SQL SERVER

SQL server is a repository, when we can store the organizational Data in the form of tables and views.

This SQL server is provided by the Microsoft in different kind of editions. We have following versions in the SQL Server

- SQL SERVER -2005
- SQL SERVER -2008----------(killimanjari)
- SQL SERVER -2008R2
- SQL SERVER -2012----------(Denali)
- SQL SERVER -2014

We have different kind of versions also in sql server

- enterprise edition----------(100% server machine)
- standard edition----------(single user machine)
- express edition ----------(only sql server no msbi)
- developer edition ----------(limited features)

How to connect SSMS (SQL server management studio) once we install the SQL server we will get ICON in the all programs

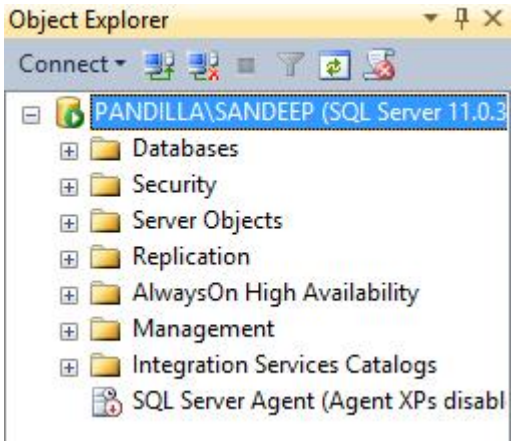
ALL PROGRAMS

↳ **MS SQL SERVER**

↳ **SQL SERVER MANAGEMENT STUDIO (SSMS)**

Then it will open a window provide the server name as or Local, (local host)

Click on connect then it will connect automatically, once we connect the server we will get following options.



Here the data will be stored in terms of database available in the server.

DDL (DATA DEFINATION LANGUAGE)

- CREATE
- ALTER
- DROP
- TRUNCATE

DML (DATA MANIPULATION LANGUAGE)

- INSERT
- UPDATE
- DELETE

TCL (TRANSACTION CONTROL LANGUAGE)

- ROLL BACK
- COMMIT

DDL Is 'DATA DEFINATION LANGUAGE 'and we have following commands in this. **"CREATE", "ALTER", "DROP" , "TRUNCATE".**

CREATE: It will create database as well as database objects in your database.

SYNTAX : CREATE DATABASE <DATABASE NAME>
EXAMPLE: CREATE DATABASE MY-DB

CREATION OF TABLE

Syntax:

```
CREATE TABLE <TABLE NAME>
(
<COLUMN NAME><DATATYPE>,
<COLUMN NAME><DATATYPE>,
<COLUMN NAME><DATATYPE>
)
```

EXAMPLE:

```
CREATE TABLE CUST
(
CUSTID INT,
CUSTNAME VARCHAR (50),
CUSTSAL INT
)
```

DROP: will remove database or database objects from the server

Syntax: `DROP TABLE <TABLE NAME>`

EXAMPLE: DROP TABLE CUST

DROPING DATABASE:

SYNTAX: DROP DATABASE <DATABASE NAME>

EXAMPLE: DROP DATABASE MY-DB

- DROP WILL REMOVE THE ENTIRE TABLE (STRUCTURE + DATA)
- DROP WILL BE APPLICABLE FOR DATABASE LEVEL AS WELL AS TABLE
- WE CAN DROP THE TABLE OR DATABASE BY USING GUI

TRUNCATE: TRUNCATE Will clean the data in the table level.

SYNTAX: TRUNCATE TABLE <TABLE NAME>

EXAMPLE: TRUNCATE TABLE CUST

- TRUNCATE WILL REMOVE ONLY DATA, CAN'T STRUCTURE.
- IT WILL APPLICABLE IN THE TABLE LEVEL ONLY
- IT IS NOT WORK FOR DATABSE TO TRUNCATE
- WE CAN'T ROLLBACK THE TRUNCATE

ALTER: WILL CHANGE THE EXISTING STRUCTURE OF THE TABLE AVAILABLE IN THE DATABSE.

*********Adding New column in the existing table********

Syntax: Alter table <table name > ADD <column name ><data type>

Example: Alter table CUST add custadd varchar(50)

Example: Alter table cust add custsal int,cusadd varchar(50),custhomeadd varchar(50)

*********Removing the existing column table********

Syntax: Alter table <table name > DROP <column name >

Example: Alter table CUST DROP custadd

Example: Alter table cust DROP custsal,cusadd,custhomeadd

********Modifying the data types of existing table********

Syntax: Alter table <table name > alter <column name ><data types>

Example: Alter table CUST ALTER custadd VARCHAR(50)

Example: Alter table cust ALTER custsal INT,custhomeadd VARCHAR(50)

DML DATA MANIPULATION LANGUAGE

INSERT: Insert will insert the data into corresponding tables. For the specified columns

******** **Insert for all columns**********

```
Insert into EMP values (100,'ABC', 45000)
```

******** **Insert for Selected columns**********

```
Insert into emp (empid,empname)  values(100,'ABC',)
```

******** **Insert for multiple columns**********

```
Insert into emp    values (102,'ABC', 35000), (103,'PARSHI',
65000), (104,'SAMBA', 55000), (105,'RAJU', 45000)
```

******** **Insert multiple ROWS Different Approach**********

```
Insert into EMP
Select      345,'NARESH', 55000 UNION ALL
Select      345,'SRIKANTH', 65000 UNION ALL
Select      234,'SAMBA', 75000
```

UPDATE: To update the existing data available in the table for corresponding columns which you provided in the update statement.

******** **Update for all rows**********

```
Update emp set empsalary=45000
```

******** **Update for selected rows**********

```
Update emp set empsalary=45000 whrere empid=345
```

******** **Update for multiple columns**********

```
Update emp set empsalary=45000 , empname='siva'
where empid in (345,123).
```

DELETE: It will delete the entire data or specific data from specified table.

- In this use where condition also
- Delete is a DML command
- We can rollback delete

******** **Delete entire data**********

```
Delete from emp
```

******** **Delete Specified data from the table**********

```
Delete from emp where empid=345
```

TCL TRANSACTION CONTROL LANGUAGE

TCL is a Transaction control language which will help us to do the transaction level operations for SQL statements.
COMMIT, ROLLBACK

ROLLBACK: Rollback will undo the transaction it something based on with provided statement in the TRANSACTION level.

Example: update the empname with 'SURESH'

```
Begin transaction
Update emp set empname='SURESH' where empid=345
Rollback
```

COMMIT: The changes which we made in the Transaction will be saved permanently in the table. Once the transaction saves, we can't rollback it.

Example:

```
Begin transaction
Update emp set empname='SURESH' where empid=345
Commit
```

Example:

```
Begin transaction transaction1---------------can rollback
Update emp set empname='SURESH' where empid=345

Begin transaction transaction2----------------cannot rollback
Update emp set empname='MAHESH' where empid=789
Save transaction transaction2

Begin transaction transaction3-------------------cannot rollback
Update emp set empname='rakesh' where empid=123
Save transaction transaction3
```

ROLLBACK TRANSACTION WILLS ROLLBACK THE UNSAVED TRANSATIONS.

DATA INSERTING INTO NEW TABLES THE FOLLOWING METHODS

******** **COPY THE STRUCTURE AND DATA INTO NEW TABLE********

Syntax

```
SELECT * INTO <NEWTABLE> FROM <OLD TABLE>

SELECT * INTO EMP_NEW FROM EMP
```

******** **COPY THE STRUCTURE ONLY TO THE NEW TABLE NOT THE DATA********

Syntax

```
SELECT * INTO <NEWTABLE> FROM <OLD TABLE>WHERE <FALSE CONDITION>

SELECT * INTO EMP_NEW FROM EMP WHERE 10=30
```

******** **COPY THE DATA INTO TABLE FROM OLD TABLE********

Syntax

```
INSERT INTO <EXISTING TABLE NAME> SELECT * FROM <OLD TABLE>

INSERT INTO EMPNEW SELECT * FROM EMP
```

OPERATORS:

=, <, >, <=, >=, !=

Example table: CUST

Cust id	Custname	Custbal
1	A	12000
2	B	14000
3	C	23000
4	D	19000
5	E	25000

```
SELECT * FROM CUST WHERE CUSTBAL    =     25000
                                    <     23000
                                    >     15000
                                    <=    23000
                                    >=    10000
                                    !=    29000
```

IN, AND, OR, BETWEEN, NOT IN

IN: If we are providing more than one value in the where condition for a specified column then we have to use in operator.

Example
```
select * from emp where empid in (1, 2, 3)
```

AND: AND operator is used to specified more than one condition in where clause,

Example
```
 select * from emp where empid =2 and empname=B
```

OR: if any one condition satisfy which is provided in the where clause then it will display the result.

Example
```
select * from emp where empid=3 or empname=D
```

BETWEEN: Between operators will be applicable for int, date, varchar fields.

Example
```
select * from emp Where empsal between 10000 and 45000
```

Example

```
select * from emp
Where substring (empname(1,1)) between  'A'  and 'K'
```

NOT IN: Not in will filter the data which we provided in the 'not in' clause and remaining rows will be displayed.

Example

```
select * from emp where empid  not in (1,2,3)
```

CONSTRAINTS

Not Null unique Check Default
Primary key foreign key

NOT NULL: Not null constraint will restrict the user, while inserting the null values into the specified columns.

Syntax:

```
create table <table name>
(
<Column name><data type> not null,
<Column name><data type>,
)
```

Example:

```
create table emp
(
empid int not null,
Empname varchar(50),
Empsal int,
)
```

In the above example not null value applied on the empid column whenever user inserts into the data it will restrict and throw the error.

```
Insert into emp values (null, 'A', 12000) ----------invalid
Insert into emp values (100,'B', 15000) ----------valid
```

HOW TO ADD NOT NULL CONSTRAINT IN EXISTING COLUMN

Syntax:

```
Alter table <table name>
Alter column <column name><data type> not null
```

Example:

```
Alter table emp alter column empid int not null
```

UNIQUE: Unique constraint will restrict the user to insert the duplicates to the specified columns.

