**DATABASE SCOPED TRIGGER:** (A DDL TRIGGER THAT IS ONLY VISIBLE ONLY FROM A PARTICULAR DATABASE IS CALLED DATABASE SCOPED TRIGGER)

[<DATABASE NAME>🡪PROGRAMMABLITY🡪 DATABASE TRIGGERS]

**CREATE TRGGER FOR TABLE🡪**

create trigger my\_first\_ddl\_triggers

on database

for CREATE\_TABLE

as

begin

print 'youve successfully created this [table]'

end

create table ddl\_test\_table1(id int) --youve successfully created this [table]

**TRIGGER FOR CREATE, ALTER, DELETE🡪**

create trigger second\_ddl\_for\_alter

on database

for CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE

AS

BEGIN

PRINT 'YOURE A BITCH'

END

create table ddl\_test\_table2(id int)

--youve successfully created this [table]

--YOURE A BITCH

**TRIGGER FOR BLOCKING🡪**

CREATE TRIGGER second\_ddl\_for\_BLOCKING --DELETE TRIGGER TO GET ACCESS

ON DATABASE

FOR CREATE\_TABLE

AS

BEGIN

ROLLBACK

PRINT 'DAMN IDIOT'

END

create table ddl\_test\_table3(id int)

O/P:

*youve successfully created this [table]*

*YOURE A BITCH.*

*DAMN IDIOT*

*Msg 3609, Level 16, State 2, Line 1*

*The transaction ended in the trigger. The batch has been aborted.*

**ENABLING & DISABLING TRIGGER🡪**

DISABLE TRIGGER my\_first\_ddl\_triggers ON DATABASE

ENABLE TRIGGER my\_first\_ddl\_triggers ON DATABASE

DROP TRIGGER second\_ddl\_for\_BLOCKING ON DATABASE

**RENAME SOMETHING WITH DDL TRIGGER**

|  |  |
| --- | --- |
| CREATE TRIGGER RENAME\_WORK  ON DATABASE.  FOR RENAME  AS.  BEGIN  PRINT ‘RENAMED SOMETHING’  END | **SP\_RENAME 'ddl\_test\_table1', 'BULLSHIT'**  --Caution: Changing any part of an object name could break scripts and stored procedures.  RENAMED SOMETHING  **SP\_RENAME 'BULLSHIT.ID', 'COW', 'COLUMN'**  --Caution: Changing any part of an object name could break scripts and stored procedures.  RENAMED SOMETHING |

**SERVER SCOPED TRIGGER:** (A TRIGGER THAT CAN BE VISIBLE THROUGHOUT ANY DATABASE UNDER A SERVER IS CALLED SERVER SCOPED TRIGGER)

[SERVER🡪 SERVER OBJECTS🡪 TRIGGERS]

**CREATING A SERVER LEVEL TRIGGER THAT WILL RESIST CREATING TABLE FROM ANY DATABASE UNDER A SERVER🡪**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CREATE TRIGGER BLOCKAGE  ON ALL SERVER  FOR CREATE\_TABLE  AS  BEGIN  PRINT 'BLOCKED'  END | DROP ddl\_test\_table3  --BLOCKED | | DISABLE TRIGGER my\_first\_ddl\_triggers ON ALL SERVER  ENABLE TRIGGER my\_first\_ddl\_triggers ON ALL SERVERS | |
| **\*\*\*SERVER trggers executes before then DATABASE TRIGGERS\*\*\*** | | | | |
| CREATE TRIGGER BLOCKAGE\_2  ON ALL SERVER  FOR CREATE\_TABLE  AS  BEGIN  PRINT 'SERVER'  END | | CREATE TRIGGER POKPOK  ON DATABASE  FOR CREATE\_TABLE  AS  BEGIN  PRINT 'DATABASE'  END | | CREATE TABLE ERROR(ID, INT)  O/P SHOWS LIKE THIS🡪  SERVER  DATABASE |

**ORDER OF TRIGGERING IN A EXECUTION🡪**

|  |  |  |
| --- | --- | --- |
| create trigger ddl\_wwe  on database  for create\_table, alter\_table, drop\_table  as  begin  print 'trigger 3'  end | create trigger ddl\_ufc  on database  for create\_table, alter\_table, drop\_table  as  begin  print 'trigger 2'  end | create trigger ddl\_football  on database  for create\_table, alter\_table, drop\_table  as  begin  print 'trigger 1'  end |
| EXEC **sp\_settriggerorder**  @triggername = 'ddl\_wwe',  @order = 'last',  @stmttype = 'CREATE\_TABLE',  @namespace = 'DATABASE'  GO  sp\_settriggerorder  @triggername='ddl\_football',  @order='first',  @stmttype='create\_table',  @namespace='database' | **create** table ddl\_testing(id int)  order:  trigger 3  trigger 2  trigger 1  \*\*\*BEFORE USING SP\_SETTRIGGERORDER\*\*\* | create table ddl\_testing\_2(id int)  trigger 1  trigger 2  trigger 3  \*\*\*AFTER USING\*\*\*\*  \*\*\*we didn’t use sp\_settriggerorder for DROP\*\*\*  drop table ddl\_testing  trigger 3  trigger 2  trigger 1 |

**If you have a database-scoped and a server-scoped trigger handling the same event, and if you have set the execution order at both the levels. Here is the execution order of the triggers🡪**

* The server-scope trigger marked First
* Other server-scope triggers
* The server-scope trigger marked Last
* The database-scope trigger marked First
* Other database-scope triggers
* The database-scope trigger marked Last

**TABLE AUDITING USING DDL TRIGGERS🡪**

create trigger take\_val

on all server

for create\_table, alter\_table, drop\_table

as

begin

select EVENTDATA()

end

Create table TableChanges\_audit

(

DatabaseName nvarchar(250),

TableName nvarchar(250),

EventType nvarchar(250),

LoginName nvarchar(250),

SQLCommand nvarchar(2500),

AuditDateTime datetime

)

Go

--click on the XML link🡪

|  |
| --- |
| <EVENT\_INSTANCE>  <EventType>CREATE\_TABLE</EventType>  <PostTime>2017-09-27T11:43:27.710</PostTime>  <SPID>52</SPID>  <ServerName>DESKTOP-88GJBMO\SOHAM</ServerName>  <LoginName>DESKTOP-88GJBMO\HP</LoginName>  <UserName>dbo</UserName>  <DatabaseName>master</DatabaseName>  <SchemaName>dbo</SchemaName>  <ObjectName>TableChanges\_audit</ObjectName>  <ObjectType>TABLE</ObjectType>  <TSQLCommand>  <SetOptions ANSI\_NULLS="ON" ANSI\_NULL\_DEFAULT="ON" ANSI\_PADDING="ON" QUOTED\_IDENTIFIER="ON" ENCRYPTED="FALSE" />  <CommandText>Create table TableChanges\_audit  (  DatabaseName nvarchar(250),  TableName nvarchar(250),  EventType nvarchar(250),  LoginName nvarchar(250),  SQLCommand nvarchar(2500),  AuditDateTime datetime  )  </CommandText>  </TSQLCommand>  </EVENT\_INSTANCE> |

create trigger book\_my\_show

on all server

for create\_table, alter\_table, drop\_table

as

begin

declare @EventData xml

select @EventData=EVENTDATA()

insert into testy.dbo.TableChanges\_audit

values

(

@EventData.value('(/EVENT\_INSTANCE/DatabaseName)[1]', 'varchar(250)'),

@EventData.value('(/EVENT\_INSTANCE/ObjectName)[1]', 'varchar(250)'),

@EventData.value('(/EVENT\_INSTANCE/EventType)[1]', 'nvarchar(250)'),

@EventData.value('(/EVENT\_INSTANCE/LoginName)[1]', 'varchar(250)'),

@EventData.value('(/EVENT\_INSTANCE/TSQLCommand)[1]', 'nvarchar(2500)'),

GetDate()

)

end

--testing

select \* from TableChanges\_audit

|  |
| --- |
| testy idiot\_testing CREATE\_TABLE DESKTOP-88GJBMO\HP create table idiot\_testing (id int) 2017-09-27 12:07:03.020  testy idiot\_testing DROP\_TABLE DESKTOP-88GJBMO\HP drop table idiot\_testing 2017-09-27 12:09:40.493  testy idiot\_testing CREATE\_TABLE DESKTOP-88GJBMO\HP create table idiot\_testing (id int) 2017-09-27 12:10:53.477  testy idiot\_testing ALTER\_TABLE DESKTOP-88GJBMO\HP alter table idiot\_testing  alter column ID nvarchar(MAX) 2017-09-27 12:11:37.287 |

create table idiot\_testing (id int)

drop table idiot\_testing

alter table idiot\_testing

alter column ID nvarchar(MAX)

LOGON TRIGGERS🡪

As the name implies Logon triggers fire in response to a LOGON event. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established.

