nulls last**

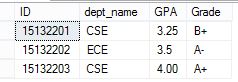
**select** *ID*, **rank** ( ) **over** (**order by** *GPA* **desc nulls last**) **as** *s\_rank*  **from** *student\_grades*

* For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.
* Divides into the each group and group will count ntile() function.

E.g..

**select** *ID*, **ntile**(4) **over** (**order by** *GPA* **desc**) **as** *quartile* **from** *student\_grades;*

**Ranking Consider Table Example:**

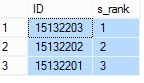


1. **Find the rank of each student**

Solution:

select ID, rank() over (order by GPA desc) as s\_rank from dept\_grades

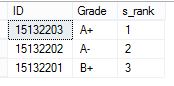
Output:



1. **Find the rank and grade of each student**

select ID,Grade, (1 + (select count(\*) from dept\_grades B

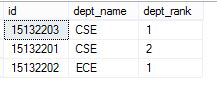
where B.GPA > A.GPA)) as s\_rank from dept\_grades A order by s\_rank;



1. **“Find the rank of students within each department.”**

select id, dept\_name, rank () over (partition by dept\_name order by GPA desc) as

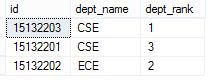
dept\_rank from dept\_grades order by dept\_name, dept\_rank;



select id, dept\_name, rank () over ( order by GPA desc) as dept\_rank from dept\_grades

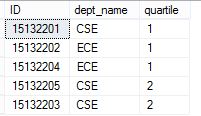
order by dept\_name, dept\_rank;

Output:

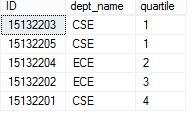


**ntile():**

select ID,dept\_name, ntile(2) over (order by GPA asc) as quartile from dept\_grades;



select ID,dept\_name, ntile(4) over (order by GPA desc) as quartile from dept\_grades;



**Windowing:**

* Window functions belong to a type of function known as a ‘set function’, which means a function that applies to a set of rows. The word ‘window’ is used to refer to the set of rows that the function works on.
* Used to smooth out random variations.
* E.g., moving average: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
* Window specification in SQL:
  + Given relation *sales(date, value)*

select IName, sum(sizeSF) over (order by IName rows 1 preceding)

from location

select TID, sum(Salary) over (order by TID ROWS BETWEEN 1 preceding and 1 following ) from teacher

**Examples of other window specifications:**

* + between rows unbounded preceding and current
  + rows unbounded preceding
  + range between 10 preceding and current row
    - All rows with values between current row value –10 to current value
  + range interval 10 day preceding
    - Not including current row
* Can do windowing within partitions
* E.g., Given a relation *transaction* (*account\_number, date\_time, value*), where value is positive for a deposit and negative for a withdrawal
  + **“Find total balance of each account after each transaction on the account”**

select *account\_number, date\_time*, sum (*value*) over  
 (partition by *account\_number* order by *date\_time* rows unbounded preceding)  
 as *balance* from *transaction* order by *account\_number, date\_time*

SELECT Dept, Salary,

SUM(Salary) OVER(PARTITION BY Dept ) AS "Sum",

AVG(Salary) OVER(PARTITION BY Dept) AS "Avg",

COUNT(Salary) OVER(PARTITION BY Dept) AS "Quantity"

FROM teacher

SELECT Dept, Salary,

SUM(Salary) OVER(PARTITION BY Dept order by TID\_rows unbounded preceding) AS "Sum",

AVG(Salary) OVER(PARTITION BY Dept) AS "Avg",

COUNT(Salary) OVER(PARTITION BY Dept) AS "Quantity"

FROM teacher

**Procedural Constructs in SQL**

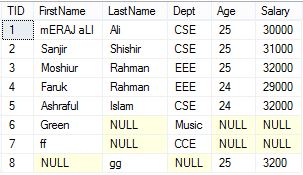
**Procedural Extensions and Stored Procedures:**

* SQL provides a module language
  + Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
* Stored Procedures
  + Can store procedures in the database
  + then execute them using the call statement
  + Permit external applications to operate on the database without knowing about internal details.