Example:

```
create table cust_data
(
custid int unique,
Custname varchar(50),
Cusatbal int,
)

Insert into cust_data values (120,'samba', 56000) --------valid

Insert into cust_data values (120,'RAJU', 66000) -------Invalid

Insert into cust_data values (NULL,'SRIKANTH', 52000) -----valid

Insert into cust_data values (NULL,'CHANBASHA', 56000) ---Invalid
```

Unique constraint will not allow the duplicates as mentioned above but it will allow one null value to insert the column.

*******HOW TO ADD UNIQUE CONSTRAINT TO EXISTING COLUMN*******

Syntax:

```
Alter table cust
add constraint <constraint name> unique (<column name>)
```

Example:

```
Alter table cust add constrint uni_key unique (custid)
```

CHECK: Check constraint will check the values which are provided by the user and validate the condition, if check constraint satisfies then it will insert the data otherwise, it will throw an error.

Example:

```
create table cust_data_check
(
custid int ,
Custname varchar(50),
Custbal int check(custbal>45000)
)
Insert into cust_data_check values (123,'RAJU', 50000) -----VALID

Insert into cust_data_check values (234,'SAGAR', 34000) --INVALID
```

*******HOW TO ADD CHECK CONSTRAINT TO EXISTING COLUMN*******

Syntax:

```
Alter table cust_data_check
add constraint ck_key check (custbal>45000)
```

DEFAULT: Default will insert value which is specified by the user as "default" whenever user not providing any values for that specified column

Example:

```
create table custdata_default
(
custid int not null,
Custname varchar(50),
Custbal int,
Cuustlocation varchar(50) default 'BNG'
)

Insert into custdata_default values (120,'RAJU', 50000,'MUM')

Insert into custdata_default (CUSTID,CUSTNAME,CUSTBAL)
values (145,'SAGAR', 34000)
```

For custid 120 it will insert the location automatically which is provided by the user, But for custid 145 user is not providing any location to that, it will insert default location as 'BNG'

PRIMARY KEY: Primary key is a key which will restrict the user to allow null values as well as duplicates.

- It does not allow duplicates,
- It does not allow null values,
- Can create only one primary key in a table
- It works same as like combination of not null + unique

Example:

```
create table sampledata
(
custid int primary key,
Custname varchar(50),
Custbal int
)
Insert into sampledata values (123,'RAJU', 55000) ---------VALID
Insert into sampledata values (123,'SURESH', 50000) -----INVALID
Insert into sampledata values (NULL,'SRIKANTH', 58000) ---INVALID
```

HOW TO ADD PRIMARY KEY ON EXSISTING TABLE

```
Alter table sampledata
add constraint pk_custid primary key (custid)
```

Composite primary key:

if we create primary key on more than one column then that type of primary key is called as “composite primary key”

HOW TO ADD COMPOSITE PRIMARY KEY ON TABLE

First we have to make the required columns as ‘not null’ and then run the alter command

```
Alter table sampledata
add constraint pk_custid primary key(custid,custtxid)
```

Foreign key: foreign key is a reference key, which will insert the data in the’ fact customer’ table after checking it in the parent table.

- If the value is not present in the table it will throw an error while inserting data.
- Primary key and foreign key having parent and child relation in the tables

DIM PRODUCT(parent table)		
PID	**PNAME**	**AMOUNT**
1	LUX	30
2	SANTOOR	20
3	LIRIL	15
4	CINTHOL	40

FACT CUSTOMERSALE (child table)		
CUSTID	**PID**	**AMOUNT**
125	1	35
132	2	42
140	7	37
142	4	32
145	6	46

Parent table as DIM product table in this PID column as having primary key

```
Create table dimproduct
(
```

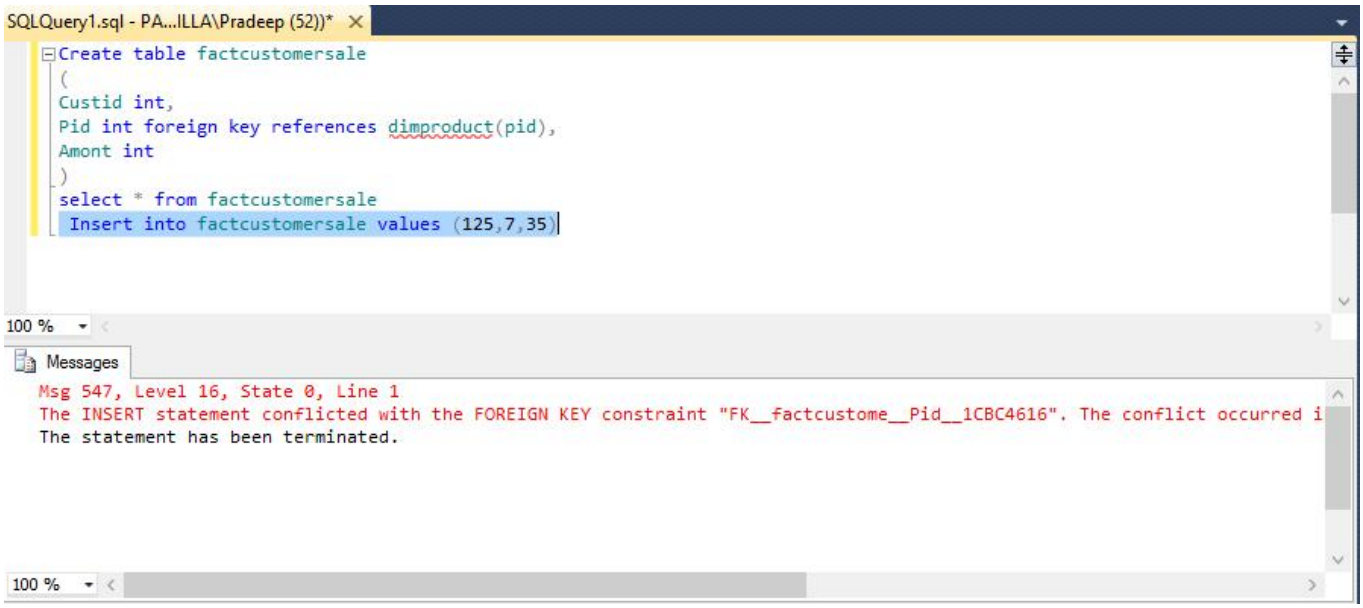
```
Pid int primary key,
Pname varchar(50),
Amount int
)

Insert into dimproduct values (1,'lux', 30)
Insert into dimproduct values (2,'santoor', 20)
Insert into dimproduct values (3,'liril', 15)
Insert into dimproduct values (4,'cinthol', 40)
```

Child table

```
Create table factcustomersale
(
Custid int,
Pid int foreign key references  dimproduct(pid),
Amont int
)

Insert into factcustomersale
values (125,1,35), (132,2,42), (140,7,37), (142,4,32), (145,6,46)
```



HOW TO ADD FOREIGN KEY TO EXISTING TABLE

```
Alter table factcustomersale
Add constraint fk_pid foreign key (pid) references
dimproduct(pid)
```

WHERE GROUP BY HAVING ORDER BY

WHERE: Where clause is used to filter the data in the tables for a specified column or multiple columns by using the given condition.

Example:
```
select *from dimcustomer
Where marital_status='s' and gendar = 'm'
```

GROUP BY: Group by will group the data based on the provided column in the group by clause to do the aggregate operations in the select list.

Example:
```
select product key, sum(salesamount) as totalsale,
MAX(SALESAMOUNT) AS MAXSALE
FROM FACTINTERESTSALE GROUP BY PRODUCTKEY
```

GROUP BY THUMB RULE:

The columns which are mentioned in the select clause must be in the group by, but the columns which are specified in the group by may or may not be specified in the select list.

HAVING: Having is used to filer the aggregated data after applying the group by clause in a table.

We can't apply having on direct columns it will be applicable only for aggregated columns
Whenever having is there, group by will be there before having.

Example:
```
select product key, sum (salesamount) as totalsale,
MAX(SALESAMOUNT) AS MAXSALE FROM FACTINTERESTSALE
GROUP BY PRODUCTKEY  having sum(salesamount)>50000
```

ORDEER BY: It is used to ascending or descending of the specified columns

Example:
```
select product key, sum (salesamount) as totalsale,
MAX(SALESAMOUNT) AS MAXSALE FROM FACTINTERESTSALE
GROUP BY PRODUCTKEY order by 2 desc
```

Example:
```
select * from dimproduct order by pid desc
```

CAST and CONVERT

CAST: Cast is used to convert the data type logically while selecting the data from the table for example

We have "saleamount" column in the table as "varchar" but we need as "int", Instead of changing data type physically design we can apply the cast and convert it in to logical way to do the operations.

Syntax:

```
select cast (saleamount as int) from cust
```

The disadvantage of cast is, if we are trying to convert a column into 'int' but we have varchar values in that field then it automatically fails the entire column.

TRY CAST: Try_ cast is cast but only the difference is 'cast' will fail entire column, if we have unwanted values in it. But 'TRYCAST' will not fail entire column and it will display null values in place of unwanted values.

Syntax:

```
select try_cast (salesamount as int) from cust
```

CONVERT: Convert is used to convert the data type from one to another logically while selecting data from table

Syntax:

```
select convert (int, salesamount) from cust
```

As same as try_cast we have try_convert also in sql server 2012.

Example:

```
select try_convert(int, salesamount) from cust
```

The advantage of in the convert we can change the date formats as we required for the specified column which is not possible by using "cost"

Example:

```
select convert (varchar(20), getdate(),105)

Select convert (varchar(50),BIRTHDATE(),105) From dimcustomer
```

JOINS

Joins will combine two or more tables to get the data from multiple tables based on the key column.