Logon triggers can be used for🡪

1. Tracking login activity
2. Restricting logins to SQL Server
3. Limiting the number of sessions for a specific login

SELECT \* FROM SYS.dm\_exec\_sessions

SELECT is\_user\_process, original\_login\_name, \*

FROM SYS.dm\_exec\_sessions

ORDER BY login\_time ASC

--52 THIS COONECTION

--51 OBJECT EXPLORER

--BLOCKS MORE THAN 3 CONNECTIONS:

CREATE TRIGGER SQLGOD

ON ALL SERVER

FOR LOGON

AS

BEGIN

DECLARE @LOGNAME NVARCHAR(MAX)

SET @LOGNAME=ORIGINAL\_LOGIN()

IF(

SELECT COUNT(\*)

FROM SYS.dm\_exec\_sessions

WHERE is\_user\_process=1 AND original\_login\_name=@LOGNAME)>3

BEGIN

PRINT 'BLOCKED'+@LOGNAME

ROLLBACK;

END

END

SP\_READERRORLOG –SHOWS THE ERROR LOGS

different cases of SELECT \* INTO statements

|  |  |
| --- | --- |
| tables:  1 Mark Male 50000 1  2 Sara Female 65000 2  3 Mike Male 48000 3  4 Pam Female 70000 1  5 John Male 55000 2 | 1 IT  2 HR  3 Payroll |

The SELECT INTO statement in SQL Server, selects data from one table and inserts it into a new table

**OPERATIONS🡪**

|  |  |
| --- | --- |
| PROBLEMS & QUERIES | SOLVES |
| **1. Copy all rows and columns from an existing table into a new table. This is extremely useful when you want to make a backup copy of the existing table.**  *SELECT \* INTO EmployeesBackup FROM Employees* | 1 Mark Male 50000 1  2 Sara Female 65000 2  3 Mike Male 48000 3  4 Pam Female 70000 1  5 John Male 55000 2 |
| 2. Copy all rows and columns from an existing table into a new table in an external database.  SELECT \* INTO HRDB.dbo.EmployeesBackup2 FROM Employees | 1 Mark Male 50000 1  2 Sara Female 65000 2  3 Mike Male 48000 3  4 Pam Female 70000 1  5 John Male 55000 2 |
| 3. Copy only selected columns into a new table  SELECT Id, Name, Gender INTO EmployeesBackup FROM Employees | 1 Mark Male  2 Sara Female  3 Mike Male  4 Pam Female  5 John Male |
| 4. Copy only selected rows into a new table  SELECT \* INTO EmployeesBackup FROM Employees WHERE DeptId = 1 | 1 Mark Male 50000 1  4 Pam Female 70000 1 |
| 5. Copy columns from 2 or more table into a new table  SELECT \* INTO EmployeesBackup  FROM Employees  INNER JOIN Departments  ON Employees.DeptId = Departments.DepartmentId | 1 Mark Male 50000 1 1 IT  2 Sara Female 65000 2 2 HR  3 Mike Male 48000 3 3 Payroll  4 Pam Female 70000 1 1 IT  5 John Male 55000 2 2 HR |
| 6. WE DONT WANT THE DEPTiD AND DepartmentId repetation  select Employees.\*, Departments.DepartmentId into EmployeesBackup  from Employees  inner join Departments  on Employees.DeptId = Departments.DepartmentId | 1 Mark Male 50000 1 IT  2 Sara Female 65000 2 HR  3 Mike Male 48000 3 Payroll  4 Pam Female 70000 1 IT  5 John Male 55000 2 HR |
| 7. Create a new table whose columns and datatypes match with an existing table.  SELECT \* INTO EmployeesBackup FROM Employees WHERE 1 <> 1 | ONLY SHOWS THE TABLE SAME AS EXISTING, WITHOUT ADDING VALUE… |
| 8. Copy all rows and columns from an existing table into a new table on a different SQL Server instance. For this, create a linked server and use the 4 part naming convention  SELECT \* INTO TargetTable  FROM [SourceServer].[SourceDB].[dbo].[SourceTable] | Please note : You cannot use SELECT INTO statement to select data into an existing table. For this you will have to use INSERT INTO statement. |

**DIFFERENCE BETWEEN WHERE & HAVING CLAUSE🡪**

1. WHERE clause cannot be used with aggregates where as HAVING can. This means WHERE clause is used for filtering individual rows where as HAVING clause is used to filter groups.

|  |  |  |
| --- | --- | --- |
| select Product, sum(SaleAmount) as value  from Sales\_testing\_where\_n\_having  group by Product  having sum(SaleAmount)>1000  **iPhone 1500**  **Laptop 1400** | select Product, sum(SaleAmount) as mal  from Sales\_testing\_where\_n\_having  where sum(SaleAmount)>1000 --syntax error  group by Product | iPhone 500  Laptop 800  iPhone 1000  Speakers 400  Laptop 600 |

1. WHERE comes before GROUP BY. This means WHERE clause filters rows before aggregate calculations are performed. HAVING comes after GROUP BY. This means HAVING clause filters rows after aggregate calculations are performed. So from a performance standpoint, HAVING is slower than WHERE and should be avoided when possible
2. WHERE and HAVING can be used together in a SELECT query. In this case WHERE clause is applied first to filter individual rows. The rows are then grouped and aggregate calculations are performed, and then the HAVING clause filters the groups

Ex:

select Product, sum(SaleAmount) as mal

from Sales\_testing\_where\_n\_having

where Product in ('iPhone', 'Laptop')

group by Product

having sum(SaleAmount)>1000 and sum(SaleAmount)<1450

**TABLE VALUED PARAMETER IN T-SQL**

Table Valued Parameter allows a table (i.e. multiple rows of data) to be passed as a parameter to a stored procedure from T-SQL code or from an application.

**Step 1 :** **Create User-defined Table Type**

CREATE TYPE TAB AS TABLE

(

Id int primary key,

Name nvarchar(50),

Gender nvarchar(10)

)

**Step 2 :** **Use the User-defined Table Type as a parameter in the stored procedure. Table valued parameters must be passed as read-only to stored procedures, functions etc. This means you cannot perform DML operations like INSERT, UPDATE or DELETE on a table-valued parameter in the body of a function, stored procedure etc**

CREATE PROC TAKEVALUE

@come TAB readonly

as

begin

insert into Employees

select \* from @come

end

Step 3 : **Declare a table variable, insert the data and then pass the table variable as a parameter to the stored procedure**

declare @come TAB

insert into @come values(1, 'suck', 'M')

insert into @come values(2, 'suck', 'M')

insert into @come values(3, 'suck', 'M')

insert into @come values(4, 'suck', 'M')

insert into @come values(5, 'suck', 'M')

execute TAKEVALUE @come

**GROUPING SETS IN T-SQL🡪**

grouping sets can be used as an alternative with “union all” operator. A large number of union all can be simplified by using group set parameters.

EX🡪

|  |  |  |
| --- | --- | --- |
| TAKING TABLE | | 1 Mark Male 5000 USA  2 John Male 4500 India  3 Pam Female 5500 USA  4 Sara Female 4000 India  5 Todd Male 3500 India  6 Mary Female 5000 UK  7 Ben Male 6500 UK  8 Elli Female 7000 USA  9 Tom Male 5500 UK  10 Ron Male 5000 USA |
| We want to calculate **Sum of Salary by Country and Gender🡪**  select Country, Gender, SUM(Salary) as total  from [dbo].[Employees\_for\_grooping\_set\_for\_grooping\_set\_2]  group by Country, Gender  union all  select Country, NULL, SUM(Salary) as total  from [dbo].[Employees\_for\_grooping\_set\_for\_grooping\_set\_2]  group by Country  union all  select null, Gender, SUM(Salary) as total  from [dbo].[Employees\_for\_grooping\_set\_for\_grooping\_set\_2]  group by Gender  union all  select null, null, SUM(Salary) as total  from [dbo].[Employees\_for\_grooping\_set\_for\_grooping\_set\_2] | India Female 4000  UK Female 5000  USA Female 12500  India Male 8000  UK Male 12000  USA Male 10000 |  |
| India NULL 12000  UK NULL 17000  USA NULL 22500 | India Female 4000  UK Female 5000  USA Female 12500  India Male 8000  UK Male 12000  USA Male 10000  India NULL 12000  UK NULL 17000  USA NULL 22500 |
| NULL Female 21500  NULL Male 30000 | India Female 4000  UK Female 5000  USA Female 12500  India Male 8000  UK Male 12000  USA Male 10000  India NULL 12000  UK NULL 17000  USA NULL 22500  NULL Female 21500  NULL Male 30000 |
| NULL NULL 51500 | India Female 4000  UK Female 5000  USA Female 12500  India Male 8000  UK Male 12000  USA Male 10000  **India NULL 12000**  **UK NULL 17000**  **USA NULL 22500**  *NULL Female 21500*  *NULL Male 30000*  NULL NULL 51500 |
| select Country, Gender, SUM(Salary) as total  from [dbo].[Employees\_for\_grooping\_set\_for\_grooping\_set\_2]  group by  GROUPING SETS  (  (Country, Gender),  (Country),  (Gender),  ()  )  order by GROUPING(Country), GROUPING(Gender) | India Female 4000  UK Female 5000  USA Female 12500  India Male 8000  UK Male 12000  USA Male 10000  **India NULL 12000**  **UK NULL 17000**  **USA NULL 22500**  *NULL Female 21500*  *NULL Male 30000*  NULL NULL 51500 | |