**Functions and Procedures:**

* SQL:1999 supports functions and procedures
  + Functions/procedures can be written in SQL itself, or in an external programming language.
  + Functions are particularly useful with specialized data types such as images and geometric objects.
    - Example: functions to check if polygons overlap, or to compare images for similarity.
  + Some database systems support table-valued functions, which can return a relation as a result.
* SQL:1999 also supports a rich set of imperative constructs, including
  + Loops, if-then-else, assignment
* Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

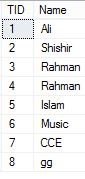
**Example of Storage Procedure:**

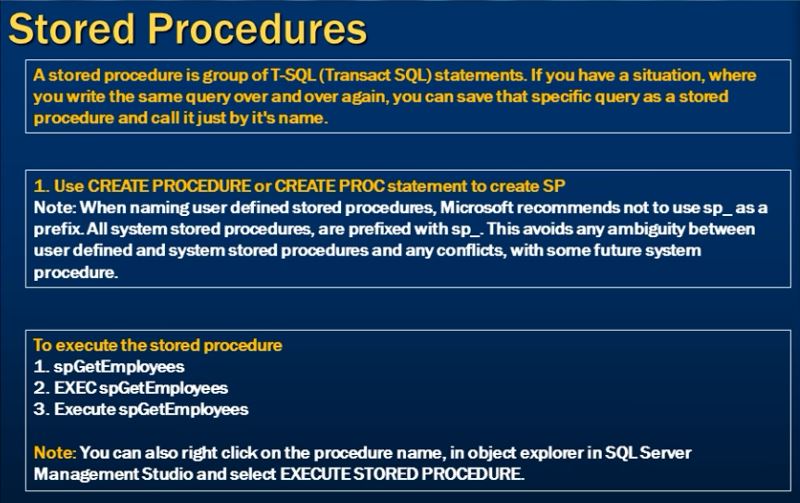


**COALESCE ( ) Function** – Returns the first NON NULL value

Select TID, coalesce (LastName, dept) as Name from Teacher

Output:





**Create procedure:**

create procedure spEmployee

as begin

select firstName,Salary from Teacher

end

Call This **procedure:**

spEmployee;

**Short curt** :

create proc spGetEmployee

as begin

select firstName,Salary from Teacher

end

Call This **procedure:**

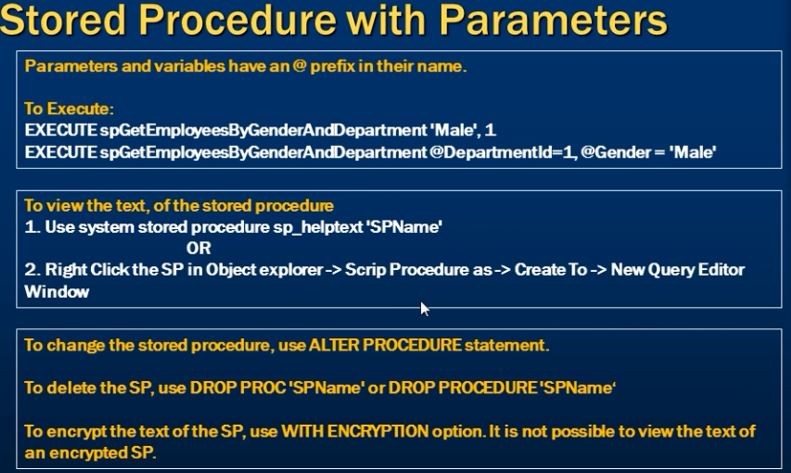
spGetEmployee;

or

Exec spGetEmployee

or

execute spGetEmployee



**Create procedure with parameter:**

create proc spGetEmployeeWithParameter

@Dept1 varchar(50),

@Age1 int

As begin

select firstname ,dept,age,Salary from teacher where Age=@Age1 and Dept=@Dept1

end

**Call This procedure:**

spGetEmployeeWithParameter 'CSE',25

**To view the text of the stored procedure:** To show what are write to create any procedure

sp\_helptext spGetEmployeeWithParameter;

**If we want to Encryption this procedure to write this:**

Create proc spGetEmployeeWithParameter

@Dept1 varchar(50),

@Age1 int

with Encryption

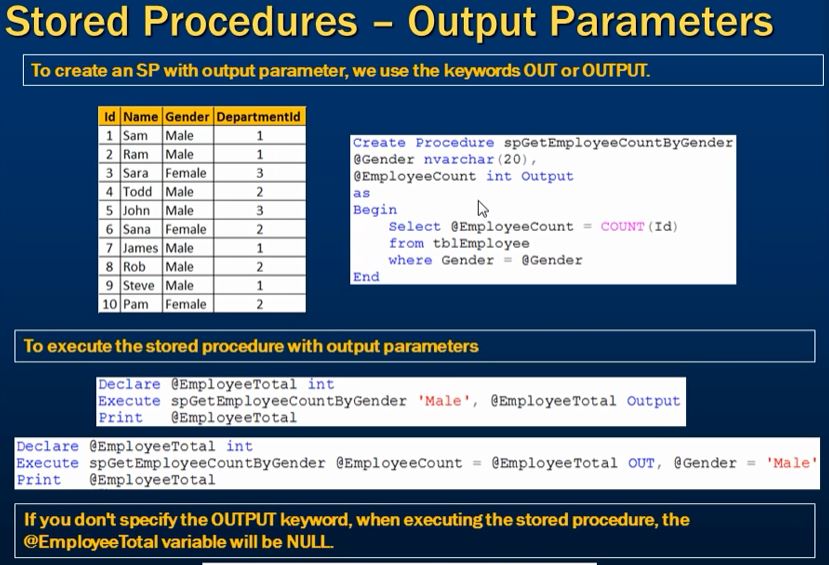
As

begin

select firstname ,dept,age,Salary from teacher where Age=@Age1 and Dept=@Dept1

end

If we use this Encryption we cannot modify or newly show create procedure.



