SQL Server

* **SQL Constraints:]**
* SQL Constraints are used to specify rules for the data in a table.
* Constraints are used to limit the type of data that can go into a table.

This ensures the accuracy and reliability of the data in the table. It there is any violation between the constraint and the data action, the action is aborted.

* Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

* NOT NULL – Ensures that a column cannot have a NULL value.
* UNIQUE- Ensures that all values in a column are different.
* PRIMARY KEY- A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table.
* FOREIGN KEY- Uniquely identifies a row/record in another table.
* CHECK- Ensures that all values in a column satisfies a specific condition.
* DEFAULT – Sets a default value for a column when no value is specified.
* INDEX – Use to create and retrieve data from the database very quickly.

**Query 1:**

--Create database query

create database Batch\_17028;

use Batch\_17028;

create table Student\_Bio\_Data(STDID int, STD\_NAME varchar(50),ROLL\_NO int, CLASS varchar(50),FATHER\_NAME varchar(50));

--View data from table

select \*from Student\_Bio\_Data;

**--insert data into table**

insert into Student\_Bio\_Data values(1,'Keshav',1,'BTech','Sunil Kumar Choudhary');

insert into Student\_Bio\_Data values(2,'Madhav',2,'HR','Sunil Kumar Choudhary');

insert into Student\_Bio\_Data values(3,'Pranav',3,'graphics','Murari Kumar Choudhary');

insert into Student\_Bio\_Data values(4,'Suraj',4,'7','Murari Kumar Choudhary');

insert into Student\_Bio\_Data values(5,'Pranay',5,'1','Rupesh Kumar Choudhary');

insert into Student\_Bio\_Data (STDID,STD\_NAME) values(6,'Kukki');

QUERY 2:

use Batch\_17028;

create table Employee(Empid int unique not null, EmpName varchar(50) not null, EmpDpt varchar(50) not null, Salary varchar(50));

select \* from Employee;

insert into Employee values(1,'Keshav','Developer','33000');

insert into Employee values(2,'Madhav','HR','23000');

insert into Employee values(3,'Anjali','Finance','30000');

-- delete one data from table

delete from Employee where EmpName='Anjali';

--delete all data from table

truncate table employee;

select \*from employee;

-- Update table employee

update Employee set Salary=25000 where EmpId=2;

update Employee set EmpName='Rachna' where EmpId=3;

update Employee set EmpDpt='Software Developer' where EmpId=1;

-- delete table

drop table Employee;

**QUERY 3:**

create table Voter\_list(Voter\_id int primary key,Voter\_Name varchar(50) not null, Voter\_Age int not null default(18));

select \* from Voter\_list;

insert into Voter\_list values(1,'Keshav Kumar',26);

insert into Voter\_list values(2,'Madhav Kumar',27);

insert into Voter\_list (Voter\_id,Voter\_Name) values(3,'Pranav Kumar');

insert into Voter\_list values(4,'Anjali',26);

insert into Voter\_list values(5,'Govind',33);

-- Select statement for view data

select voter\_name,voter\_age from Voter\_list;

select \* from Voter\_list where Voter\_id=4 or Voter\_id=2;

select voter\_name,voter\_age from Voter\_list where Voter\_id=2;

select \* from Voter\_list order by Voter\_Name desc;

select \* from Voter\_list order by Voter\_Name;

select Voter\_name from Voter\_list where Voter\_id=3 or Voter\_id=4;

--delete voter list table

drop table Voter\_list;

**QUERY 4:**

CUSTOMER TABLE:

CUSTOMER ID CUSTOMER NAME ADDRESS CITY

1 Keshav Sector 7 Madanpur Gurgaun

2 Madhav Sherpur Chadni Chowk Samastripur

3 Anjali Sector 7 Madanpuri Gurgaun

4 Pranav Joyti Park Indor

5 Suraj Joyti Park Indor

ORDER TABLE

ORDER ID ORDER ITEM ORDER QUANTITY PRICE OF 1 CUSTOMER ID

111 Hardisk 2 500 3

222 Ram 1 300 1

333 Keyboard 3 400 4

444 mouse 2 200 2

555 Speaker 1 3000 2

666 USB 2 1000 5

* Primary Key cannot be null or duplicate.
* Foreign key can be null or duplicate.
* We cannot insert in order table with id that is not present in Customer table.
* We can not delete from Customer table if id is present in Order table.