**ROLLUP IN TSQL🡪**

|  |  |  |  |
| --- | --- | --- | --- |
| **ROLL UP** | **UNION ALL** | | **GROUPING SETS** |
| SELECT Country, sum(Salary)  FROM Employees\_for\_grooping\_set\_for\_grooping\_set\_2  GROUP BY ROLLUP(Country) | select Country, SUM(Salary)  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2  group by Country  union all  select null, SUM(Salary)  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2 | | select Country, SUM(Salary)  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2  group by grouping sets  (  (Country),  ()  ) |
| SELECT Country,Gender, sum(Salary)  FROM Employees\_for\_grooping\_set\_for\_grooping\_set\_2  GROUP BY ROLLUP(Country, Gender) | select Country, Gender, SUM(Salary)  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2  group by Country, Gender  union all  select Country, null, SUM(Salary)  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2  group by Country  union all  select null, null, SUM(Salary)  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2  order by Country | | select Country,Gender, SUM(Salary)  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2  group by grouping sets  (  (Country, Gender),  (Country),  ()  ) |
| Case 1 | | Case 2 | |
| India 12000  UK 17000  USA 22500  NULL 51500 | | India Female 4000  India Male 8000  India NULL 12000  UK Female 5000  UK Male 12000  UK NULL 17000  USA Female 12500  USA Male 10000  USA NULL 22500  NULL NULL 51500 | |

**Write a query to retrieve Sum of Salary grouped by all combinations of the following 2 columns as well as Grand Total. Country, Gender**

|  |  |  |
| --- | --- | --- |
| select Country, Gender, SUM(Salary) as shit  from Employees\_for\_grooping\_set\_for\_grooping\_set\_2  group by CUBE(Country, Gender) | 1 Mark Male 5000 USA  2 John Male 4500 India  3 Pam Female 5500 USA  4 Sara Female 4000 India  5 Todd Male 3500 India  6 Mary Female 5000 UK  7 Ben Male 6500 UK  8 Elli Female 7000 USA  9 Tom Male 5500 UK  10 Ron Male 5000 USA | India Female 4000  UK Female 5000  USA Female 12500  NULL Female 21500  India Male 8000  UK Male 12000  USA Male 10000  NULL Male 30000  NULL NULL 51500  India NULL 12000  UK NULL 17000  USA NULL 22500 |

**DIFFERENCE BETWEEN ROLLUP VS CUBE?**

CUBE generates a result set that shows aggregates for all combinations of values in the selected columns, whereas ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

|  |  |  |
| --- | --- | --- |
|  | select Country, City, SUM(SaleAmount)  from [dbo].[rollup\_vs\_cube]  group by cube(Country, City) | select Country, City, SUM(SaleAmount)  from [dbo].[rollup\_vs\_cube]  group by rollup(Country, City) |
| **1 Asia India Bangalor 1000**  **2 Asia India Chennai 2000**  **3 Asia Japan Tokyo 4000** | **India Bangalore 1000**  **NULL Bangalore 1000**  **India Chennai 2000**  **NULL Chennai 2000**  **Japan Tokyo 4000**  **NULL Tokyo 4000**  **NULL NULL 7000**  **India NULL 3000**  **Japan NULL 4000** | **India Bangalor 1000**  **India Chennai 2000**  **India NULL 3000**  **Japan Tokyo 4000**  **Japan NULL 4000**  **NULL NULL 7000** |
| \*\*\*Putting one ATTRIBUTE in ROLLUP or CUBE function make any identical result\*\*\* | | |

**USING GROUPING FUNCTIONS🡪**

Grouping(Column) indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set

|  |  |
| --- | --- |
| select Continent, Country, City, SUM(SaleAmount),  grouping(Continent) as continent\_id,  grouping(Country) as Country\_id,  grouping(City) as City\_id  from rollup\_vs\_cube  group by rollup(Continent, Country, City) | select case when  grouping(Continent)=1 then 'ALL' else ISNULL(Continent, 'unknown') end as Continent,  case when  grouping(Country)=1 then 'ALL' Else isnull(Country, 'UNKNOWN') end as City,  case when  grouping(City)=1 then 'ALL' else isnull(City, 'unknown') end as city,  sum(SaleAmount)  from rollup\_vs\_cube  group by rollup(Continent, Country, City) |
| Asia India Bangal 1000 0 0 0  Asia India Chenna 2000 0 0 0  Asia India NULL 3000 0 0 1  Asia Japan Tokyo 4000 0 0 0  Asia Japan NULL 4000 0 0 1  Asia NULL NULL 7000 0 1 1  NULL NULL NULL 7000 1 1 1 | Asia India Bangalor 1000  Asia India Chennai 2000  Asia India ALL 3000  Asia Japan Tokyo 4000  Asia Japan ALL 4000  Asia ALL ALL 7000  ALL ALL ALL 7000 |

**Why we shouldn’t use ISNULL function instead of GROUPING?**

|  |  |  |  |
| --- | --- | --- | --- |
| select  **ISNULL(Continent, 'all'),**  **isnull(City, 'All'),**  **ISNULL(Country, 'All'),**  sum(SaleAmount)  from rollup\_vs\_cube  group by rollup(Continent, Country, City) | Asia India **Bangal** 1000  Asia India Chennai 2000  Asia India All 3000  Asia Japan Tokyo 4000  Asia Japan All 4000  Asia All All 7000  all All All 7000 | 1 Asia India NULL 1000  2 Asia India Chennai 2000  3 Asia Japan Tokyo 4000  ***\*\*\*WHEN WE GET A COL WITH null VALUE, EXECUTING ISNULL, WE GET🡪*** | Asia India **All** 1000  Asia India Chennai 2000  Asia India All 3000  Asia Japan Tokyo 4000  Asia Japan All 4000  Asia All All 7000  all All All 7000 |
| select case when  grouping(Continent)=1 then 'ALL' else ISNULL(Continent, 'unknown') end as Continent,  case when  grouping(Country)=1 then 'ALL' Else isnull(Country, 'UNKNOWN') end as City,  case when  grouping(City)=1 then 'ALL' else isnull(City, 'unknown') end as city,  sum(SaleAmount)  from rollup\_vs\_cube  group by rollup(Continent, Country, City) | | Asia India **unknown** 1000  Asia India Chennai 2000  Asia India ALL 3000  Asia Japan Tokyo 4000  Asia Japan ALL 4000  Asia ALL ALL 7000  ALL ALL ALL 7000 |
| **\*\*\*CUBE FUNCTION IS ALSO AVAILABLE IN “GROUPING”\*\*\*\*** | | | |

**GROUPING\_ID IN T-SQL🡪**

GROUPING indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set.   
GROUPING\_ID() function concatenates all the GOUPING() functions, perform the binary to decimal conversion, and returns the equivalent integer. In short  
GROUPING\_ID(A, B, C) =  GROUPING(A) + GROUPING(B) + GROUPING(C)

|  |  |
| --- | --- |
| select Continent, Country, City, SUM(SaleAmount),  CAST(grouping(Continent) AS nvarchar(1))+  CAST(grouping(Country)AS nvarchar(1)) +  CAST(grouping(City)AS nvarchar(1)) as BIN,  GROUPING\_ID(Continent, Country, City) AS GPID  from rollup\_vs\_cube  group by rollup(Continent, Country, City) | Asia India NULL 1000 000 0  Asia India Chennai 2000 000 0  Asia India NULL 3000 001 1  Asia Japan Tokyo 4000 000 0  Asia Japan NULL 4000 001 1  Asia NULL NULL 7000 011 3  NULL NULL NULL 7000 111 7 |
| select Continent, Country, City, SUM(SaleAmount),  CAST(grouping(Continent) AS nvarchar(1))+  CAST(grouping(Country)AS nvarchar(1)) +  CAST(grouping(City)AS nvarchar(1)) as BIN,  GROUPING\_ID(Continent, Country, City) AS GPID  from rollup\_vs\_cube  group by rollup(Continent, Country, City)  ORDER BY GPID | Asia Japan Tokyo 4000 000 0  Asia India NULL 1000 000 0  Asia India Chennai 2000 000 0  Asia India NULL 3000 001 1  Asia Japan NULL 4000 001 1  Asia NULL NULL 7000 011 3  NULL NULL NULL 7000 111 7 |
| select Continent, Country, City, SUM(SaleAmount),  GROUPING\_ID(Continent, Country, City) AS GPID  from rollup\_vs\_cube  group by rollup(Continent, Country, City)  HAVING GROUPING\_ID(Continent, Country, City)>3 | NULL NULL NULL 7000 111 7 |
| the number of attributes in GROUP BY(A, B, C) should be same after GROPING\_ID(A, B, C) column🡪  select Continent, Country, City, SUM(SaleAmount),  CAST(grouping(Continent) AS nvarchar(1))+  CAST(grouping(Country)AS nvarchar(1)) +  CAST(grouping(City)AS nvarchar(1)) as City\_id,  GROUPING\_ID( **Country, City**) AS GPID  from rollup\_vs\_cube  group by rollup(**Continent, Country, City**) | Asia India NULL 1000 000 0  Asia India Chennai 2000 000 0  Asia India NULL 3000 001 1  Asia Japan Tokyo 4000 000 0  Asia Japan NULL 4000 001 1  Asia NULL NULL 7000 011 3  NULL NULL NULL 7000 111 3 |