**Create procedure With Parameter:**

create proc spGetTeacherCountByDept

@Dept1 varchar(50),

@TeacherCount int output

As begin

select @TeacherCount=count(TID ) from teacher

end

**To Execute Output Procedure:**

Declare @EmployeeTotal int

Exec spGetTeacherCountByDept 'CSE',@EmployeeTotal output

print @EmployeeTotal

In this we write to declare a variable @EmployeeTotal because we have return something. And this variable catch this value. So that to declare this variable.In this query we write @EmployeeTotal output,

If we not write the output beside @EmployeeTotal they cannot received the value.

Declare @EmployeeTotal int

Exec spGetTeacherCountByDept 'CSE' ,@EmployeeTotal

if (@EmployeeTotal is null)

print '@EmployeeTotal is null'

else

print '@EmployeeTotal is not null'

print @EmployeeTotal

**Output is:**

@EmployeeTotal is null

If we write the output beside @EmployeeTotal they cannot received the value.

Declare @EmployeeTotal int

Exec spGetTeacherCountByDept 'CSE' ,@EmployeeTotal output

if(@EmployeeTotal is null)

print '@EmployeeTotal is null'

else

print '@EmployeeTotal is not null'

print @EmployeeTotal

**Output is:**

@EmployeeTotal is not null

3

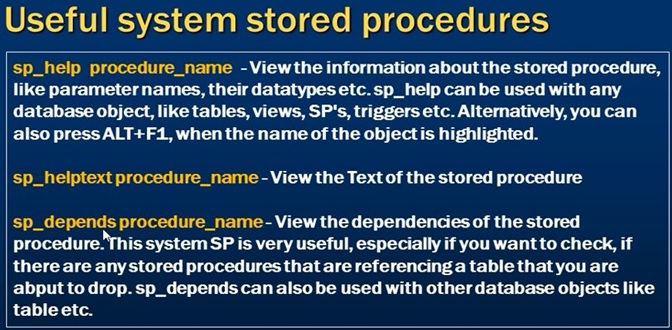
**We Run this Another way:**

Declare @EmployeeTotal int

Exec spGetTeacherCountByDept @TeacherCount =@EmployeeTotal output,

@Department = 'CSE'

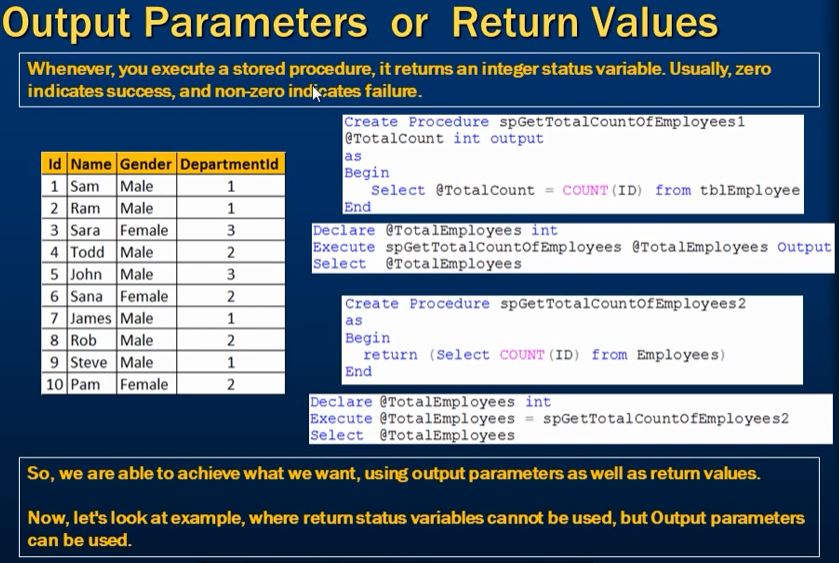
print @EmployeeTotal



sp\_help teacher

sp\_helptext spGetEmployeeWithParameter

sp\_depends spGetEmployeeWithParameter

****

**Create Return Value with procedure**

create proc spGettotalTID1

as begin

return (select count (TID) from teacher)