We have following types of join in SQL server

- INNER JOIN (JOIN)
- LEFT OUTER JOIN (LEFT JOIN)
- RIGHT OUTER JOIN (RIGHT JOIN)
- FULL OUTER JOIN (FULL JOIN)
- CROSS JOIN

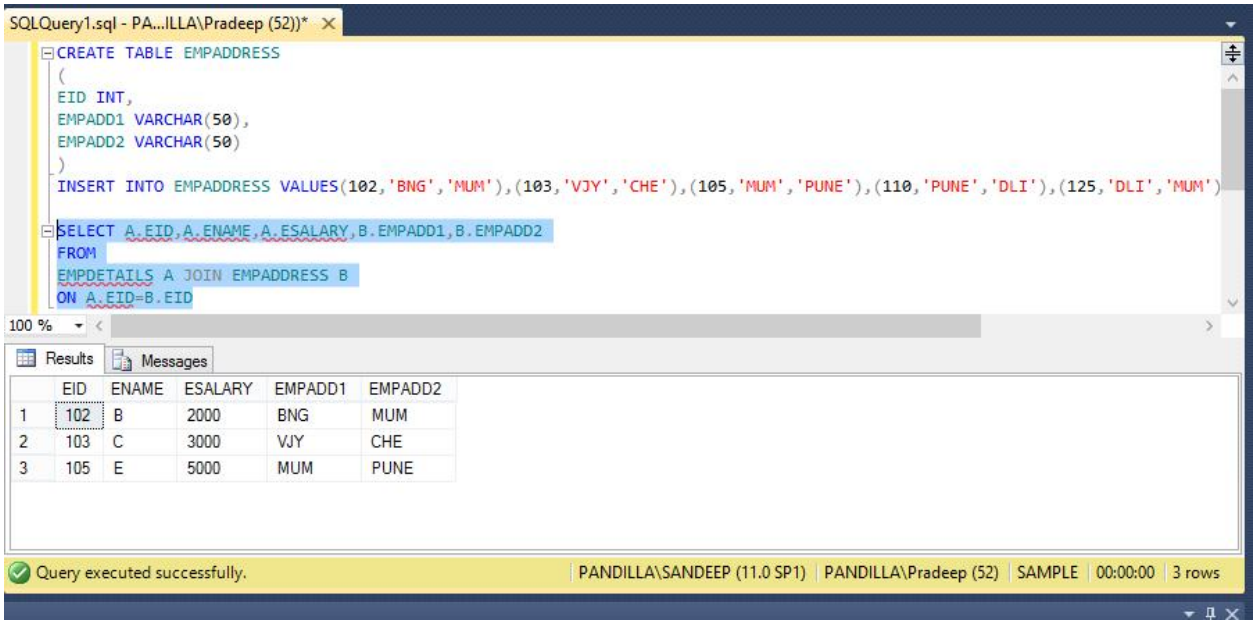
EMP DETAILS		
EID	**ENAME**	**ESALARY**
101	A	1000
102	B	2000
103	C	3000
104	D	4000
105	E	5000

EMP ADDRESS		
EID	**EADD1**	**EADD2**
102	BNG	MUM
103	VJY	CHE
105	MUM	PUNE
110	PUNE	DLI
125	DLI	MUM

INNER JOIN: Inner join will give you the making records from both the tables based on the key column.

Example:

```
SELECT
      A.EID,
      A.ENAME,
      A.ESALARY,
      B.EMPADD1,
      B.EMPADD2
FROM
EMPDETAILS A JOIN EMPADDRESS B
ON A.EID=B.EID
```



INNER LOOP JOIN: Inner loop join work as same as 'inner join', but we will use this loop join in following synario

If our left table having very less no of records and right table is having huge amount of records then inner loop join will works very fastly.

EXAMPLE:

```
SELECT
      A.EID,
      A.ENAME,
      A.ESALARY,
      B.EMPADD1,
      B.EMPADD2
FROM
EMPDETAILS A inner LOOP JOIN EMPADDRESS B
ON A.EID=B.EID
```

OUTPUT AS SAME AS INNER JOIN

LEFT OUTER JOIN (LEFT JOIN): This join will give all the records from the left table and matching records from the right table.

Example:

```
SELECT
      A.EID,
      A.ENAME,
      A.ESALARY,
      B.EMPADD1,
      B.EMPADD2
FROM
```

```
EMPDETAILS A LEFT JOIN EMPADDRESS B
ON A.EID=B.EID
```

Output:

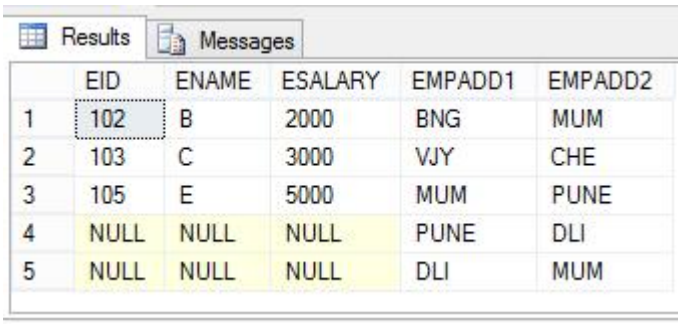
	EID	ENAME	ESALARY	EMPADD1	EMPADD2
1	101	A	1000	NULL	NULL
2	102	B	2000	BNG	MUM
3	103	C	3000	VJY	CHE
4	104	D	4000	NULL	NULL
5	105	E	5000	MUM	PUNE

RIGHT OUTER JOIN (RIGHT JOIN): This join will give all records from the right table and matching records from the left table.

Example:

```
SELECT
      A.EID,
      A.ENAME,
      A.ESALARY,
      B.EMPADD1,
      B.EMPADD2
FROM
EMPDETAILS A RIGHT JOIN EMPADDRESS B
ON A.EID=B.EID
```

Output:



	EID	ENAME	ESALARY	EMPADD1	EMPADD2
1	102	B	2000	BNG	MUM
2	103	C	3000	VJY	CHE
3	105	E	5000	MUM	PUNE
4	NULL	NULL	NULL	PUNE	DLI
5	NULL	NULL	NULL	DLI	MUM

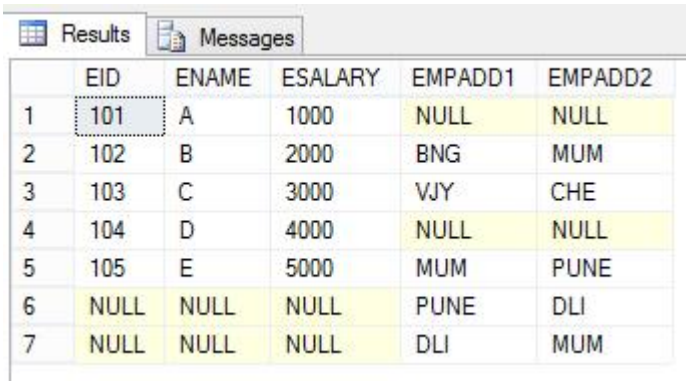
FULL OUTER JOIN (FULL JOIN): This join will give all the records from left table and all the records from right table (complete records from both the tables)

Example:

```
SELECT
      A.EID,
      A.ENAME,
      A.ESALARY,
      B.EMPADD1,
      B.EMPADD2
FROM
EMPDETAILS A full JOIN EMPADDRESS B
```

```
ON A.EID=B.EID
```

Output:



Results | Messages

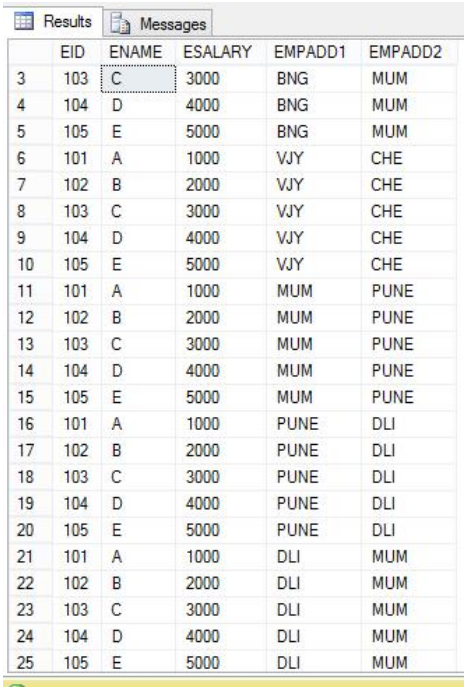
	EID	ENAME	ESALARY	EMPADD1	EMPADD2
1	101	A	1000	NULL	NULL
2	102	B	2000	BNG	MUM
3	103	C	3000	VJY	CHE
4	104	D	4000	NULL	NULL
5	105	E	5000	MUM	PUNE
6	NULL	NULL	NULL	PUNE	DLI
7	NULL	NULL	NULL	DLI	MUM

CROSS JOIN: Cross join will give all the possible combinations from the both tables

Example:

```
SELECT
      A.EID,
      A.ENAME,
      A.ESALARY,
      B.EMPADD1,
      B.EMPADD2
FROM
EMPDETAILS A CROSS JOIN EMPADDRESS B
```

OUTPUT:



Results | Messages

	EID	ENAME	ESALARY	EMPADD1	EMPADD2
3	103	C	3000	BNG	MUM
4	104	D	4000	BNG	MUM
5	105	E	5000	BNG	MUM
6	101	A	1000	VJY	CHE
7	102	B	2000	VJY	CHE
8	103	C	3000	VJY	CHE
9	104	D	4000	VJY	CHE
10	105	E	5000	VJY	CHE
11	101	A	1000	MUM	PUNE
12	102	B	2000	MUM	PUNE
13	103	C	3000	MUM	PUNE
14	104	D	4000	MUM	PUNE
15	105	E	5000	MUM	PUNE
16	101	A	1000	PUNE	DLI
17	102	B	2000	PUNE	DLI
18	103	C	3000	PUNE	DLI
19	104	D	4000	PUNE	DLI
20	105	E	5000	PUNE	DLI
21	101	A	1000	DLI	MUM
22	102	B	2000	DLI	MUM
23	103	C	3000	DLI	MUM
24	104	D	4000	DLI	MUM
25	105	E	5000	DLI	MUM

This cross join will give results like example if we have ‘m’ records in left table and ‘n’ records in the right table then we will get ‘m*n’ records .