**OVER(PARTITION BY) CLAUSE IN T-SQL?**

Determines the partitioning and ordering of a rowset before the associated window function is applied. That is, the OVER clause defines a window or user-specified set of rows within a query result set. A window function then computes a value for each row in the window. You can use the OVER clause with functions to compute aggregated values such as moving averages, cumulative aggregates, running totals, or a top N per group results.

|  |  |  |
| --- | --- | --- |
| 1 Mark Male 5000  2 John Male 4500  3 Pam Female 5500  4 Sara Female 4000  5 Todd Male 3500  6 Mary Female 5000  7 Ben Male 6500  8 Jodi Female 7000  9 Tom Male 5500  10 Ron Male 5000 | Female 4 5375 7000 4000  Male 6 5000 6500 3500 | Pam Female 5500 4 5375 7000 4000  Sara Female 4000 4 5375 7000 4000  Mary Female 5000 4 5375 7000 4000  Jodi Female 7000 4 5375 7000 4000  Tom Male 5500 6 5000 6500 3500  Ron Male 5000 6 5000 6500 3500  Ben Male 6500 6 5000 6500 3500  Todd Male 3500 6 5000 6500 3500  Mark Male 5000 6 5000 6500 3500  John Male 4500 6 5000 6500 3500 |
|  | select distinct gender, COUNT(Gender), AVG(Salary), MAX(Salary), MIN(Salary)  from Employees\_over\_testing  group by Gender | select Name, Gender, Salary,  count(Gender) over(partition by Gender),  avg(Salary) over(partition by Gender),  max(Salary) over(partition by Gender),  min(Salary) over(partition by Gender)  from Employees\_over\_testing |

**Limitations of OVER clause:**

The OVER clause cannot be used with the CHECKSUM aggregate function.

**ROW\_NUMBER function in T-SQL?**

Numbers the output of a result set. More specifically, returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition

Row\_Number function🡪

* Introduced in SQL Server 2005
* Returns the sequential number of a row starting at 1
* ORDER BY clause is required
* PARTITION BY clause is optional
* When the data is partitioned, row number is reset to 1 when the partition changes

**Syntax🡪**  **ROW\_NUMBER () OVER (ORDER BY Col1, Col2)**

|  |  |
| --- | --- |
| Select Name, Gender, Salary,  Row\_number() over(order by Gender) as Number  from Employees\_over\_testing | select Name, Gender, Salary, ROW\_NUMBER() over(partition by Gender order by Gender) as Number  from Employees\_over\_testing |
| Pam Female 5500 1  Sara Female 4000 2  Mary Female 5000 3  Jodi Female 7000 4  Tom Male 5500 5  Ron Male 5000 6  Ben Male 6500 7  Todd Male 3500 8  Mark Male 5000 9  John Male 4500 10 | Pam Female 5500 1  Sara Female 4000 2  Mary Female 5000 3  Jodi Female 7000 4  Tom Male 5500 1  Ron Male 5000 2  Ben Male 6500 3  Todd Male 3500 4  Mark Male 5000 5  John Male 4500 6 |

**USE in real life🡪** Deletes the duplicate rows…

with ctetab

as

(

select \*, ROW\_NUMBER() over(partition by FirstName order by Gender) as Valk from Employees\_duplicate

)

delete from ctetab where Valk>1

|  |  |
| --- | --- |
| 1 Mark Hastings Male 60000  1 Mark Hastings Male 60000  1 Mark Hastings Male 60000  2 Mary Lambeth Female 30000  2 Mary Lambeth Female 30000  3 Ben Hoskins Male 70000  3 Ben Hoskins Male 70000  3 Ben Hoskins Male 70000 | 1 Mark Hastings Male 60000  2 Mary Lambeth Female 30000  3 Ben Hoskins Male 70000 |

**RANK() and DENSE\_RANK() functions in T\_SQL?**

|  |  |  |  |
| --- | --- | --- | --- |
| **RANK()** | | **DENSE\_RANK()** | |
| * Returns the rank of rows within the partition of a result set, with gaps in the ranking. * RANK is nondeterministic. * RETURN type: bigint | | * Returns the rank of rows within the partition of a result set, without any gaps in the ranking. * RANK is deterministic. * RETURN type: bigint | |
| 1 Mark Male 8000  2 John Male 8000  3 Pam Female 5000  4 Sara Female 4000  5 Todd Male 3500  6 Mary Female 6000  7 Ben Male 6500  8 Jodi Female 4500  9 Tom Male 7000  10 Ron Male 6800 | 1 Mark Male 8000 1  2 John Male 8000 1  9 Tom Male 700 3  10 Ron Male 6800 4  7 Ben Male 6500 5  6 Mary Female 6000 6  3 Pam Female 5000 7  8 Jodi Female 4500 8  4 Sara Female 4000 9  5 Todd Male 3500 10 | | 1 Mark Male 8000 1  2 John Male 8000 1  9 Tom Male 7000 2  10 Ron Male 6800 3  7 Ben Male 6500 4  6 Mary Female 6000 5  3 Pam Female 5000 6  8 Jodi Female 4500 7  4 Sara Female 4000 8  5 Todd Male 3500 9 |
|  | select \*, RANK() over(order by Salary desc) as Rank\_test  from Employees\_testing\_rank | | select \*, dense\_RANK() over(order by Salary Desc) as Rank\_test  from Employees\_testing\_rank |

select \*, dense\_RANK() over(partition by Gender order by Salary Desc) as Rank\_test

from Employees\_testing\_rank

6 Mary Female 6000 1

3 Pam Female 5000 2

8 Jodi Female 4500 3

4 Sara Female 4000 4

1 Mark Male 8000 **1**

2 John Male 8000 **1**

9 Tom Male 7000 2

10 Ron Male 6800 3

7 Ben Male 6500 4

5 Todd Male 3500 5

|  |  |
| --- | --- |
| select top 3 Salary as g  from Employees\_testing\_rank | 8000  8000  5000 |
| with functe as  (  select Name, Gender, Salary,  RANK()over(order by Salary desc) as sal  from Employees\_testing\_rank  )  select top 1 Salary from functe where sal=1 | 1🡪8000  2🡪8000 |
| with functe as  (  select Name, Gender, Salary,  RANK()over(order by Salary desc) as sal  from Employees\_testing\_rank  )  select top 1 Salary from functe where sal=1 | 1🡪 8000  2🡪 7000 |

**Rank() vs Row\_Number() vs Dense\_Rank() in T-SQL?**

|  |  |
| --- | --- |
| select Name, Gender, Salary, **n**  ROW\_NUMBER()over(order by Salary desc) as [row\_number], **o**  RANK() over (order by Salary desc) as [rank], **t** DENSE\_RANK() over(order by Salary desc) as [dense\_ranke] **i**  from Employees\_testing\_rank **e** | Mark Male 8000 1 1 1  John Male 8000 2 1 1  Pam Female 5000 3 3 2  Sara Female 4000 4 4 3  Todd Male 3500 5 5 4 |
| select Name, Gender, Salary,  ROW\_NUMBER()over(order by Salary desc) as [Row\_number], **t**  RANK() over(order by Salary desc) as [rank], **i**  DENSE\_RANK() over(order by Salary desc) as [Dense\_rank] **e**  from Employees\_testing\_rank | Mark Male 8000 1 1 1  John Male 8000 2 1 1  Pam Female 5000 3 3 2  Sara Female 4000 4 4 3  Todd Male 3500 5 5 4 |

**RUNNING SUM function in T-SQL**

|  |  |  |
| --- | --- | --- |
| 1 Mark Male 5000  2 John Male 4500  3 Pam Female 5500  4 Sara Female 4000  5 Todd Male 3500  6 Mary Female 5000  7 Ben Male 6500  8 Jodi Female 7000  9 Tom Male 5500  10 Ron Male 5000 | select Id, Name, Gender, Salary,  sum(Salary) over(order by Id) as Running  from Employees\_testing\_Running\_sum | 1 Mark Male 5000 5000  2 John Male 4500 9500  3 Pam Female 5500 15000  4 Sara Female 4000 19000  5 Todd Male 3500 22500  6 Mary Female 5000 27500  7 Ben Male 6500 34000  8 Jodi Female 7000 41000  9 Tom Male 5500 46500  10 Ron Male 5000 51500 |
|  | select Id, Name, Gender, Salary,  sum(Salary) over(partition by Gender order by Id) as Running  from Employees\_testing\_Running\_sum  select Id, Name, Gender, Salary,  sum(Salary) over(order by **Salary**) as Running  from Employees\_testing\_Running\_sum  ---salary is not a primary key, that is why we gonna get an error where PK violation occurs at SALARY  \*\*\*solution @pg🡪16 | 3 Pam Female 5500 5500  4 Sara Female 4000 9500  6 Mary Female 5000 14500  8 Jodi Female 7000 21500  1 Mark Male 5000 5000  2 John Male 4500 9500  5 Todd Male 3500 13000  7 Ben Male 6500 19500  9 Tom Male 5500 25000  10 Ron Male 5000 30000  5 Todd Male 3500 3500  4 Sara Female 4000 7500  2 John Male 4500 12000  1 Mark Male 5000 27000  6 Mary Female 5000 27000  10 Ron Male 5000 27000  9 Tom Male 5500 38000  3 Pam Female 5500 38000  7 Ben Male 6500 44500  8 Jodi Female 7000 51500 |