end

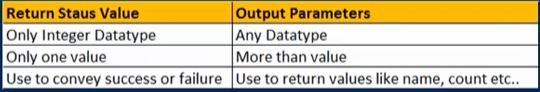
**Execute the program:**

Declare @totalID int

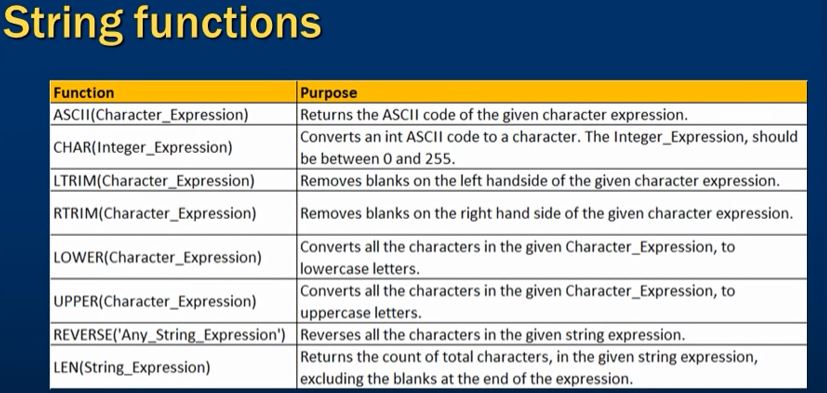
exec @totalID=spGettotalTID1

select @totalID

**Difference between Return value & output parameters**



**Function:**



**ASCII():**

select ascii('A')

**CHAR():**

Declare @a int

set @a=65

while(@a<=90)

Begin

print char( @a)

set @a=@a+1

end

-------------

Declare @a int

set @a=1

while(@a<5)

Begin

select Tid from Teacher

print @a

set @a=@a+1

end

**LTRIM():**

select ' hello '

select LTRIM (' hello ')

select LTRIM(FirstName )from Teacher

select LTRIM(firstname ),LastName ,LTRIM(firstName)+' '+ lastname as fullName from

Teacher

select RTRIM(FirstName )from Teacher

select LTRIM(firstname ),LastName, RTRIM( LTRIM(firstName ) ) +' '+ lastname

as fullName from Teacher

**UPPER() & LOWER():**

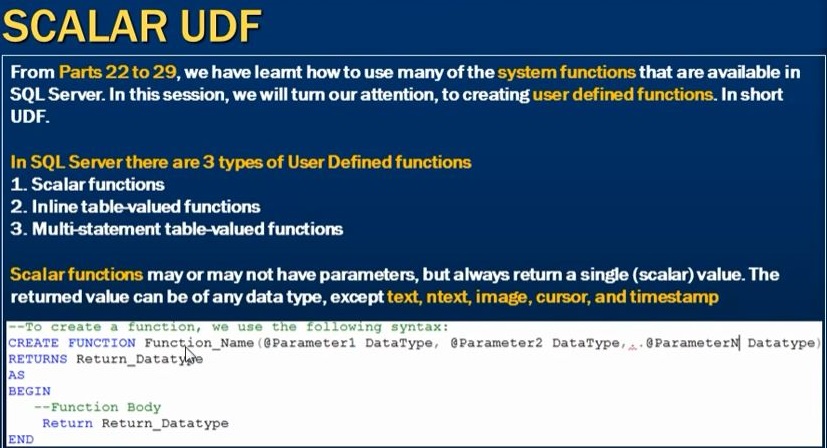
select UPPER(firstname ),LOWER(LastName) from Teacher

**REVERSE():**

select REVERSE(LastName) as ReverseLastName,lastname from Teacher

**LEN():**

select LEN(LastName) as ReverseLastName,lastname from Teacher

****

**Scalar Function:**

create Function CalculateSalary(@Salary float)

