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 **iPhone1500** **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

2. 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

3. 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→**

India	Female	4000
UK	Female	5000
USA	Female	12500
India	Male	8000
UK	Male	12000
USA	Male	10000

```
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	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	select Country,Gender, SUM(Salary) from Employees_for_grooping_set_for_grooping_set_2 group by grouping sets

	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	((Country, Gender), (Country), ())

Case 1		Case 2		
India	12000	India	Female	4000
UK	17000	India	Male	8000
USA	22500	India	NULL	12000
NULL	51500	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	India	Female	4000
	2	John	Male	4500	India	UK	Female	5000
	3	Pam	Female	5500	USA	USA	Female	12500
	4	Sara	Female	4000	India	NULL	Female	21500
	5	Todd	Male	3500	India	India	Male	8000
	6	Mary	Female	5000	UK	UK	Male	12000
	7	Ben	Male	6500	UK	USA	Male	10000
	8	Elli	Female	7000	USA	NULL	Male	30000
	9	Tom	Male	5500	UK	NULL	NULL	51500
	10	Ron	Male	5000	USA	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**	**India**	**Bangalore**	**1000**		**India**	**Bangalor**	**1000**
2	**Asia**	**India**	**Chennai**	**2000**	**NULL**	**Bangalore**	**1000**		**India**	**Chennai**	**2000**
3	**Asia**	**Japan**	**Tokyo**	**4000**	**India**	**Chennai**	**2000**		**India**	**NULL**	**3000**
					NULL	**Chennai**	**2000**		**Japan**	**Tokyo**	**4000**
					Japan	**Tokyo**	**4000**		**Japan**	**NULL**	**4000**
					NULL	**Tokyo**	**4000**		**NULL**	**NULL**	**7000**
					NULL	**NULL**	**7000**				

	India	**NULL**	**3000**
	Japan	**NULL**	**4000**

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)
```

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

```
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	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,
```

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

```
        GROUPING_ID(Continent, Country, City) AS GPID
from rollup_vs_cube
group by rollup(Continent, Country, City)
```

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
```

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

```
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	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()

- Returns the rank of rows within the partition of a result set, with gaps in the ranking.
- RANK is nondeterministic.
- RETURN type: bigint

DENSE_RANK()

- 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	

```
select *, RANK() over(order by
Salary desc) as Rank_test
from Employees_testing_rank
```

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

```
select *, dense_RANK() over(order by
Salary Desc) as Rank_test
from Employees_testing_rank
```

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 *, 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,
       ROW_NUMBER()over(order by Salary desc) as [row_number],
       RANK() over (order by Salary desc) as [rank],
       DENSE_RANK() over(order by Salary desc) as [dense_ranke]
from Employees_testing_rank
```

```
n
o
t
i
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],
       RANK() over(order by Salary desc) as [rank],
       DENSE_RANK() over(order by Salary desc) as [Dense_rank]
from Employees_testing_rank
```

```
t
i
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
```

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
```

```
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
```

```
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
```

```
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
```

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

```
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	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
```

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

```
select Name, Gender, Salary,
       FIRST_VALUE(Name)over(partition by Gender
order by Salary) as fb
from Employees_testing_Running_sum
```

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