**NTILE() function in T-SQL🡪**

* Introduced in SQL Server 2005
* ORDER BY Clause is required
* PARTITION BY clause is optional
* Distributes the rows into a specified number of groups
* If the number of rows is not divisible by number of groups, you may have groups of two different sizes.
* Larger groups come before smaller groups

|  |  |
| --- | --- |
| select Id, Name, Gender, Salary,  NTILE(2)over(order by Salary) as [Ntile]  from Employees\_testing\_Running\_sum | 5 Todd Male 3500 1  4 Sara Female 4000 1  2 John Male 4500 1  1 Mark Male 5000 1  6 Mary Female 5000 1  10 Ron Male 5000 2  9 Tom Male 5500 2  3 Pam Female 5500 2  7 Ben Male 6500 2  8 Jodi Female 7000 2 |
| select Name, Gender, Salary,  NTILE(3)over(order by Salary) as [NTILE]  from Employees\_testing\_Running\_sum | Todd Male 3500 1  Sara Female 4000 1  John Male 4500 1  Mark Male 5000 1  Mary Female 5000 2  Ron Male 5000 2  Tom Male 5500 2  Pam Female 5500 3  Ben Male 6500 3  Jodi Female 7000 3 |
| select Name, Gender, Salary, NTILE(11)over(order by Salary) as [NTILE]  from Employees\_testing\_Running\_sum | Todd Male 3500 1  Sara Female 4000 2  John Male 4500 3  Mark Male 5000 4  Mary Female 5000 5  Ron Male 5000 6  Tom Male 5500 7  Pam Female 5500 8  Ben Male 6500 9  Jodi Female 7000 10 |
| select Name, Gender, Salary,  ROW\_NUMBER() over( order by Gender),  NTILE(2)over(partition by Gender order by Salary) as [ntile]  from Employees\_testing\_Running\_sum | Sara Female 4000 1 1  Mary Female 5000 2 1  Pam Female 5500 3 2  Jodi Female 7000 4 2  Todd Male 3500 5 1  John Male 4500 6 1  Mark Male 5000 7 1  Ron Male 5000 8 2  Tom Male 5500 9 2  Ben Male 6500 10 2 |
| with takecte as  (  select Name, Gender, Salary,  NTILE(3)over(order by Salary) as [ntile]  from Employees\_testing\_Running\_sum  )  select \* from takecte where [ntile]=2 | Mary Female 5000 2  Ron Male 5000 2  Tom Male 5500 2 |

**LEAD() & LAG() functions in T-SQL🡪**

Accesses data from a previous row in the same result set without the use of a self-join in SQL Server 2017. LAG provides access to a row at a given physical offset that comes before the current row. Use this analytic function in a SELECT statement to compare values in the current row with values in a previous row

* Introduced in SQL Server 2012
* Lead function is used to access subsequent row data along with current row data
* Lag function is used to access previous row data along with current row data
* ORDER BY clause is required
* PARTITION BY clause is optiona

**Syntax**

LEAD(Column\_Name, Offset, Default\_Value) OVER (ORDER BY Col1, Col2, ...)

LAG(Column\_Name, Offset, Default\_Value) OVER (ORDER BY Col1, Col2, ...)

**Offset -**Number of rows to lead or lag.

**Default\_Value -** The default value to return if the number of rows to lead or lag goes beyond first row or last row in a table or partition. If default value is not specified NULL is returned

|  |  |
| --- | --- |
| select Id, Name, Gender, Salary,  lead(Salary)over(order by gender) as [Lead],  LAG(Salary)over(order by gender)as [Lag]  from Employees\_testing\_Running\_sum | select Id, Name, Gender, Salary,  lead(Salary, 2, -1)over(order by gender) as [Lead],  LAG(Salary, 2, -1)over(order by gender)as [Lag]  from Employees\_testing\_Running\_sum |
| 3 Pam Female 5500 4000 NULL  4 Sara Female 4000 5000 5500  6 Mary Female 5000 7000 4000  8 Jodi Female 7000 5500 5000  9 Tom Male 5500 5000 7000  10 Ron Male 5000 6500 5500  7 Ben Male 6500 3500 5000  5 Todd Male 3500 5000 6500  1 Mark Male 5000 4500 3500  2 John Male 4500 NULL 5000 | 3 Pam Female 5500 5000 -1  4 Sara Female 4000 7000 -1  6 Mary Female 5000 5500 5500  8 Jodi Female 7000 5000 4000  9 Tom Male 5500 6500 5000  10 Ron Male 5000 3500 7000  7 Ben Male 6500 5000 5500  5 Todd Male 3500 4500 5000  1 Mark Male 5000 -1 6500  2 John Male 4500 -1 3500 |
| select Id, Name, Gender, Salary,  lead(Salary, 2, -1)over( partition by Gender order by gender) as [Lead],  LAG(Salary, 2, -1)over(partition by Gender order by gender)as [Lag]  from Employees\_testing\_Running\_sum 🡺🡺🡺🡺🡺🡺🡺 | 3 Pam Female 5500 5000 -1  4 Sara Female 4000 7000 -1  6 Mary Female 5000 -1 5500  8 Jodi Female 7000 -1 4000  9 Tom Male 5500 6500 -1  10 Ron Male 5000 3500 -1  7 Ben Male 6500 5000 5500  5 Todd Male 3500 4500 5000  1 Mark Male 5000 -1 6500  2 John Male 4500 -1 3500 |

**FIRST\_VALUE() functions in T-SQL🡪**

Returns the first value in an ordered set of values in SQL Server 2017.

* Is the same type as scalar\_expression.
* FIRST\_VALUE is nondeterministic.

**Syntax🡪**

FIRST\_VALUE ( [scalar\_expression ] ) OVER ( [ partition\_by\_clause ] order\_by\_clause [ rows\_range\_clause ] )

|  |  |
| --- | --- |
| select Name, Gender, Salary,  FIRST\_VALUE(Name)over(order by Salary) as fb  from Employees\_testing\_Running\_sum | select Name, Gender, Salary,  FIRST\_VALUE(Name)over(partition by Gender order by Salary) as fb  from Employees\_testing\_Running\_sum |
| Todd Male 3500 Todd  Sara Female 4000 Todd  John Male 4500 Todd  Mark Male 5000 Todd  Mary Female 5000 Todd  Ron Male 5000 Todd  Tom Male 5500 Todd  Pam Female 5500 Todd  Ben Male 6500 Todd  Jodi Female 7000 Todd | Sara Female 4000 Sara  Mary Female 5000 Sara  Pam Female 5500 Sara  Jodi Female 7000 Sara  Todd Male 3500 Todd  John Male 4500 Todd  Mark Male 5000 Todd  Ron Male 5000 Todd  Tom Male 5500 Todd  Ben Male 6500 Todd |

**WINDOW FUNCTIONS IN T-SQL?**

In SQL Server we have different categories of window functions

* **Aggregate functions -** AVG, SUM, COUNT, MIN, MAX etc..
* **Ranking functions -** RANK, DENSE\_RANK, ROW\_NUMBER etc..
* **Analytic functions -** LEAD, LAG, FIRST\_VALUE, LAST\_VALUE etc...

**OVER** Clause defines the partitioning and ordering of a rows (i.e a window) for the above functions to operate on. Hence these functions are called window functions. The OVER clause accepts the following three arguments to define a window for these functions to operate on.

* **ORDER BY :** Defines the logical order of the rows
* **PARTITION BY :** Divides the query result set into partitions. The window function is applied to each partition separately.
* **ROWSor RANGE clause :** Further limits the rows within the partition by specifying start and end points within the partition.