returns float

as

begin

declare @maxSalary float

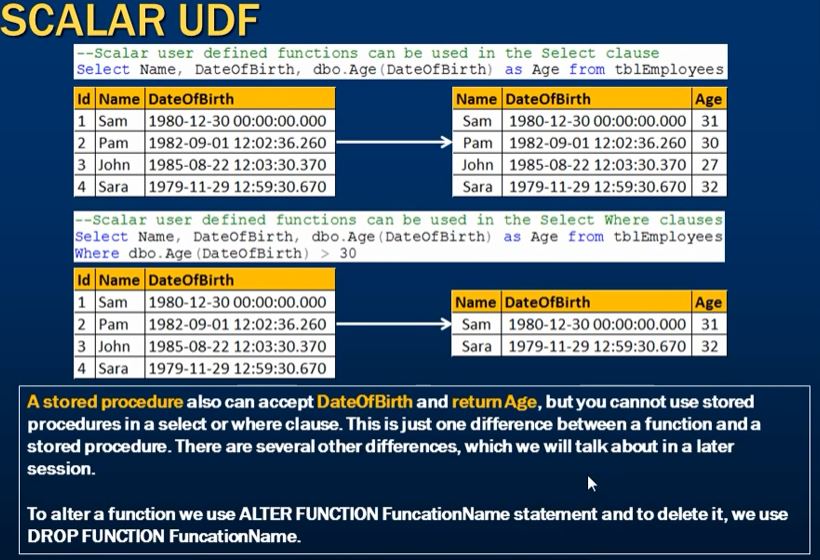
set @maxSalary = max(@Salary)\*100

return @maxSalary

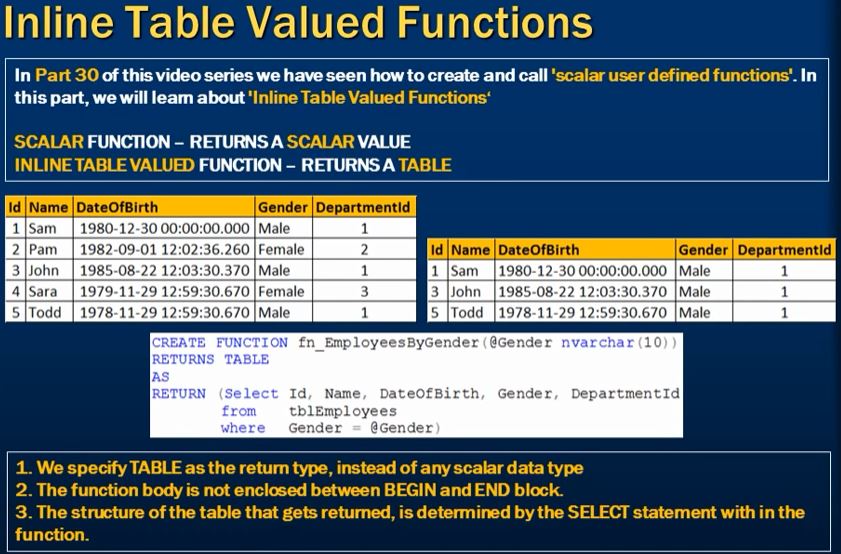
end

**Call that function:**

select dbo.CalculateSalary (30000)



**Inline Function:**

****

**Create Inline Function:**

CREATE FUNCTION dept1(@dept varchar(50))

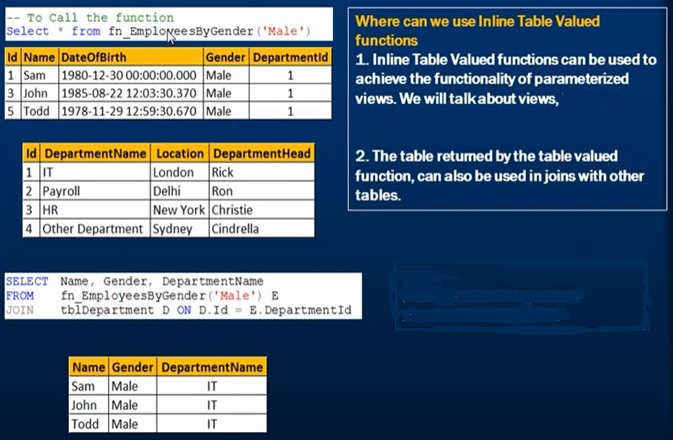
returns Table

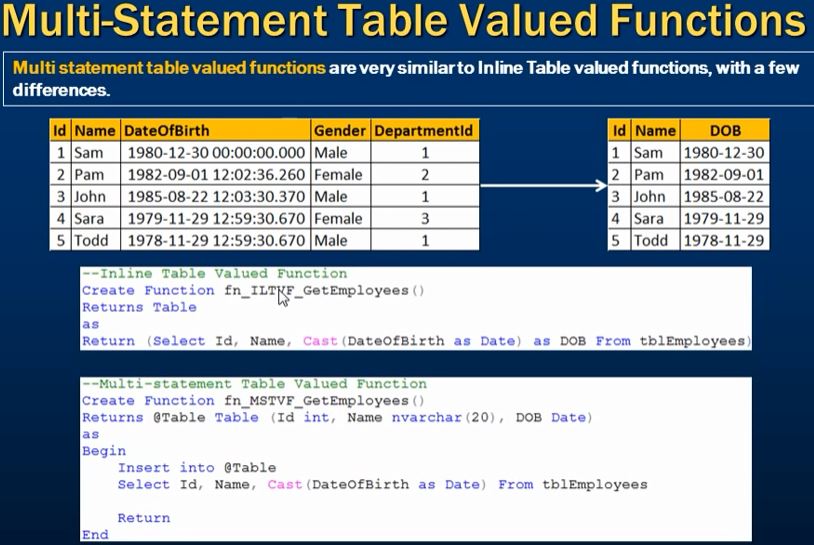
as

return (select Tid,FirstName,Dept from Teacher where Dept = @dept)