SYSTEM FUNCTIONS

We have some predefined functions available in the SQL server to do the operations as per requirement. We have 4 types of system functions as following

- MATHEMATICAL FUNCTIONS
- AGGREGATE FUNCTIONS
- STRING FUNCTIONS
- DATE FUNCTIONS

AGGREAGTE FUNCTIONS: Aggregate functions will do aggregate operations like

SUM MAX MIN AVG COUNT -----etc

Example:
```
SELECT SUM (ESALARY) AS TOTAL_SALARY FROM EMPDETAILS
```

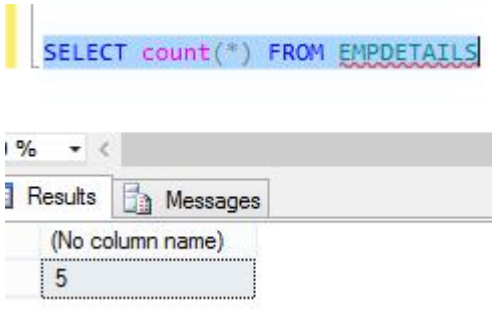
Example:
```
SELECT MAX (ESALARY) AS MAX_SALARY FROM EMPDETAILS
```

Example:
```
SELECT min (ESALARY) AS min_SALARY FROM EMPDETAILS
```

Example:
```
SELECT avg (ESALARY) AS avg_SALARY FROM EMPDETAILS
```

COUNT: It will display the total no of rows available in the table or specified column

Example:
```
SELECT count (*) FROM EMPDETAILS
```

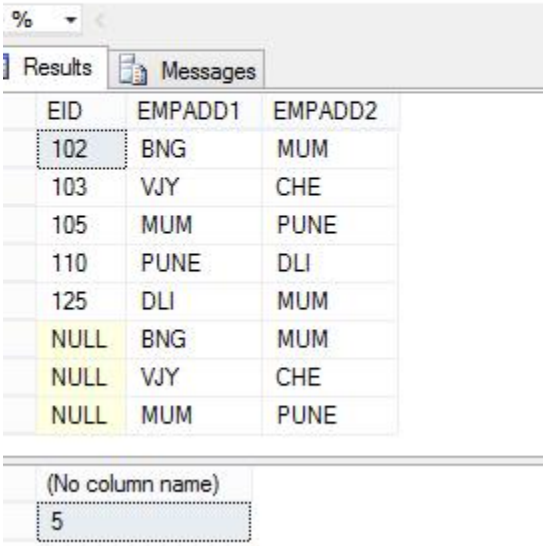


The table having 5 no’s of rows

Example: example for column

```
SELECT count (EID) FROM EMPADDRESS
```

```
select * from EMPADDRESS
SELECT count(EID) FROM EMPADDRESS
```



%

Results | Messages

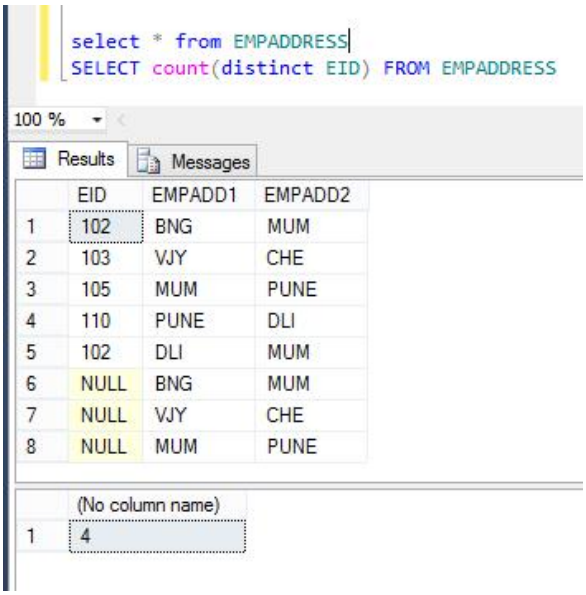
EID	EMPADD1	EMPADD2
102	BNG	MUM
103	VJY	CHE
105	MUM	PUNE
110	PUNE	DLI
125	DLI	MUM
NULL	BNG	MUM
NULL	VJY	CHE
NULL	MUM	PUNE

(No column name)
5

Example: Count distinct

```
SELECT count (distinct EID) FROM EMPADDRESS
```

```
select * from EMPADDRESS
SELECT count(distinct EID) FROM EMPADDRESS
```



100 %

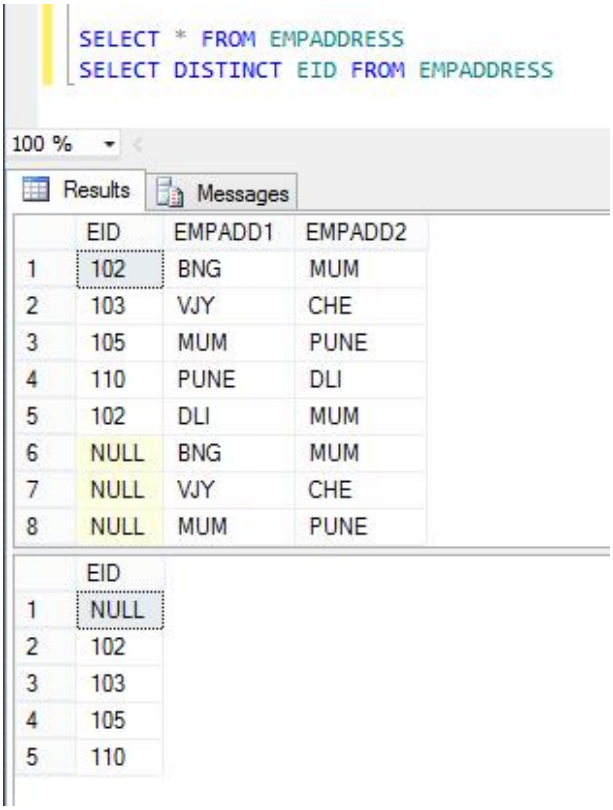
Results | Messages

	EID	EMPADD1	EMPADD2
1	102	BNG	MUM
2	103	VJY	CHE
3	105	MUM	PUNE
4	110	PUNE	DLI
5	102	DLI	MUM
6	NULL	BNG	MUM
7	NULL	VJY	CHE
8	NULL	MUM	PUNE

	(No column name)
1	4

DISTINCT WILL GIVE NO DUPLICATES

Example: SELECT DISTINCT EID FROM EMPADDRESS



```
SELECT * FROM EMPADDRESS
SELECT DISTINCT EID FROM EMPADDRESS
```

100 %

Results | Messages

	EID	EMPADD1	EMPADD2
1	102	BNG	MUM
2	103	VJY	CHE
3	105	MUM	PUNE
4	110	PUNE	DLI
5	102	DLI	MUM
6	NULL	BNG	MUM
7	NULL	VJY	CHE
8	NULL	MUM	PUNE

	EID
1	NULL
2	102
3	103
4	105
5	110

MATHEMATICAL FUNCTIONS: We can perform the mathematical operations to use these following functions

Sin() cos() tan() sqrt ceiling() floor() Round() abs() exponential() log() power()

Example:

```
select sin (0) -------o/p     0
Select sin (90) -------o/p    1

Select cos (0) --------o/p    1
Select cos (90) -------o/p    0

Select tan (0) ---------o/p   0

Select SQRT (225) -----------15
Select SQRT (169) -----------13
```

```
Select ceiling (45.5) --------- 46
Select ceiling (44.2) --------- 45

Select FLOOR (45.8) ----------45
Select FLOOR (56.8) ----------56

Select ROUND (23.122456748, 2) --------23.12
Select ROUND (23.129456748, 2) --------- 23.13

Select ABS (34.467) -----------absolute value 34.467

Select EXP (345) --------exponential 6.7857250200

Select log (5) ---------logarithms        1.6094379124341

Select POWER (5, 6) --------- 5^6    15625
```

STRING FUNCTIONS:

LEFT: Left function will display the specified no of characters in the given string or the column.

```
Example:    select LEFT ('MICROSOFT', 4)        MICR
            Select LEFT ('SQLSERVER', 5)        SQLSE
```

RIGHT: Right function will display the specified no of characters from the right side of the given string.

```
Example:    select right ('SQLSERVER', 5)       ERVER
            Select right ('microsoft', 5)       osoft
```

LENGTH: It will calculate the length of the given string or column.

```
Example     select len ('microsoft')                  9
            Select len ('MICROSOFT SQLSERVER')        19
```

LOWER: It will convert the given string or column into LOWER case.

```
Example:    select lower ('MICROSOFT SQLSERVER')      microsoft sqlserver
            Select lower ('MOTOROLA')                 Motorola
```

UPPER: It will convert the given string or column into UPPER case

```
Example     select upper ('MotoRoLa')                 MOTOROLA
            select upper ('microsoft')                MICROSOFT
```

LTRIM: It will trim the unwanted space from LEFT SIDE of the given string or column.

```
Example     Select LTRIM ('       microsoft')        microsoft
            Select UPPER ('       microsoft')              MICROSOFT
```

RTRIM: It will trim the unwanted space from RIGHT SIDE of the given string or column.

```
Example     Select LTRIM ('microsoft           ')    microsoft
            Select UPPER ('microsoft           ')    MICROSOFT
```

REVERSE: It will reverse the given string or column value specified in the reverse function.

```
Example     select reverse ('MICROSOFT')       TFOSORCIM
```

SUBSTRING: Substring will give required string in the given character or column.

```
Example     select substring ('MICROSOFT', 3, 7)      CROSOFT
            Select substring ('MICROSOFT', 2, 5)      ICROS
```

Example for table: - `select substring ('column name', 2, 5) from table_name`

REPLACE: It will replace the characters from the given string or column.

```
Example     Select REPLACE ('MICROSOFT','O','Z')      MICRZSZFT
            Select REPLACE ('SQLSERVER','E','Q')      SQLSQRVQR
```

EXAMPLE PROGRAM

```
SELECT UPPER (LEFT (EMPNAME, 1)) +
LOWER (SUBSTRING (EMPNAME, 2, LEN (EMPNAME)))
FROM EMPSALDETAILS
```

OUTPUT LIKE THESE

Chanbasha
Samba
Parshi
Karthik
Raju

CHAR INDEX: Char index will find the specified character position which is available in the given string.

It will return the position of the specified character for the first occurrence in the given string.

Example

```
select CHARINDEX ('s','MICROSOFT')            6
Select CHARINDEX ('L','INDIAN RAIL WAY')      11
```

REPLICATE: Replicate will display the same values for this specified no of times the given string or column.

Example

```
Select REPLICATE ('micro   ', 4)   micro   micro   micro   micro
Select REPLICATE ('A', 5)          AAAAA
Select REPLICATE (6, 10)           6666666666
```

STUFF: Stuff is as same as replace to replace the character with other string or character.

In stuff function we will always provide the range to replace the characters but not original string.