The default for **ROWS**or **RANGE**clause is

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

|  |  |
| --- | --- |
| SELECT Name, Gender, Salary,  AVG(Salary)over(order by Salary) as average,  COUNT(Salary) over(order by Salary) as [count],  SUM(Salary)over(order by Salary) as [sum]  from Employees\_testing\_Running\_sum ----duplication | **Todd Male 3500 3500 1 3500**  **Sara Female 4000 3750 2 7500**  **John Male 4500 4000 3 12000**  **Mark Male 5000 4500 6 27000**  **Mary Female 5000 4500 6 27000**  **Ron Male 5000 4500 6 27000**  **Tom Male 5500 4750 8 38000**  **Pam Female 5500 4750 8 38000**  **Ben Male 6500 4944 9 44500**  **Jodi Female 7000 5150 10 51500** |
| SELECT Name, Gender, Salary,  AVG(Salary)over(order by Salary) as average,  COUNT(Salary) over(order by Id) as [count],  SUM(Salary)over(order by Id) as [sum]  from Employees\_testing\_Running\_sum -----no duplication | **Mark Male 5000 4500 1 5000**  **John Male 4500 4000 2 9500**  **Pam Female 5500 4750 3 15000**  **Sara Female 4000 3750 4 19000**  **Todd Male 3500 3500 5 22500**  **Mary Female 5000 4500 6 27500**  **Ben Male 6500 4944 7 34000**  **Jodi Female 7000 5150 8 41000**  **Tom Male 5500 4750 9 46500**  **Ron Male 5000 4500 10 51500** |
| select Name, Gender, Salary,  AVG(Salary)over(order by Id ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as [avg],  COUNT(Salary) over(order by Id rows between unbounded preceding and unbounded following) as [count],  SUM(Salary)over(order by Id rows between unbounded preceding and unbounded following) as [sum]  from Employees\_testing\_Running\_sum | **Mark Male 5000 5150 10 51500**  **John Male 4500 5150 10 51500**  **Pam Female 5500 5150 10 51500**  **Sara Female 4000 5150 10 51500**  **Todd Male 3500 5150 10 51500**  **Mary Female 5000 5150 10 51500**  **Ben Male 6500 5150 10 51500**  **Jodi Female 7000 5150 10 51500**  **Tom Male 5500 5150 10 51500**  **Ron Male 5000 5150 10 51500** |
| --TO CALCULTE THE VALUE BETWEEN  --FOR ROW2= ROW1+ROW2+ROW3  --FOR ROW3=ROW2+ROW3+ROW4  select Name, Gender, Salary,  AVG(Salary)over(order by Id ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) as [avg],  COUNT(Salary) over(order by Id rows between 1 preceding and 1 following) as [count],  SUM(Salary)over(order by Id rows between 1 preceding and 1 following) as [sum]  from Employees\_testing\_Running\_sum | **Mark Male 5000 4750 2 9500**  **John Male 4500 5000 3 15000**  **Pam Female 5500 4666 3 14000**  **Sara Female 4000 4333 3 13000**  **Todd Male 3500 4166 3 12500**  **Mary Female 5000 5000 3 15000**  **Ben Male 6500 6166 3 18500**  **Jodi Female 7000 6333 3 19000**  **Tom Male 5500 5833 3 17500**  **Ron Male 5000 5250 2 10500** |
| Select Name, Gender, Salary,  AVG(Salary)over(partition by Gender order by Id) as [Average],  SUM(Salary) over(partition by Gender order by Id) as [Sum],  COUNT(Id) over (partition by Gender order by Id) as [Count]  from Employees\_testing\_Running\_sum | **Pam Female 5500 5500 5500 1**  **Sara Female 4000 4750 9500 2**  **Mary Female 5000 4833 14500 3**  **Jodi Female 7000 5375 21500 4**  **Mark Male 5000 5000 5000 1**  **John Male 4500 4750 9500 2**  **Todd Male 3500 4333 13000 3**  **Ben Male 6500 4875 19500 4**  **Tom Male 5500 5000 25000 5**  **Ron Male 5000 5000 30000 6** |
| Select Name, Gender, Salary,  AVG(Salary)over(partition by Gender order by Id rows between unbounded preceding and unbounded following) as [Average],  SUM(Salary) over(partition by Gender order by Id rows between unbounded preceding and unbounded following) as [Sum],  COUNT(Id) over (partition by Gender order by Id rows between unbounded preceding and unbounded following) as [Count]  from Employees\_testing\_Running\_sum | **Pam Female 5500 5375 21500 4**  **Sara Female 4000 5375 21500 4**  **Mary Female 5000 5375 21500 4**  **Jodi Female 7000 5375 21500 4**  **Tom Male 5500 5000 30000 6**  **Ron Male 5000 5000 30000 6**  **Ben Male 6500 5000 30000 6**  **Todd Male 3500 5000 30000 6**  **Mark Male 5000 5000 30000 6**  **John Male 4500 5000 30000 6** |

**RANGE vs ROWS in T-SQL?**

So, the main difference between ROWS and RANGE is in the way duplicate rows are treated. ROWS treat duplicates as distinct values, where as RANGE treats them as a single entity

|  |  |  |
| --- | --- | --- |
| 1 Mark 1000  2 John 2000  3 Pam 3000  4 Sara 4000  5 Todd 5000 | SELECT Name, Salary,  sum(Salary) over(order by Salary) as[no tie row]  from Employees\_testing\_ROWS\_VS\_RANK 🡪🡪🡪 | Mark 1000 1000  John 2000 3000  Pam 3000 6000  Sara 4000 10000  Todd 5000 15000 |
| works perfectly, Not using any EXPLICIT VALUE, THAT IS WHY SQL IS USING DEFAULT VALUE IS(RANGE BETWEEN UNBOUNDED PROCEDING AND CURRENT FOLLOWING) | | |
| SELECT Name, Salary,  SUM(Salary) over(order by Salary range between unbounded preceding and current row) as [no tie row]  from Employees\_testing\_ROWS\_VS\_RANK | select Name, Salary,  SUM(Salary) over (order by Salary rows between unbounded preceding and current row) As [no tie row]  from Employees\_testing\_ROWS\_VS\_RANK | Mark 1000 1000  John 2000 3000  Pam 3000 6000  Sara 4000 10000  Todd 5000 15000 |
| SELECT Name, Salary,  SUM(Salary) over(order by Salary range between unbounded preceding and unbounded following) as [no tie row]  from Employees\_testing\_ROWS\_VS\_RANK | | Mark 1000 15000  John 2000 15000  Pam 3000 15000  Sara 4000 15000  Todd 5000 15000 |
| select Name, Gender, Salary,  SUM(Salary) over (order by Salary **rows between unbounded preceding and current row**) as [tie row]  from Employees\_testing\_Running\_sum | | Todd Male 3500 3500  Sara Female 4000 7500  John Male 4500 12000  Mark Male 5000 17000  Mary Female 5000 22000  Ron Male 5000 27000  Tom Male 5500 32500  Pam Female 5500 38000  Ben Male 6500 44500  Jodi Female 7000 51500 |
| select Name, Gender, Salary,  SUM(Salary)over(order by Salary **range between unbounded preceding and current row**) as [tie row]  from Employees\_testing\_Running\_sum | | Todd Male 3500 3500  Sara Female 4000 7500  John Male 4500 12000  Mark Male 5000 **27000**  Mary Female 5000 **27000**  Ron Male 5000 **27000**  Tom Male 5500 38000  Pam Female 5500 38000  Ben Male 6500 44500  Jodi Female 7000 51500 |

**\*\* UNBOUNDED PRECEDING🡪**

Function starts execution with the first row of the table.

**\*\* CURRENT ROW🡪**

Function runs up to the current row as expecting.

**OFFSET – FETCH clause in T-SQL?**

The OFFSET-FETCH clause provides you with an option to fetch only a window or page of results from the result set. OFFSET-FETCH can be used only with the ORDER BY clause.

**Limitations of OFFSET – FETCH🡪**

* ORDER BY is mandatory to use OFFSET and FETCH clause.
* OFFSET clause is mandatory with FETCH. You can never use, ORDER BY … FETCH.
* TOP cannot be combined with OFFSET and FETCH in the same query expression.
* The OFFSET/FETCH rowcount expression can be any arithmetic, constant, or parameter expression that will return an integer value. The rowcount expression does not support scalar sub-queries

|  |  |
| --- | --- |
| select Name+' '+ cast(Gender as nvarchar(10))+' '+cast(Salary as nvarchar(10))  from Employees\_testing\_Running\_sum  order by Name offset 4 rows | Mary Female 5000  Pam Female 5500  Ron Male 5000  Sara Female 4000  Todd Male 3500  Tom Male 5500 |
| select Name+' '+ cast(Gender as nvarchar(10))+' '+cast(Salary as nvarchar(10))  from Employees\_testing\_Running\_sum  order by Name  offset 4 rows fetch next 2 rows only | Mary Female 5000  Pam Female 5500 |
| select Name+' '+ cast(Gender as nvarchar(10))+' '+cast(Salary as nvarchar(10))  from Employees\_testing\_Running\_sum  order by Name  offset 4 rows fetch FIRST 2 rows only | Mary Female 5000  Pam Female 5500 |

**OFFSET { integer\_constant | offset\_row\_count\_expression } { ROW | ROWS }**

Specifies the number of rows to skip, before starting to return rows from the query expression. The argument for the OFFSET clause can be an integer or expression that is greater than or equal to zero. You can use ROW and ROWS interchangeably.

**FETCH { FIRST|NEXT } <rowcount expression> { ROW|ROWS } ONLY**

Specifies the number of rows to return, after processing the OFFSET clause. The argument for the FETCH clause can be an integer or expression that is greater than or equal to one. You can use ROW and ROWS interchangeably. Similarly, FIRST and NEXT can be used interchangeably.

**LAST\_VALUE() is T-SQL🡪**

* Returns the last value in an ordered set of values in SQL Server 2012.
* Is the same type as scalar\_expression.
* LAST\_VALUE is nondeterministic.