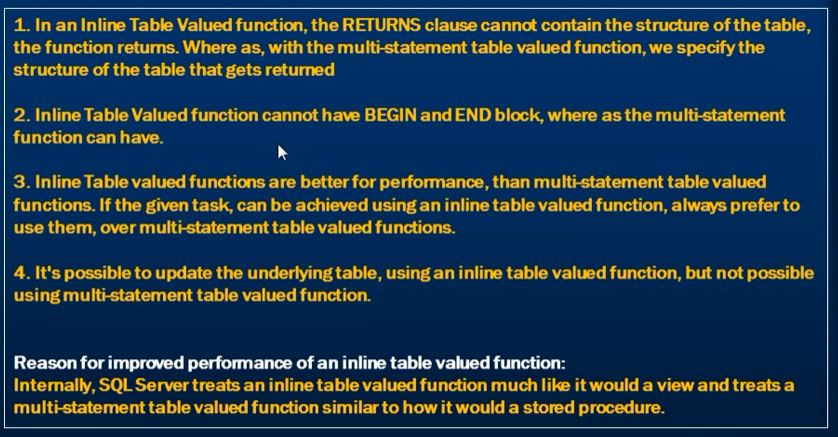
**Call that function:**

Select \* from dept1 ('CSE')





**Difference between Inline Function & Multi Value Function**

****

**Type Calsting:**

**Cast ( ):**

select cast(Salary as varchar) from Teacher

**Recursion in SQL:**

with rec\_prereq(Id,Name)

as

( select Id, Name from tbEmployee  
 union all

select tbEmployee.Id, tbEmployee.Name from tbEmployee join rec\_prereq  
 on rec\_prereq.Id = tbEmployee.Id  
)

select \* from rec\_prereq;

**The Power of Recursion:**

* Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  + Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      * See procedure *findAllPrereqs* in book
* Computing transitive closure using iteration, adding successive tuples to *rec\_prereq*
  + The next slide shows a *prereq* relation
  + Each step of the iterative process constructs an extended version of *rec\_prereq* from its recursive definition.
  + The final result is called the *fixed point* of the recursive view definition.
* Recursive views are required to be **monotonic***.* That is, if we add tuples to *prereq* the view *rec\_prereq* contains all of the tuples it contained before, plus possibly more

**Triggers:**

* A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.

Or

* **Trigger** is a series of PL/SQL statements attached to a database table that execute whenever a triggering event (select, update, insert, delete) occurs.

**Or**

* A database trigger is a PL/SQL block that can defined to automatically execute for insert, update, and delete statements against a table.
  + The trigger can be defined to execute once for the entire statement or once for every row that is inserted, updated, or deleted. For any one table, there are twelve events for which you can define database triggers.
  + A database trigger can call database procedures that are also written in PL/SQL.
* **To design a trigger mechanism, we must**:
  + Specify the conditions under which the trigger is to be executed.
  + Specify the actions to be taken when the trigger executes.
* Unlike stored procedures and functions, **they not explicitly called**, but they are activated when a triggering event occurs.
* **Main purpose is** to implement the complex integrity constraints that can’t be done with the CREATE TABLE or ALTER TABLE command.
* Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  + Syntax illustrated here may not work exactly on your database system; check the system manuals

**Trigger Types:**

* **Application Trigger**
  + Trigger will be activated if there is an event on certain application
* **Database Trigger**
  + Trigger will be activated if there is a data event (DML Operation – Insert, Update, Delete) or system event (logon or shutdown) on a schema or database

**In Sql Server there are 3 types of Triggers**

1. DML Triggers
2. DDL Triggers
3. Logon Triggers

**DML Triggers:** DML triggers are fired automatically in response to DML events

(INSERT,UPDATE & DELETE)

**DML triggers can be again classified into 2 types**

1. After triggers(some time called FOR triggers)
2. Instead of triggers

**After Triggers:** After triggers fires after the triggering action. The INSERT,UPDATE and DELETE statements, causes an after trigger to ire after the respective statements completed.

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

**Trigger Timing:**

* **BEFORE**  
  where a trigger will be activated before DML process on table occur
* **AFTER**  
  where a trigger will be activated after DML process on table occur
* **INSTEAD OF**  
  Trigger that just function ate on VIEW and usually used to update data on complex view

**Syntax:**

CREATE TRIGGER triggers\_name

ON table\_name

FOR INSERT/DELETE

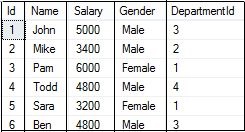
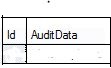
AS

BEGIN

…..