Example

```
Select STUFF ('ABCDEFGH', 2, 5,'XYZ')    AXYZGH
Select STUFF ('MICROSOFT', 1, 1,'XYZ')   XYZICROSOFT
```

EXAMPLE PROGRAMS

```
SELECT REPLICATE ('0',
(SELECT (MAX (LEN (EMPID))) FROM EMP1)-LEN (EMPID))
 +EMPID FROM EMP1
```

OUTPUT LIKE THIS

```
001
002
001
004
001
321
321
```

DATE FUNCTIONS

We have following date functions available in the SQL server to perform the date operations.

- **GETDATE()**
- **YEAR()**
- **MONTH()**
- **DAY()**
- **DATEPART**

- **DATEADD**
- **DATENAME**
- **DATEDIFF**

GETDATE (): Getdate will give present date & time from the server

Example

```
Select GETDATE ()       2015-02-22 23:40:35.68
```

YEAR (): This function will extract the year from the given date.

Example

```
Select YEAR (GETDATE ())            2015
Select YEAR ('1988-01-13')          1988
```

MONTH (); This function will extract the MONTH from the given date.

Example

```
select MONTH ('1988-01-13')         1
SELECT MONTH (GETDATE ())           2
```

DAY (): This function will extract the DAY from the given date.

Example

```
select DAY ('1988-01-13')           13
SELECT DAY (GETDATE ())             22
```

DATEPART: Datepart is used to extract the required part of the date like you month,year,day,…..etc

YY - YEAR
MM - MONTH
DD - DAY
HH - HOUR
MI - MINUTES
SS - SECONDS
MS - MILLISECONDS
DW - DAY WEEK
WW - WEEK NUMBER

Examples

```
SELECT DATEPART (YY, GETDATE ())          2015
SELECT DATEPART (MM,'1988-01-13')         1
SELECT DATEPART (DD, GETDATE ())          22
SELECT DATEPART (HH, GETDATE ())          23
SELECT DATEPART (DW, GETDATE ())          2
SELECT DATEPART (WW, GETDATE ())          9
```

DATEADD: Dateadd function will add the mentioned part of the date to this specified date value.

Example

```
SELECT DATEADD (YY, -1, '2013-04-23')
                        2012-04-23 00:00:00.000
```

```
SELECT DATEADD (MM, 5, '2013-04-23')
                          2013-09-23 00:00:00.000
SELECT DATEADD (DD, 17, GETDATE ())
                          2015-03-12 00:06:38.933
```

DATENAME: It will display the month name or date name from the given date value.

```
Example     SELECT DATENAME (MM, GETDATE ())        February
            SELECT DATENAME (DW, GETDATE ())        Monday
            SELECT DATENAME (DW, '1988-01-13')      Wednesday
```

DATEDIFF: It will calculate the difference between two given dates

```
Example     SELECT DATEDIFF (YY, '1988-01-13', GETDATE ())      27
            SELECT DATEDIFF (MM, '1988-01-13', GETDATE ())      325
```

EOMONTH () EOMONTH Will give the end of the month values.

```
Example     SELECT EOMONTH ('1988-01-13') 1988-01-31
            SELECT EOMONTH (GETDATE ())       2015-02-28
```

ISDATE() Checking for valid date

```
Example     SELECT ISDATE ('1988-01-13')      1
            SELECT ISDATE ('1988-20-13')      0
```

```
1) DISPLAY STARTING DATE OF CURRENT MONTH?

SELECT DATEADD (DD, -DAY (GETDATE ()-1), GETDATE ())
                                  2015-02-01 00:31:08.680

2) DISPLAY LAST DATE OF THE PREVIOUS MONTH?

SELECT DATEADD (DD, -DAY (GETDATE ()), GETDATE ())
                                  2015-01-31 00:30:05.357
```

SET OPERATORS

UNIONALL UNION INTERSECT EXCEPT

UNION ALL: Union all will combine two data inputs and display as single result set.

For example if we have 5 records in the first table and 10 records in the second table then we will get 10+5 records

The disadvantage of union all is it will consider the duplicates as well.

FIRST TABLE	
ID	**NAME**
1	SAMBA
2	SAGAR

SECOND TABLE	
ID	**NAME**
2	SAMBA
3	RAJU

```
SELECT * FROM FIRST_TABLE
UNOIN ALL
SELECT * FROM SECOND_TABLE
```

OUTPUT TABLE LIKE THIS

UNION ALL	
ID	**NAME**
1	SAMBA
2	SAGAR
2	SAMBA
3	RAJU

NOTE: DATA TYPE SHOULD BE SAME FOR BOTH TABLES

NOTE: THE COLUMNS WHICH WE MENTIONED IN THE SELECT LIST SHOULD BE SAME WITH OTHER TABLE, IT WILL NOT ALLOW THE DIFFERENT NO OF COLUMNS FOR BOTH TABLES

UNION: UNION will combine two data inputs and display as single results set.

For example if we have 5 records in the first table and 10 records in the second table then it will combine and remove duplicates and display as.

Mentioned above tables output for union like this

```
SELECT * FROM FIRST_TABLE
UNOIN
SELECT * FROM SECOND_TABLE
```

UNION	
ID	**NAME**
1	SAMBA
2	SAGAR
3	RAJU

Except: except will give all the records from the left table which are not available in the right table

```
SELECT * FROM FIRST_TABLE
EXCEPT
SELECT * FROM SECOND_TABLE
```

EXCEPT	
ID	**NAME**
1	SAMBA

INTERSECT: It will display the matching records from both the tables

```
SELECT * FROM FIRST_TABLE
INTERSECT
SELECT * FROM SECOND_TABLE
```

INTERSECT	
ID	**NAME**
2	SAMBA

DIFFERENCES BETWEEN INTERSECT AND INNERJOIN

INTERSECT	INNERJOIN
The no of columns should be Same for both the tables	No need to maintain same columns for both the tables
It will compare the null value and display to the results set	It will not compare the null value in the tables.

IS NULL, ISNULL ()

IS NULL: Generally if we want to find the NULL values in a specified column of a table we will use the following syntax to get the NULL values.

Example

```
select * from emp1 where empid = null ----invalid

Select * from emp1 where empid is null ----valid
```

We cannot put = operator for null why because one null value will not compare with other null value.

ISNULL (): ISNULL () operator will replace the user value in place of null.

Example

```
Select ISNULL (empid, 123) from emp1
```

CASE

Case condition will validate the given condition and it will apply the provided logic to the data based on the given condition.

Syntax

```
SELECT
CASE
```

```
                <CONDITION>
                ELSE
                <CONDITION>
                END
```

Example

```
                select marital_status,
                Case
                When marital_status = 'M' then 'married'
                When marital_status = 'S' then 'single'
                End as maritalstatus from employee
```

Example

```
                SELECT DPD,
                CASE
                WHEN DPD <= 30 THEN '1ST BUCKET'
                WHEN DPD <= 60 THEN '2ND BUCKET'
                WHEN DPD <= 90 THEN '3RD BUCKET'
                END AS DPD_BUCKET
                FROM CUSTOMER_DETAILS
```

Example

```
                SELECT *,
                CASE
            WHEN MATHS > 35 AND PHYSICS > 35 AND CHEMISTRY > 35 THEN 'PASS'
            ELSE 'FAIL'
            END as STATUS
            FROM RANKING
```

OUTPUT TABLE FOR STUDENT PASS FAIL STATUS

STUID	STUNAME	MATHS	PHYSICS	CHEMISTRY	TOTAL	STATUS
1	A	75	46	93	214	PASS
2	B	82	92	**37**	211	PASS
3	C	93	38	27	158	FAIL
4	D	28	67	91	186	FAIL
5	E	63	89	28	180	FAIL
6	F	79	83	92	254	PASS
7	G	89	92	86	267	PASS
8	H	36	76	74	186	PASS

RANKING FUNCTIONS

- ROW_NUMBER ()
- RANK ()
- DENSE_RANK ()

For example table

STUID	STUNAME	MATHS	PHYSICS	CHEMISTRY	TOTAL
1	A	75	46	93	214
2	B	82	92	**37**	211
3	C	93	38	27	158
4	D	28	67	91	186
5	E	63	89	28	180
6	F	79	83	92	254
7	G	89	92	86	267
8	H	36	76	74	186

DISPLAYING ROW NUMBER: ROW Number will generate auto generated number based on the given column order by.

Example

```
SELECT ROW_NUMBER ()
OVER (ORDER BY STUID) AS ROLL_NO, * FROM RANKING
```

OUTPUT TABLE

ROLL_NO	STUID	STUNAME	MATHS	PHYSICS	CHEMISTRY	TOTAL
1	1	A	75	46	93	214
2	2	B	82	92	**37**	211
3	3	C	93	38	27	158
4	4	D	28	67	91	186
5	5	E	63	89	28	180
6	6	F	79	83	92	254
7	7	G	89	92	86	267
8	8	H	36	76	74	186

RANK (): Rank will generate the values based on the given column in the order by clause as same as Roll NO. But rank will skip the intermediate values on the given column.

Example

```
SELECT RANK () OVER (ORDER BY TOTAL) AS RANK_NO, * FROM RANKING
```

OUTPUT TABLE FOR RANK_NO

RANK_NO	STUID	STUNAME	MATHS	PHYSICS	CHEMISTRY	TOTAL
1	3	C	93	38	27	158
2	5	E	63	89	**28**	180
3	8	H	36	76	74	186
3	4	D	28	67	91	186
5	2	B	82	92	37	211
6	1	A	75	46	93	214
7	6	F	79	83	92	254
8	7	G	89	92	86	267

DENSE RANK (): Dense_rank will generate the values based on the given column in the order by clause as same as roll no but Dense_rank will not skip the intermediate values on the given column.