**SYNTAX:**  **LAST\_VALUE ( [scalar\_expression ) OVER ( [ partition\_by\_clause ] order\_by\_clause rows\_range\_clause )**

|  |  |
| --- | --- |
| select top (1) Salary  from Employees\_testing\_Running\_sum  order by Salary desc | 7000 |
| select Name, Gender, Salary,  LAST\_VALUE(Name)over(order by Salary range between unbounded preceding and unbounded following) as sort  from Employees\_testing\_Running\_sum  select Name, Gender, Salary,  LAST\_VALUE(Name)over(order by Salary rows between unbounded preceding and unbounded following) as sort  from Employees\_testing\_Running\_sum | Todd Male 3500 Jodi  Sara Female 4000 Jodi  John Male 4500 Jodi  Mark Male 5000 Jodi  Mary Female 5000 Jodi  Ron Male 5000 Jodi  Tom Male 5500 Jodi  Pam Female 5500 Jodi  Ben Male 6500 Jodi  Jodi Female 7000 Jodi |
| select Name, Gender, Salary,  LAST\_VALUE(Name)over(partition by Gender order by Salary rows between unbounded preceding and unbounded following) as sort  from Employees\_testing\_Running\_sum | Sara Female 4000 Jodi  Mary Female 5000 Jodi  Pam Female 5500 Jodi  Jodi Female 7000 Jodi  Todd Male 3500 Ben  John Male 4500 Ben  Mark Male 5000 Ben  Ron Male 5000 Ben  Tom Male 5500 Ben  Ben Male 6500 Ben |
| select Name, Gender, Salary,  NTILE(5)over(order by(LAST\_VALUE(Name)over(order by Salary rows between unbounded preceding and unbounded following))) as sort  from Employees\_testing\_Running\_sum | Msg 4109, Level 15, State 1, Line 2  Windowed functions cannot be used in the context of another windowed function or aggregate. |

**DETERMINISTIC & NON- DETERMINISTIC functions in T-SQL🡪**

Always deterministic--> (All built-in functions are deterministic)

|  |  |  |
| --- | --- | --- |
| ABS | DATEDIFF | POWER |
| ACOS | DAY | RADIANS |
| ASIN | DEGREES | ROUND |
| ATAN | EXP | SIGN |
| ATN2 | FLOOR | SIN |
| CEILING | ISNULL | SQUARE |
| COALESCE | ISNUMERIC | SQRT |
| COS | LOG | TAN |
| COT | LOG10 | YEAR |
| DATALENGTH | MONTH |  |
| DATEADD | NULLIF |  |

Sometime DETERMINESTIC & sometime not:🡪

|  |  |
| --- | --- |
| all aggregate functions | All aggregate functions are deterministic unless they are specified with the OVER and ORDER BY clauses. For a list of these functions, see [Aggregate Functions (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql). |
| CAST | Deterministic unless used with **datetime**, **smalldatetime**, or **sql\_variant**. |
| CONVERT | Deterministic unless one of these conditions exists: Source type is **sql\_variant**. Target type is **sql\_variant** and its source type is nondeterministic. Source or target type is **datetime** or **smalldatetime**, the other source or target type is a character string, and a nondeterministic style is specified. To be deterministic, the style parameter must be a constant. Additionally, styles less than or equal to 100 are nondeterministic, except for styles 20 and 21. Styles greater than 100 are deterministic, except for styles 106, 107, 109 and 113. |
| CHECKSUM | Deterministic, with the exception of CHECKSUM(\*). |
| ISDATE | Deterministic only if used with the CONVERT function, the CONVERT style parameter is specified and style is not equal to 0, 100, 9, or 109. |
| RAND | RAND is deterministic only when a *seed* parameter is specified. |

NON-DETERMINESTIC🡪

All the configuration, cursor, metadata, security, and system statistical functions are nondeterministic.

[Configuration Functions (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/configuration-functions-transact-sql)🡪

|  |  |
| --- | --- |
| [@@DATEFIRST](https://docs.microsoft.com/en-us/sql/t-sql/functions/datefirst-transact-sql) | [@@OPTIONS](https://docs.microsoft.com/en-us/sql/t-sql/functions/options-transact-sql) |
| [@@DBTS](https://docs.microsoft.com/en-us/sql/t-sql/functions/dbts-transact-sql) | [@@REMSERVER](https://docs.microsoft.com/en-us/sql/t-sql/functions/remserver-transact-sql) |
| [@@LANGID](https://docs.microsoft.com/en-us/sql/t-sql/functions/langid-transact-sql) | [@@SERVERNAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/servername-transact-sql) |
| [@@LANGUAGE](https://docs.microsoft.com/en-us/sql/t-sql/functions/language-transact-sql) | [@@SERVICENAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/servicename-transact-sql) |
| [@@LOCK\_TIMEOUT](https://docs.microsoft.com/en-us/sql/t-sql/functions/lock-timeout-transact-sql) | [@@SPID](https://docs.microsoft.com/en-us/sql/t-sql/functions/spid-transact-sql) |
| [@@MAX\_CONNECTIONS](https://docs.microsoft.com/en-us/sql/t-sql/functions/max-connections-transact-sql) | [@@TEXTSIZE](https://docs.microsoft.com/en-us/sql/t-sql/functions/textsize-transact-sql) |
| [@@MAX\_PRECISION](https://docs.microsoft.com/en-us/sql/t-sql/functions/max-precision-transact-sql) | [@@VERSION](https://docs.microsoft.com/en-us/sql/t-sql/functions/version-transact-sql-configuration-functions) |
| [@@NESTLEVEL](https://docs.microsoft.com/en-us/sql/t-sql/functions/nestlevel-transact-sql) |  |

CORSOR FUNCTIONS(T-SQL)🡪

|  |  |
| --- | --- |
| [@@CURSOR\_ROWS](https://docs.microsoft.com/en-us/sql/t-sql/functions/cursor-rows-transact-sql) | [CURSOR\_STATUS](https://docs.microsoft.com/en-us/sql/t-sql/functions/cursor-status-transact-sql) |
| [@@FETCH\_STATUS](https://docs.microsoft.com/en-us/sql/t-sql/functions/fetch-status-transact-sql) |  |

METEDETA FUNCTIONS(T-SQL)🡪

The following scalar functions return information about the database and database objects:

|  |  |
| --- | --- |
| [@@PROCID](https://docs.microsoft.com/en-us/sql/t-sql/functions/procid-transact-sql) | [INDEX\_COL](https://docs.microsoft.com/en-us/sql/t-sql/functions/index-col-transact-sql) |
| [APP\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/app-name-transact-sql) | [INDEXKEY\_PROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/indexkey-property-transact-sql) |
| [APPLOCK\_MODE](https://docs.microsoft.com/en-us/sql/t-sql/functions/applock-mode-transact-sql) | [INDEXPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/indexproperty-transact-sql) |
| [APPLOCK\_TEST](https://docs.microsoft.com/en-us/sql/t-sql/functions/applock-test-transact-sql) | [NEXT VALUE FOR](https://docs.microsoft.com/en-us/sql/t-sql/functions/next-value-for-transact-sql) |
| [ASSEMBLYPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/assemblyproperty-transact-sql) | [OBJECT\_DEFINITION](https://docs.microsoft.com/en-us/sql/t-sql/functions/object-definition-transact-sql) |
| [COL\_LENGTH](https://docs.microsoft.com/en-us/sql/t-sql/functions/col-length-transact-sql) | [OBJECT\_ID](https://docs.microsoft.com/en-us/sql/t-sql/functions/object-id-transact-sql) |
| [COL\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/col-name-transact-sql) | [OBJECT\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/object-name-transact-sql) |
| [COLUMNPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/columnproperty-transact-sql) | [OBJECT\_SCHEMA\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/object-schema-name-transact-sql) |
| [DATABASE\_PRINCIPAL\_ID](https://docs.microsoft.com/en-us/sql/t-sql/functions/database-principal-id-transact-sql) | [OBJECTPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/objectproperty-transact-sql) |
| [DATABASEPROPERTYEX](https://docs.microsoft.com/en-us/sql/t-sql/functions/databasepropertyex-transact-sql) | [OBJECTPROPERTYEX](https://docs.microsoft.com/en-us/sql/t-sql/functions/objectpropertyex-transact-sql) |
| [DB\_ID](https://docs.microsoft.com/en-us/sql/t-sql/functions/db-id-transact-sql) | [ORIGINAL\_DB\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/original-db-name-transact-sql) |
| [DB\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/db-name-transact-sql) | [PARSENAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/parsename-transact-sql) |
| [FILE\_ID](https://docs.microsoft.com/en-us/sql/t-sql/functions/file-id-transact-sql) | [SCHEMA\_ID](https://docs.microsoft.com/en-us/sql/t-sql/functions/schema-id-transact-sql) |
| [FILE\_IDEX](https://docs.microsoft.com/en-us/sql/t-sql/functions/file-idex-transact-sql) | [SCHEMA\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/schema-name-transact-sql) |
| [FILE\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/file-name-transact-sql) | [SCOPE\_IDENTITY](https://docs.microsoft.com/en-us/sql/t-sql/functions/scope-identity-transact-sql) |
| [FILEGROUP\_ID](https://docs.microsoft.com/en-us/sql/t-sql/functions/filegroup-id-transact-sql) | [SERVERPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/serverproperty-transact-sql) |
| [FILEGROUP\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/filegroup-name-transact-sql) | [STATS\_DATE](https://docs.microsoft.com/en-us/sql/t-sql/functions/stats-date-transact-sql) |
| [FILEGROUPPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/filegroupproperty-transact-sql) | [TYPE\_ID](https://docs.microsoft.com/en-us/sql/t-sql/functions/type-id-transact-sql) |
| [FILEPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/fileproperty-transact-sql) | [TYPE\_NAME](https://docs.microsoft.com/en-us/sql/t-sql/functions/type-name-transact-sql) |
| [FULLTEXTCATALOGPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/fulltextcatalogproperty-transact-sql) | [TYPEPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/typeproperty-transact-sql) |
| [FULLTEXTSERVICEPROPERTY](https://docs.microsoft.com/en-us/sql/t-sql/functions/fulltextserviceproperty-transact-sql) | [VERSION](https://docs.microsoft.com/en-us/sql/t-sql/functions/version-transact-sql-metadata-functions) |