END

**Trigger Example of DML triggers:**

**After Triggers:**

CREATE TRIGGER tr\_Employee\_For\_Insert

ON tbEmployee

FOR INSERT

AS

BEGIN

DECLARE @id int

SELECT @id=Id from inserted

INSERT INTO tbEmployeeAuditData

VALUES ('New employee with Id =' + Cast(@Id as varchar(5))+

'is added at '+ cast(Getdate() as varchar(20)) )

END

In this Query inserted table is special table use by triggers ,is table is only available in trigger

* **insert the value in** tbEmployee **table**

insert into tbEmployee values(8,'Meraj',40000,'Male',3)

**Again create the delete instruction table**

CREATE TRIGGER tr\_Employee\_For\_Delete

ON tbEmployee

FOR DELETE

AS

BEGIN

DECLARE @id int

SELECT @id=Id from deleted

INSERT INTO tbEmployeeAuditData

VALUES ('New employee with Id =' + Cast(@Id as nvarchar(5))+

'is deleted at '+ cast(Getdate() as nvarchar(20)) )

END

In this Query deleted table is special table use by triggers ,is table is only available in trigger. that back up the memory.

* Delete **the value in** tbEmployee **table**

delete from tbEmployee where Id=2

The output show in tbEmployeeAuditData table is bellow:

C:\Users\Meraj Ali\Desktop\Capture.JPG

**Trigger Example of Instead of Insert triggers:**

* **Create the view**

Create view vwEmployeeDetails

as

Select Id,name,Gender, DeptName from tbEmployee

join Department on

Department.DeptId=tbEmployee.DepartmentId

And in this insert the value:

insert into vwEmployeeDetails values (7,'Ali','Male','IT')

The value is not inserted because the modification affects multiple base tables. In this problem we overcome using by triggers, following rules is below.

Create trigger tr\_vwEmployeeDetails\_InsteadOfInsert

on vwEmployeeDetails

Instead of insert

as

begin

select \* from inserted

select \* from deleted

End

Run this code then insert the value in view. No any problem

Insert into vwEmployeeDetails values (7,'Ali','Male','IT')

Again add some query

Alter trigger tr\_vwEmployeeDetails\_InsteadOfInsert

On vwEmployeeDetails

Instead of insert

as

begin

Declare @DeptId int

-- Check if there is a valid DepartmentId

-- for the given Depatment Name

Select @DeptId = DeptId from Department join inserted

On inserted.DeptName=Department.DeptName

-- If DepartmentId is null throw an error

if(@DeptId is null)

Begin

Raiserror ('Invalid Department Name. Statement terminated', 16 ,1)

Return

End

Insert into tbEmployee (Id,Name,Gender ,DepartmentId)

select Id,name,gender ,@DeptId from inserted

End

* Raiserror ('Invalid Department Name. Statement terminated', seriates level , State)

Several seriates level, but 16 means user can correct query

Then insert the value that is not have in department table, in this situation show an error

insert into vwEmployeeDetails values (8,'Ali','Male','ITsss')

Then insert the value that is have in department table, row will be inserted correctly

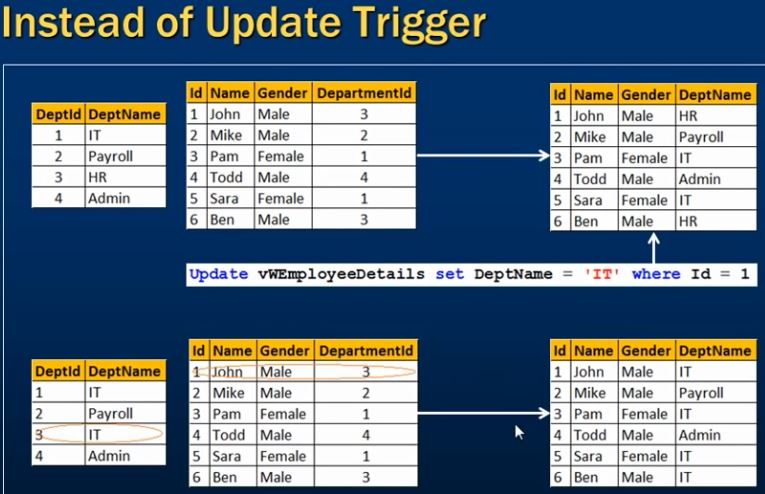
insert into vwEmployeeDetails values (8,'Ali','Male','IT')

**Trigger Example of Instead of Update triggers:**

In this update only view is update not update the main table

Update vwEmployeeDetails set DeptName='IT' where Id=1

In this update only view is update not update the main table, so



Update vwEmployeeDetails set DeptName='HR' where id=1

alter trigger tr\_EmployeeDetails\_InsteadOfUpdate1122

on vwEmployeeDetails

Instead of Update

As Begin

if(Update(Gender))

begin

Update tbEmployee set Gender=inserted.Gender

from inserted join tbEmployee on tbEmployee.id=inserted.id

End

-- if name is update

if(Update(name))