Example

```
SELECT DENSE_RANK ()
OVER (ORDER BY TOTAL) AS DENSE_RANK, * FROM RANKING
```

OUTPUT TABLE FOR DENSE_RANK

DENSE_RANK	STUID	STUNAME	MATHS	PHYSICS	CHEMISTRY	TOTAL
1	3	C	93	38	27	158
2	5	E	63	89	**28**	180
3	8	H	36	76	74	186
3	4	D	28	67	91	186
4	2	B	82	92	37	211
5	1	A	75	46	93	214
6	6	F	79	83	92	254
7	7	G	89	92	86	267

Example for combining of ranking functions

```
SELECT ROW_NUMBER () OVER (ORDER BY TOTAL) AS ROLL_NO,
RANK () OVER (ORDER BY TOTAL) AS RANK_NO,
DENSE_RANK () OVER (ORDER BY TOTAL) AS DENSE_RANK, * FROM RANKING
```

OUTPUT FOR RANKING FUNCTIONS

ROLL_NO	RANK_NO	DENSE_RANK	STUID	STUNAME	MATHS	PHYSICS	CHEMISTRY	TOTAL
1	1	1	3	C	93	38	27	158
2	2	2	5	E	63	89	**28**	180
3	3	3	8	H	36	76	74	186
4	3	3	4	D	28	67	91	186
5	5	4	2	B	82	92	37	211
6	6	5	1	A	75	46	93	214
7	7	6	6	F	79	83	92	254
8	8	7	7	G	89	92	86	267

PARTITION BY: Partition by is as similar as group by and it will group the data in the specified column and generate the row no's based on it.

Example

```
SELECT ROW_NUMBER ()
OVER (PARTITION BY EMPDEPT ORDER BY EMPSAL DESC)
AS ROLL_NO,* FROM empsaldetails
```

Output like this

ROLL_NO	EMPID	EMPNAME	EMPSAL	EMPDEPT
1	101	SAMBA	45000	SE
2	100	CHANBASHA	35000	SE
1	106	BINDHU KUMAR	54000	SSE
2	103	KARTHIK	42000	SSE
3	102	PARSHI	38000	SSE
1	105	NARESH	52000	TL
2	104	RAJU	45000	TL

CTE (COMMON TABLE EXPRESSION)

CTE is common table expression and if will be a derived table which holds the results of an inner query.

- CTE is a logical table which will not create physically in the database.
- Once CTE got created we have to use it immediate to the creation.
- The data will be stored in the CTE always logical not stores as physical.
- Whenever we are doing operations on CTE that will affect the physical table also.

Syntax

```
WITH CTE
AS
(
SELECT ROW_NUMBER ()
OVER (ORDER BY EMPSAL DESC) AS ROLL_NO,* FROM EMPSALDETAILS

)

SELECT * FROM CTE

OR

DELETE FROM CTE WHERE ROLL_NO=5
```

HOW TO DELETE DUPLICATE IN TABLE

Example

```
------ CREATE CTE----

      WITH CTE
      AS
      (
SELECT ROW_NUMBER ()
OVER (PARTITION BY EMPID, EMPNAME, EMPSAL ORDER BY EMPSAL DESC) AS
ROLL_NO,* FROM EMPSALDETAILS

      )

-----DELETING THE DUPLICATE----

DELETE FROM CTE WHERE ROLL_NO > 1
```

We can have multiple columns for partition to identify the duplicates

Examples for CTE

EMPSALDETAILS			
EMPID	**EMPNAME**	**EMPSAL**	**EMPDEPT**
101	SAMBA	45000	SE
100	CHANBASHA	35000	SE
106	BINDHU KUMAR	54000	SSE
103	KARTHIK	42000	SSE
102	PARSHI	38000	SSE
105	NARESH	52000	TL
104	RAJU	45000	TL

1) MIN SALARY FOR EACH DEPARTMENT

```
WITH CTE
AS
(
SELECT ROW_NUMBER () OVER (PARTITION BY EMPDEPT ORDER BY EMPSAL DESC) AS
ROLL_NO,* FROM EMPSALDETAILS

)

SELECT MIN (EMPSAL) AS MIN_SAL, EMPDEPT FROM CTE GROUP BY EMPDEPT
```

MIN_SAL	EMPDEPT
35000	SE
38000	SSE
45000	TL

2) 6TH MAX SALARY FROM EMPSALDETAILS

```
WITH CTE
AS
(
SELECT ROW_NUMBER () OVER (ORDER BY EMPSAL DESC) AS ROLL_NO,* FROM
EMPSALDETAILS
)

SELECT * FROM CTE WHERE ROLL_NO=6
```

ROLL_NO	EMPID	EMPNAME	EMPSAL	EMPDEPT
6	102	PARSHI	38000	SSE

3) LOWEST SALARY IN EMP LIST

```
WITH CTE
AS
(
SELECT ROW_NUMBER () OVER (ORDER BY EMPSAL ASC) AS ROLL_NO,* FROM
EMPSALDETAILS
)

SELECT * FROM CTE WHERE ROLL_NO=1
```

ROLL_NO	EMPID	EMPNAME	EMPSAL	EMPDEPT
1	100	CHANBASHA	35000	SE

4) MAX SAL FOR EACH DEPT

```
WITH CTE
AS
(
SELECT ROW_NUMBER () OVER (PARTITION BY EMPDEPT ORDER BY EMPSAL DESC) AS
ROLL_NO,* FROM EMPSALDETAILS
)

SELECT * FROM CTE WHERE ROLL_NO=1
```

ROLL_NO	EMPID	EMPNAME	EMPSAL	EMPDEPT
1	101	SAMBA	45000	SE
1	106	BINDHU KUMAR	54000	SSE
1	105	NARESH	52000	TL

IDENTITY

Identity is an auto incremental column based on the "seed and incremental value".

Example

```
CREATE TABLE IDENTITY_TEST
(
ID INT IDENTITY (1, 1),
NAME VARCHAR (50),
SALARY INT,
)

INSERT INTO IDENTITY_TEST VALUES ('A', 45000), ('B', 35000),
('C', 43000), ('D', 25000)
```

The above example SEED for the identity is 1
And incremental value is also 1

ID	NAME	SALARY
1	A	45000
2	B	35000
3	C	43000
4	D	25000

For example identity (10, 5) output like as Below table

ID	NAME	SALARY
10	A	45000
15	B	35000
20	C	43000
25	D	25000

HOW TO CREATE IDENTITY ON EXISTING COLUMN

- Go to table right click on design
- Then it will display the list of the columns
- Select required column which you want create identity on it.
- Then below column level properties will be displayed.
- Go to the properties identity specifications by default it will be 'NO'.
- Turn it into 'YES'
- Then automatically identity will be enable
- And default it in to 1,1

COMPUTED COLUMN

Computed column will be the auto updated column we cannot provide any data to the provided column, it will update automatically based on the provided formula for that column.

HOW TO ADD A COLUMN WITH COMPUTED

```
ALTER TABLE EMPSALDETAILS ADD UPDATED_SAL AS (EMPSAL*0.1) +EMPSAL
```

EMPID	EMPNAME	EMPSAL	EMPDEPT	UPDATED_SAL
100	CHANBASHA	35000	SE	38500
101	SAMBA	45000	SE	49500
102	PARSHI	38000	SSE	41800
103	KARTHIK	42000	SSE	46200
104	RAJU	45000	TL	49500
105	NARESH	52000	TL	57200
106	BINDHU KUMAR	54000	SSE	59400

HOW TO ALTER THE EXISTING COLUMN AS COMPUTED

- Go to table
- Select design
- Select column below properties
- Computed column specifications expand it.
- Select the formula and provide the required formula in it.

VIEWS

- SIMPLE VIEW
- COMPLEX VIEW
- VIEW WITH SCHEMABINDING
- VIEW WITH ENCRYPTION
- INDEXED VIEW

View is a mirror image of a table, it will not stored the data physically whenever we have selecting this view, it will get the data from physical and display it.

It is a select statement to fetch the data from the required table for viewing purpose.

Advantages of views

- We can reduce the database size.

- No need to write complex queries every time we can store the code in side of the view and we can execute.
- We can provide security to the physical table by creating these views.

SIMPLE VIEW: If we are creating a view on single table so that type of views are called as SIMPLE VIEWs.

Example

```
CREATE VIEW V_EXAMPLE
AS

SELECT MAX (EMPSAL) AS MAX_SAL, EMPDEPT
FROM EMPSALDETAILS GROUP BY EMPDEPT

SELECT * FROM V_EXAMPLE
```

Simple view can be available to do DML (INSERT, UPDATE, and DELETE) operations, which will affect the physical table automatically.

COMPLEX VIEW: If we are creating a view by using more than one table then that type of views are called as COMPLEX VIEWs.

Example

```
CREATE VIEW V_COMPLEX
AS
SELECT
A.EID,
A.ENAME,
A.ESALARY,
B.EMPADD1,
B.EMPADD2
FROM EMPDETAILS A JOIN EMPADDRESS B
ON A.EID = B.EID
WHERE B.EMPADD2 IN ('MUM','DLI')

SELECT * FROM V_COMPLEX
```

- If one table is updating in the complex view update is possible.
- If the multiple tables are affecting it is not possible.
- If one table deleting in the complex view is possible but multiple tables are not possible
- Hence complex view is not available for DML operations because it will affect the more base tables.

VIEW WITH SCHEMA BINDING: If we create a view with schema binding option the structure cannot be modified for the source table.

Example

```
CREATE VIEW V_EXAMPLE1 WITH SCHEMABINDING
AS
SELECT EMPID, EMPNAME, EMPSAL FROM DBO.EMPSALDETAILS

SELECT * FROM V_EXAMPLE1

ALTER TABLE EMPSALDETAILS ALTER COLUMN EMPID VARCHAR (50) Fail this command
```

Whenever we are creating the view with schema binding then we have to remind the following things

- Provide the keyword with schema binding after the view name.
- Avoid the * in select query and provide the list of the columns which we required.
- Provide the schema DBO. Before mentioned in the table.

VIEW WITH ENCRYPTION: If create a view with encryption option then we can't see the code which is returns inside of the view.

Example

```
CREATE VIEW V_EXAMPLE2 WITH ENCRYPTION
AS
SELECT * FROM EMPSALDETAILS

SELECT * FROM V_EXAMPLE2
```

INDEXED VIEW: If we create a index on view then that type of views are called as indexed views in this case view can store the data same as tables with the help of index.

We have to create only unique cluster index only these types' views.

While creating index views we have to remind the following cases.

- View should be schemabinded
- Mentioned the list of the columns instead of stocks.
- Provide schema name also preceding the table name.
- Create a unique clustered index for non duplicated column as following.
- Constraints cannot be applicable for views.

View with schemabind index

```
CREATE VIEW V_index WITH SCHEMABINDING
AS
SELECT EMPID, EMPNAME, EMPSAL FROM DBO.EMPSALDETAILS

SELECT * FROM V_index
```

Unique clustered index

```
CREATE UNIQUE CLUSTERED INDEX IX_EMPID
ON V_INDEX (EMPID)
```

SUBQUERYS

Sub query is the query with in a query

- First it will execute the inner query and get the output, and pass it into the outer query.