SEQUIRITY FUNCTIONS (T-SQL)🡪

|  |  |
| --- | --- |
| [CERTENCODED (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/certencoded-transact-sql) | [PWDCOMPARE (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/pwdcompare-transact-sql) |
| [CERTPRIVATEKEY (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/certprivatekey-transact-sql) | [PWDENCRYPT (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/pwdencrypt-transact-sql) |
| [CURRENT\_USER (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/current-user-transact-sql) | [SCHEMA\_ID (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/schema-id-transact-sql) |
| [DATABASE\_PRINCIPAL\_ID (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/database-principal-id-transact-sql) | [SCHEMA\_NAME (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/schema-name-transact-sql) |
| [sys.fn\_builtin\_permissions (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/sys-fn-builtin-permissions-transact-sql) | [SESSION\_USER (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/session-user-transact-sql) |
| [sys.fn\_get\_audit\_file (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/sys-fn-get-audit-file-transact-sql) | [SUSER\_ID (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/suser-id-transact-sql) |
| [sys.fn\_my\_permissions (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/sys-fn-my-permissions-transact-sql) | [SUSER\_SID (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/suser-sid-transact-sql) |
| [HAS\_PERMS\_BY\_NAME (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/has-perms-by-name-transact-sql) | [SUSER\_SNAME (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/suser-sname-transact-sql) |
| [IS\_MEMBER (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/is-member-transact-sql) | [SYSTEM\_USER (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/system-user-transact-sql) |
| [IS\_ROLEMEMBER (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/is-rolemember-transact-sql) | [SUSER\_NAME (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/suser-name-transact-sql) |
| [IS\_SRVROLEMEMBER (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/is-srvrolemember-transact-sql) | [USER\_ID (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/user-id-transact-sql) |
| [ORIGINAL\_LOGIN (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/original-login-transact-sql) | [USER\_NAME (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/user-name-transact-sql) |
| [PERMISSIONS (Transact-SQL)](https://docs.microsoft.com/en-us/sql/t-sql/functions/permissions-transact-sql) |  |

# **System Statistical Functions (Transact-SQL)🡪**

|  |  |
| --- | --- |
| [@@CONNECTIONS](https://docs.microsoft.com/en-us/sql/t-sql/functions/connections-transact-sql) | [@@PACK\_RECEIVED](https://docs.microsoft.com/en-us/sql/t-sql/functions/pack-received-transact-sql) |
| [@@CPU\_BUSY](https://docs.microsoft.com/en-us/sql/t-sql/functions/cpu-busy-transact-sql) | [@@PACK\_SENT](https://docs.microsoft.com/en-us/sql/t-sql/functions/pack-sent-transact-sql) |
| [fn\_virtualfilestats](https://docs.microsoft.com/en-us/sql/relational-databases/system-functions/sys-fn-virtualfilestats-transact-sql) | [@@TIMETICKS](https://docs.microsoft.com/en-us/sql/t-sql/functions/timeticks-transact-sql) |
| [@@IDLE](https://docs.microsoft.com/en-us/sql/t-sql/functions/idle-transact-sql) | [@@TOTAL\_ERRORS](https://docs.microsoft.com/en-us/sql/t-sql/functions/total-errors-transact-sql) |
| [@@IO\_BUSY](https://docs.microsoft.com/en-us/sql/t-sql/functions/io-busy-transact-sql) | [@@TOTAL\_READ](https://docs.microsoft.com/en-us/sql/t-sql/functions/total-read-transact-sql) |
| [@@PACKET\_ERRORS](https://docs.microsoft.com/en-us/sql/t-sql/functions/packet-errors-transact-sql) | [@@TOTAL\_WRITE](https://docs.microsoft.com/en-us/sql/t-sql/functions/total-write-transact-sql) |

**UNPIVOT operator in T-SQL🡪**

PIVOT operator turns ROWS into COLUMNS, where as UNPIVOT turns COLUMNS into ROWS.

|  |  |  |
| --- | --- | --- |
| David 960 520 360  John 970 540 800 | select SalesAgent, Country, product  from tblProductSales\_unpiot\_test  unpivot  (  product  for Country in(India, UK, US)  )as example | David India 960  David UK 360  David US 520  John India 970  John UK 800  John US 540 |
| with value  as  (  select Row\_Number() over(order by SalesAgent) as ID, SalesAgent, India, US, UK  from tblProductSales\_unpiot\_test  )  select ID, SalesAgent, SalesCountry, ammount  from value  unpivot  (  ammount  for SalesCountry in(India, US, UK)  )as pivtest |  | 1 David India 960  1 David US 520  1 David UK 360  2 John India 970  2 John US 540  2 John UK 800 |

**Pivot operator in T-SQL**🡪

|  |  |  |
| --- | --- | --- |
| Tom UK 200  John US 180  John UK 260  David India 450  Tom India 350  David US 200  Tom US 130  John India 540  John UK 120  David UK 220  John UK 420  David US 320  Tom US 340  Tom UK 660  John India 430  David India 230  David India 280  Tom UK 480  John US 360  David UK 140 | select SalesAgent, SalesCountry,Sum(SalesAmount)  from tblProductSales\_pivot\_test  group by SalesCountry, SalesAgent  order by SalesCountry, SalesAgent | Select SalesAgent, India, US, UK  From tblProductSales\_pivot\_test  PIVOT  (  Sum(SalesAmount)  For SalesCountry in(India, US< UK)  ) as example |
| David India 960  John India 970  Tom India 350  David UK 360  John UK 800  Tom UK 1340  David US 520  John US 540  Tom US 470 | David 960 520 360  John 970 540 800  Tom 350 470 1340 |
| 1 Tom UK 200  2 John US 180  3 John UK 260  4 David India 450  5 Tom India 350  6 David US 200  7 Tom US 130  8 John India 540  9 John UK 120  10 David UK 220  11 John UK 420  12 David US 320  13 Tom US 340  14 Tom UK 660  15 John India 430  16 David India 230  17 David India 280  18 Tom UK 480  19 John US 360  20 David UK 140 | Select SalesAgent, India, US, UK  From tblProductSales\_pivot\_test  PIVOT  (  Sum(SalesAmount)  For SalesCountry in(India, US< UK)  ) as example | David 450 NULL NULL  David NULL 200 NULL  David NULL NULL 220  David NULL 320 NULL  David 230 NULL NULL  David 280 NULL NULL  David NULL NULL 140  John NULL 180 NULL  John NULL NULL 260  John 540 NULL NULL  John NULL NULL 120  John NULL NULL 420  John 430 NULL NULL  John NULL 360 NULL  Tom NULL NULL 200  Tom 350 NULL NULL  Tom NULL 130 NULL  Tom NULL 340 NULL  Tom NULL NULL 660  Tom NULL NULL 480 |
| USING CTE🡪  with ash  as  (  select SalesAgent, SalesCountry, SalesAmount  from tblProductSales\_pivot\_test  )  select SalesAgent, India, US, UK  from ash  pivot  (  sum(SalesAmount)  for SalesCountry in ([India],[US],[UK])  )as pivoteaxp | USING DERIVED TABLE🡪  Select SalesAgent, India, US, UK  from  (  Select SalesAgent, SalesCountry, SalesAmount from tblProductSales\_pivot\_test  ) as SourceTable  Pivot  (  Sum(SalesAmount) for SalesCountry in (India, US, UK)  ) as PivotTable  🡪🡪🡪🡪🡪🡪 | David 960 520 360  John 970 540 800  Tom 350 470 1340 |

**The syntax of PIVOT operator from MSDN🡪**  
SELECT <non-pivoted column>,  
    [first pivoted column] AS <column name>,  
    [second pivoted column] AS <column name>,  
    ...  
    [last pivoted column] AS <column name>  
FROM  
    (<SELECT query that produces the data>)   
    AS <alias for the source query>  
PIVOT  
(  
    <aggregation function>(<column being aggregated>)  
FOR   
    [<column that contains the values that will become column headers>]   
    IN ( [first pivoted column], [second pivoted column], ... [last pivoted column])  
)   
AS <alias for the pivot table>  
<optional ORDER BY clause>;

\*\*\*\*\*If you have more columns in SOURCE TABLE THAN THE TABLE YOU WANT, then you have to use TABLE EXPRESSIONS…………………