Begin

Update tbEmployee set Name = inserted.name

from inserted join tbEmployee on tbEmployee.id=inserted.id

End

End

**Again change this query add some validation:**

CREATE trigger tr\_EmployeeDetails\_InsteadOfUpdate11

on vwEmployeeDetails

Instead of Update

as

Begin

-- if EnployeeId is update

if(Update(Id))

Begin

Raiserror('Id cannot be changed', 16 ,1)

return

End

-- if DeptName is update

if(Update(DeptName))

Begin

Declare @DeptId int

-- Check if there is a valid DepartmentId

-- for the given Depatment Name

select @DeptId = DeptId from Department join inserted

on inserted.DeptName=Department.DeptName

-- If DepartmentId is null throw an error

if(@DeptId is null)

begin

Raiserror ('Invalid Department Name. Statement terminated', 16 ,1)

return

End

Update tbEmployee set DepartmentId =@DeptId

from inserted join tbEmployee on tbEmployee.id=inserted.id

end

-- if Gender is update

if(Update(Gender))

begin

Update tbEmployee set Gender=inserted.Gender

from inserted join tbEmployee on tbEmployee.id=inserted.id

End

-- if name is update

if(Update(Name))

Begin

Update tbEmployee set Name = inserted.Name

from inserted join tbEmployee on tbEmployee.id=inserted.id

End

End

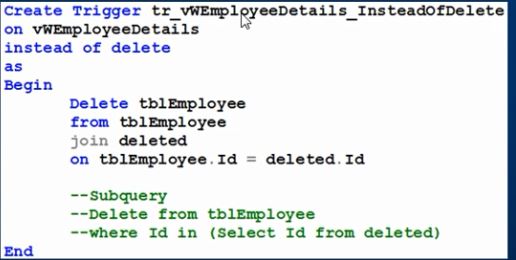
**Now update the view that work in both table**

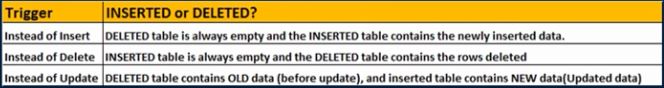
Update vwEmployeeDetails set DeptName='IT' where Id=1

Or

Update vwEmployeeDetails set Name='johny',Gender='Female', DeptName='IT' where Id=1

**Trigger Example of Instead of Delete triggers:**







Create Trigger tr\_vwEmployee\_InstedOfDelete

on vwEmployeeDetails

instead of delete

as

Begin

Delete tbEmployee from tbEmployee join deleted on tbEmployee.Id=deleted.id

End

**Again it down by subQuery:**

Create Trigger tr\_vwEmployee\_InstedOfDelete

on vwEmployeeDetails

instead of delete

as Begin

-- SubQuery

Delete from tbEmployee Where Id in (Select Id from deleted)

End

Now run that query and check it work or not

Delete from vwEmployeeDetails where id=3

**Statement Level Triggers:**

* Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  + Use **for each statement** instead of **for each row**
  + Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  + Can be more efficient when dealing with SQL statements that update a large number of rows

**When Not To Use Triggers:**

* **Triggers were used earlier for tasks such as** 
  + maintaining summary data (e.g., total salary of each department)
  + Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
* **There are better ways of doing these now:**
  + Databases today provide built in materialized view facilities to maintain summary data
  + Databases provide built-in support for replication
* **Encapsulation facilities can be used instead of triggers in many cases**
  + Define methods to update fields
  + Carry out actions as part of the update methods instead of through a trigger
* **Risk of unintended execution of triggers, for example, when**
  + loading data from a backup copy
  + replicating updates at a remote site
  + Trigger execution can be disabled before such actions.
* **Other risks with triggers:**
  + Error leading to failure of critical transactions that set off the trigger
  + Cascading execution

**Managing Trigger:**

* **Enable Trigger**

ALTER TRIGGER trigger\_name ENABLE

* **Disable Trigger**

ALTER TRIGGER trigger\_name DISABLE

* **Enable or Disable All Trigger**

ALTER TABLE table\_name DISABLE | ENABLE ALL TRIGGERS

* **Delete TRIGGER**

DROP TRIGGER nama\_trigger

**Permissions:**

* Just like with procedures and functions, creating triggers requires certain privileges which are not part of the default privilege set.

**GRANT CREATE TRIGGER TO <username>;**

select \* from sys.triggers

IF EXISTS (SELECT \* FROM sys.triggers

WHERE parent\_class = 0 AND name = 'safety')

DROP TRIGGER safety ON DATABASE;

GO

CREATE TRIGGER safety

ON DATABASE

FOR DROP\_TABLE, ALTER\_TABLE

AS

PRINT 'You must disable Trigger "safety" to drop or alter tables!'

ROLLBACK;

GO

DISABLE TRIGGER safety ON DATABASE;

GO

ENABLE TRIGGER safety ON DATABASE;

GO