Example

```
select * from empsaldetails
where empid in (select empid from empsal)
```

Example

```
SELECT
(SELECT TOP 1 EMPID FROM EMPSAL)AS EMPID, EMPNAME
 FROM EMPSALDETAILS
```

Correlated sub query

```
SELECT * FROM EMPSALDETAILS WHERE EMPID =
(SELECT TOP 1 EMPID FROM EMPSAL ORDER BY EMPID DESC)
```

LIKE OPERATOR

START WITH ‘R’

```
SELECT * FROM EMPSALDETAILS WHERE EMPNAME LIKE 'R%'
```

END WITH ‘S’

```
SELECT * FROM EMPSALDETAILS WHERE EMPNAME LIKE '%S'
```

HAVING WORD ‘AR’ IN NAME

```
SELECT * FROM EMPSALDETAILS WHERE EMPNAME LIKE '%AR%'
```

NAME WITH ALL ALPHABETS

```
SELECT * FROM EMPSALDETAILS WHERE EMPNAME LIKE '% [A-Z] %'
```

NAME WITH ALL NO'S

```
SELECT * FROM EMPSALDETAILS WHERE EMPNAME LIKE '% [0-9] %'
```

SCALAR VARIABLE: Scalar variable will store single value in it. And it will be available for further operations.

To specify the scalar variable we have to declare with corresponding data types to store appropriate values.

Example

```
DECLARE @STRING VARCHAR (50) = 'HELLO WORLD'
PRINT @STRING

OR

DECLARE @STRING VARCHAR (50)
SET @STRING = 'HELLO WORLD'
SELECT @STRING
```

SCALAR VARIABLE FOR KEYWORD SELECT

```
DECLARE @STRING VARCHAR (50),@ID INT = 103
SELECT @STRING = EMPNAME FROM EMPSALDETAILS WHERE EMPID = @ID
SELECT @STRING
```

TABLE VARIABLE: If we are storing multiple values in a single variable that type of variable are called as table variable.

In this we can store complete result set in table format.

HOW TO DECLARE TABLE VARIABLE

```
DECLARE @TMP TABLE
(
EMPID INT,
UPDATEDSAL INT
)
```

In the above example @TMP is table variable which can store the result in two columns EMPID, UPDATEDSAL

INSERTING DATA INTO TMP VARIABLE

```
INSERT INTO @TMP
SELECT EMPID, (EMPSAL*0.5) +EMPSAL AS UPDATEDSAL FROM EMPSALDETAILS
```

SELECTING DATA FROM TABLE VARIABLE

```
SELECT * FROM @TMP
```

HOW TO JOIN TABLE VARIABLE WITH PHYSICAL TABLE

```
UPDATE EMPSALDETAILS SET EMPSAL = T.UPDATEDSAL FROM @TMP T
JOIN
EMPSALDETAILS E
ON T.EMPID=E.EMPID

SELECT * FROM EMPSALDETAILS
```

DIFFERENCE BETWEEN SCALAR VARIABLE AND TABLE VARIABLE

SCALAR VARIABLE

- It can store only single value
- We will declare the data types for this scalar variable (varchar(),int,etc)
- We can directly assign a value into the scalar variable

TABLE VARIABLE

- It can store entire table
- We will declare the data types as table to store the table result.
- We have to provide the insert statement to insert the data

TEMPARORY TABLES

Temp tables will be use full to store the data physically in table which are created under TEMPDB database.

These are the session tables and will be disappear automatically whenever the session got closed.
We have following types of temp tables in SQL SERVER.

1) Normal temp table
2) Global temp table

NORMAL TEMP TABLES: If we are creating the table with # symbol that type of table are called as NORMAL TEMP TABLE.

These normal temp tables will be valid only up to the created window. If we close the created window then automatically the table will be deleted in the TEMPDB.

HOW TO CREATE NORMAL TEMP TABLE

```
CREATE TABLE #CUST
(
```

```
CUSTID INT,
CUSTNAME VARCHAR (50),
CUSTBAL INT
)
```

GLOBAL TEMP TABLE: If we are creating the table with ## symbol then that type of tables are called as GLOBAL TEMP TABLE.

These tables also will be created under TEMPDB and these will valid in multiple windows until the created window got closed.

Once we close the created window automatically global temp table will be deleted from TEMPDB database.

```
CREATE TABLE ##CUST
(
CUSTID INT,
CUSTNAME VARCHAR (50),
CUSTBAL INT
)
```

DIFFERENCES BETWEEN TABLE VARIABLE AND TEMPARORY TABLE

TABLE VARIABLE

- It will store data logically even we are executing the deleted statement,
- Every time we have to run entire block to get the result and it will be available when we run the declare statement.
- We can’t see the table variable anywhere in the bases

TEMPARORY TABLE

- It will store the data physically in the TEMPDB database.
- These are the session tables and we can run any no. of times will be the created window is open.
- We can see the TEMP table in the TEMPDB database after creating the temp table.

T-SQL

T-SQL is the TRANSACTION SQL and we are having the following concept in this.

- Iterative
- Procedures
- Functions
- Indexes
- Triggers
- Cursors

- **ITERATIVES**

- **IF:** If condition will check the given condition and the condition is true, then it will execute the block which we mentioned immediate to that condition, otherwise it will fall into else case and execute the statement provide in it.

Example

```
DECLARE @I INT = 24
IF (@I%2=0)
BEGIN
PRINT 'EVEN NUMBER'
END
ELSE
PRINT 'ODD NUMBER'
```

OUTPUT IS EVEN NUMBER

Example

```
DECLARE @DATE DATE = '2015-02-19'
IF (@DATE=EOMONTH ('2015-02-19'))
BEGIN
PRINT 'END OF THE MONTH'
END
ELSE
PRINT 'MIDDLE OF THE MONTH'
```

Example

```
DECLARE @DATE DATE = '2015-02-19'------given date
DECLARE @CDATE DATE = DATEADD (MM,1,@DATE)----ADDING ONE MONTH
DECLARE @EODATE DATE = DATEADD (DD,-DAY (@CDATE), @CDATE)
IF (@DATE=@EODATE)
BEGIN
PRINT 'END OF THE MONTH'
END
ELSE
PRINT 'MIDDLE OF THE MONTH'
```

Example

```
DECLARE @NO INT = 145
IF (@NO>100)
BEGIN
PRINT 'NO IS LARGE'
```

```
END
ELSE
IF (@NO>10)
BEGIN
PRINT 'MEDIUM VALUE'
END
ELSE
PRINT 'SMALL NUMBER'
```

WHILE: while condition will check the given condition. satisfy it will go into the loop and execute the statement until the condition is falls.

Example

```
DECLARE @I INT =1
DECLARE @J INT = 10
WHILE (@I <= @J)
BEGIN
PRINT @I
SET @I = @I+1
END
```

Example PRINT EVEN NUMBERS BETWEEN START AND END VALUES

```
DECLARE @I INT =1
DECLARE @J INT = 10
WHILE (@I <= @J)
BEGIN
IF (@I%2=0)
BEGIN
PRINT @I
END
SET @I = @I+1
END
```

PROCEDURES Procedure is a group of SQL statement and it will store as database objects, which we can use multiple times whenever we required.

We have following types of procedures.

1) Simple procedures
2) Procedures with input parameters
3) Procedure with input & output parameters
4) Procedures with default values.

- **SIMPLE PROCEDURES:** If we are using the group of SQL statements without any input required from the user those procedures are simple procedures.

Syntax

```
CREATE PROCEDURE <PROCEDURE NAME>
AS
BEGIN

<SQL STATEMENTS>

END
```

Example

```
CREATE PROCEDURE USP_EXAMPLE1
AS
BEGIN

UPDATE EMP2 SET EMPADD2 = 'BNG'
WHERE EMPADD2 IS NULL OR EMPADD2 =' '

UPDATE EMP2 SET DESIGNATION = 'ASE'
WHERE JOINDATE = CAST (GETDATE () AS DATE)


END

EXEC USP_EXAMPLE1
```

PROCEDURE WITH INPUT PARAMETERS: It we pass any value while executing the procedure then that value will be passed to the SQL statements and get the details based on that value as a result.

Syntax

```
CREATE PROCEDURE USP_EXAMPLE2 <PARAMETERS>
AS
BEGIN
 <SQL STATEMENTS>
END
```

Example

```
CREATE PROCEDURE USP_EXAMPLE2 (@LOC VARCHAR(50))
AS
BEGIN
SELECT * FROM EMP2
WHERE EMPADD2 = @LOC

END

EXEC USP_EXAMPLE2 UP
```

Example

```
CREATE PROCEDURE USP_EXAMPLE2 (@LOC VARCHAR (50), @ID INT)
AS
BEGIN
SELECT * FROM EMP2
WHERE EMPADD2 = @LOC AND EMPID=@ID

END

EXEC USP_EXAMPLE2 PUNE, 4
```

PROCEDURE WITH DEFAULT VALUES: If we have any parameter for the procedure and we are not providing any value while executing it, then it automatically throw an error PARAMETER IS EXPECTED.

To avoid this situation we can provide default values to parameter to execute and automatically even if we are not providing any value.

Example

```
ALTER PROCEDURE [dbo].[USP_EXAMPLE2] (@LOC VARCHAR(50)=NULL)
AS
BEGIN
SELECT * FROM EMP2
WHERE EMPADD2 = @LOC

END
EXEC [USP_EXAMPLE2] UP
```

```
1      puNE bNg      UP
```

PROCEDURE WITH INPUT AND OUTPUT PARAMETERS:

```
ALTER PROCEDURE [dbo].[USP_EXAMPLE2] (@ID INT,@SAL INT OUTPUT)
AS
BEGIN
SELECT @SAL= EMPSAL FROM EMPSALDETAILS
WHERE EMPID = @ID

END

DECLARE @IP INT
EXEC [USP_EXAMPLE2] @ID=104,@SAL=@IP OUTPUT
SELECT @IP
```

PERFORMANCE TUNING THE PROCEDURES

- Check the join columns or having indexes in database or null
- If we don't have created the proper indexes on the key columns
- Avoid functions in the procedures why because it will give slow performance while executing
- Instead of using function in procedures, use the procedure with in a procedure.
- Avoid sub query which will execute inner query first and give the result to the other query so it takes time.
- It is able to do please create the temp table or table variable to store the required data for test processing.
- Do OFF update statistics everywhere for better performance.
- Turn off set no count
- Try to avoid is null, cast, convert, functions, if it is required then only required.

- Don’t create the procedure with SP_ if it will check for the all system procedures as well use USP to avoid it.
- Try to avoid cursors and triggers on database and tables.

FUNCTIONS

1) SCALAR FUNCTION
2) TABLE VALUED FUNCTION

Function is as similar as store procedure and it is also group of SQL statements but the main different is function must return a value always immediate to the execution.

We have following types of functions

- Scalar valued function
- Table valued function

SCALAR VALUED FUNCTION: If a function is returning single value then that type of function are called as scalar valued function

Syntax

```
CREATE FUNCTION <FUNCTION NAME> ()
RETURNS <DATATYPE>
AS
BEGIN
<SQL STATEMENTS>
RETURN
END
```

Example WITH OUT PARAMETER

```
CREATE FUNCTION FN_EXAMPLE ()
RETURNS INT
AS
BEGIN
DECLARE @SAL INT
SELECT @SAL= EMPSAL FROM EMPSALDETAILS
WHERE EMPID = 105
RETURN @SAL
END

SELECT DBO.FN_EXAMPLE ()
```

Example with parameter

```
ALTER FUNCTION FN_EXAMPLE (@ID INT)
RETURNS INT
AS
BEGIN
DECLARE @SAL INT
SELECT @SAL= EMPSAL FROM EMPSALDETAILS
WHERE EMPID = @ID
```

```
RETURN @SAL
END

SELECT DBO.FN_EXAMPLE (100)
```

TABLE VALUED FUNCTIONS

Syntax

```
CREATE FUNCTION <FN NAME> ()
RETURNS <DECLATE TABLE>
AS
BGIN
<SQL STATEMENTS>
RETURN
END
```

If the function is returns the table of values then that type functions are called as table valued functions.

Example

```
ALTER FUNCTION FN_EXAMPLE2 (@DEPTNO VARCHAR (50))
RETURNS @TMP TABLE
(ID INT, NAME VARCHAR (50), SAL INT, DEPT VARCHAR (50))
AS
BEGIN
INSERT INTO @TMP
SELECT EMPID, EMPNAME, EMPSAL, EMPDEPT FROM EMPSALDETAILS
WHERE EMPDEPT = @DEPTNO
RETURN
END

SELECT * FROM DBO.FN_EXAMPLE2 ('SE')
```

DIFFERENCE BETWEEN PROCEDURES AND FUNCTIONS

PROCEDURES:

- Procedure may or may not return the value
- We can call function in procedure
- It will not do record by record
- It will not close with parenthesis () until we use the parenthesis
- Procedure will execute with exec statement

FUNCTIONS

- Function must return the value
- We can’t call procedure in function
- Function will perform record by record process
- Every function should close parenthesis ().
- Function will execute with select statement.

INDEXES

Indexes is like as same as index in the book. It will provide the reference to the select query for retrieve the data very fastly.

We have two types of indexes

1) Clustered index
2) Non clustered index

CLUSTERED INDEX:

- Whenever we are creating primary key then automatically clustered index will be generated on the specified column
- We can create only one clustered index per a table
- It will store the actual values in the index pages to get the data very fast.
- It will use B TREE to search the values in the pages
- When compare to the heap table we will get data in a faster way while creating the index.
- We can create clustered index on multiple column also even we don't have primary key.

Syntax

```
CREATE CLUSTERED INDEX <INDEX NAME>
ON <TABLE NAME> (COLUMN NAME)
```

```
------WITH DUPLICATES----

CREATE CLUSTERED INDEX IX_EMPID
ON EMPSALDEPT (EMPID)

------WITH OUT DUPLICATES----

CREATE UNIQUE CLUSTERED INDEX IX_EMPID
ON EMPSALDEPT (EMPID)
```

Example

```
CREATE TABLE CUST
(
CUSTID INT PRIMARY KEY, ------CRETAE CLUSTER INDEX AUTOMATICALYY
CUSTNAME VARCHAR (50),
CUSTBAL INT
)
```

NON- CLUSTERED INDEX

- Whenever we are creating unique constraint, then automatically non clustered index will create.
- We can create non-clustered index up to 249 in 2005
  999 in 2008, 2012

- It will store the address of the values in the B TREE and get the actual values using the address.
- It will use the B TREE to search the value like pages

- When compare to the heap table we will get data in the faster way to retrieve.

Syntax

```
CREATE NON CLUSTERED INDEX <NON CLUSTERED NAME>
ON <TABLE NAME> (COLUMN NAME)
```

FILL FACTOR: Fill factor is used to specify the range of the base to fill the data based on index, it will tell the percentage of page to be fill to store.

The Microsoft recommended percentage is 70%.

INDEX DISADVANTAGES

- It occupies more space in the database to store the data in the index page.
- Whenever we are doing update, insert, in index will give very low performance to do the operation
- Whenever we are inserting the intermediate value for the index it will not automatically allocate the proper position in the index pages then performance will not be high, while retrieving the data from the table
- We can't create indexes as TEXT, NTEXT, IMAGE datatypes.

TRIGGERS Triggers are used to rise and event, whenever user doing the operation against the database for table.

We have following types of triggers DDL & DML

DDL TRIGGERS: DDL triggers are database triggers if we are performing any kind of operations like CREATE, ALTER, DROP against the database.

DDL TRIGGERS TO PRINT SUCCESS MESSAGE

```
CREATE TRIGGER TR_DDL
ON DATABASE FOR
CREATE_TABLE, ALTER_TABLE, DROP_TABLE

AS
BEGIN

PRINT 'YOU HAVE CREATED TABLE IN SAMPLE DATABASE'

END

TABLE CREATION

CREATE TABLE AAAA
(
ID INT
)
```

DDL TRIGGERS TO RESTRICT THE USER

```
CRETAE TRIGGER TR_DDL1
ON DATABASE FOR
CREATE_TABLE, ALTER_TABLE, DROP_TABLE

AS
BEGIN

PRINT 'YOU DONT HAVE PERMISSIONS TO DO DDL OPERATIONS '

ROLLBACK

END

TO CRAETE TABLE
CREATE TABLE BBBB
(
ID INT
)
```

OUTPUT

```
YOU DONT HAVE PERMISSIONS TO DO DDL OPERATIONS
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

FOR DROP TRIGGERS

```
DROP TRIGGER TR_DDL ON DATABASE
```

DML TRIGGERS: DML triggers are table level triggers it will restrict the user to do the table level operations.

TO UPDATE NAME PROPER ORDER

Example

```
CREATE TRIGGER TR_DDL
ON EMPSALDETAILS FOR INSERT
AS
BEGIN
UPDATE EMPSALDETAILS SET EMPNAME = UPPER (LEFT (EMPNAME, 1)) +LOWER
(SUBSTRING (EMPNAME, 2, LEN (EMPNAME)))
END

INSERT INTO EMPSALDETAILS VALUES (107,'SAMBASHIVUDU',85000,'SE')
```

TO RESTRICT FOR ROW DELETION IN TABLE

Example

```
CREATE TRIGGER TR_DDL1
ON EMPSALDETAILS FOR DELETE
AS
BEGIN
PRINT 'NO PERMISSION FOR DELETION'
ROLLBACK-------MANDATORY
END
```

```
DELETE FROM EMPSALDETAILS WHERE EMPNAME='SAMBASHIVUDU'
```

OUTPUT NO PERMISSION FOR DELETION

```
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

MULTIPLE TABLES INSERTING VALUES AT A TIME

```
CREATE TRIGGER TR_DDL2
ON EMPSALDETAILS FOR INSERT
AS
BEGIN
INSERT INTO COPYVIEW
SELECT * FROM EMPSALDETAILS
EXCEPT
SELECT * FROM COPYVIEW

END

INSERT INTO EMPSALDETAILS VALUES (107,'SAMBASHIVUDU', 85000,'SE')
```

CURSORS

cursors are used to do the row operations on table for each and every record, it is private memory area, which will be stored the result of SQL statements.

To create the cursors we have to follow the 5 steps.

1) DECLARE CURSORS
2) OPEN CURSORS
3) FETCH STATUS
4) CLOSE CURSORS
5) DE ALLOCATE CURSORS

DECLARE CURSOR: we can declare a cursor for select statement then it will fetch the data from those tables and insert in to cursors.

OPEN CURSORS: It will open cursor to read the data record by record in to the variables.

FETCH STATUS: Fetch statement is used to fetch the data as record by record in to the variables.

CLOSE CURSORS: once the operation is completed we have to the open cursor

DE ALLOCATE CURSOR: At last we have to destroy or de allocate private memory area which is occupied by the cursor.

Syntax

```
DECLARE <CURSOR NAME> CURSOR
FOR
<SELECT STATEMENTS>
```

FOR OPEN

```
OPEN <CURSOR NAME>
<DECLARE SELECT STATEMENTS OF VARIABLES>
```

FETCH STATUS

```
@@FETCH STATUS
<INSERT STATEMENTS>
FETCH NEXT FROM <STATEMENTS>
```

CLOSE CURSOR

```
CLOSE <CURSOR NAME>
```

DE ALLOCATE CURSOR

```
DE ALLOCATE <CURSOR NAME>
```

Example

```
DECLARE ABC CURSOR
FOR SELECT * FROM EMPSAL
OPEN ABC
DECLARE @DEMPID INT,@DEMPNAME VARCHAR (50), @DEMPSAL INT

FETCH NEXT FROM ABC
INTO @DEMPID, @DEMPNAME, @DEMPSAL

WHILE (@@FETCH_STATUS=0)
BEGIN
PRINT 'THE EMPLOYEE HAVING NUMBER' +
CAST (@DEMPID AS VARCHAR)+ 'IS HAVING NAME AS' + @DEMPNAME+'WITH HIS SALARY
AS' +
CAST (@DEMPSAL AS VARCHAR)

FETCH NEXT FROM ABC
INTO @DEMPID, @DEMPNAME, @DEMPSAL
END
```

```
CLOSE ABC
DEALLOCATE ABC
```

The end.