**STEPS TO COVER:**

🡪**SYNTAX of creating foreign key while creating a table.**

* C\_ID DATA\_TYPE FOREIGN KEY REFERENCES CUSTOMER(C\_ID)

🡪**Dropping foreign key constraint with alter statement.**

* SYNTAX: ALTER TABLE TABLE\_NAME DROP CONSTRAINT CONSTRAINT\_NAME.

🡪**Creating foreign key in existing table with alter statement.**

* SYNTAX: ALTER TABLE TABLE\_NAME ADD FOREIGN KEY (C\_ID) REFERENCVES CUSTOMER(C\_ID).

**Query:**

use Batch\_17028;

create table Customer\_tbl

(

C\_id int primary key,

C\_Name varchar(50),

C\_Address varchar(max),

City varchar(50)

);

select \* from Customer\_tbl;

insert into Customer\_tbl values(1,'Keshav','Sector 7 Gurgaun','Gurgaun');

insert into Customer\_tbl values(2,'Madhav','Sherpur Chadni Chauwk','Samastipur');

insert into Customer\_tbl values(3,'Pranav','Jyoti Park Indore','Indore');

insert into Customer\_tbl values(4,'Suraj','Jyoti Park Indore','Indore');

insert into Customer\_tbl values(5,'Anjali','Sector 7 Gurgaun','Gurgaun');

create table [order]

(

Ord\_Id int primary key,

Item varchar(50),

Quantity int,

Price\_Of\_1 int,

C\_id int foreign key references Customer\_tbl(C\_id)

);

--order is keyword in SQL Server thats why we use square bracket

select \* from [order];

insert into [order] values(111,'HardDisk',2,500,3);

insert into [order] values(222,'Ram',1,300,1);

insert into [order] values(333,'Keyboard',3,400,4);

insert into [order] values(444,'Mouse',2,300,2);

insert into [order] values(555,'USB',2,1000,5);

insert into [order] values(666,'Speaker',1,3000,2);

insert into [order] values(777,'USB',1,1000,6);

delete from [order] where Ord\_Id=777;

select \*from Customer\_tbl;

select \*from [order];

--delete foreign key

alter table [order] drop constraint FK\_\_order\_\_C\_id\_\_398D8EEE;

--add foreign key

alter table [order] add foreign key (C\_id) references Customer\_tbl(C\_id);

**Cascading Referential Integrity/ Cascading Foreign Key in SQL Server:**

**What are Cascading Referential Integrity Constraints in SQL Server?**

* The Cascading Referential Integrity Constraints in SQL Server are the foreign key constraints that tell SQL Server to perform certain actions whenever a user attempts to delete or update a primary key to which an existing foreign keys point.

**What are the Actions Performed by SQL Server?**

* In order to tell the SQL Server what actions to perform whenever a user trying to delete or update a primary key value to which existing foreign key points, we are provided with the following options while working with Cascading Referential Integrity Constraints.

1. NO ACTION (By Default)
2. CASCADE
3. SET DEFAULT
4. SET NULL

* When we add a foreign key to a column then we have to add one of these action with foreign key.

**CASCADE:**

* If a user tries to delete the statement(s) which will affect the rows in the foreign key table, then those rows will be deleted when the primary key record is deleted.
* Similarly, if an update statement affects rows in the foreign key table, then those rows will be updated with the value from the primary key record after it has been updated.

**QUERY:**

use Batch\_17028;

create table Customer\_tbl

(

C\_id int primary key,

C\_Name varchar(50),

C\_Address varchar(max),

City varchar(50)

);

select \* from Customer\_tbl;

insert into Customer\_tbl values(1,'Keshav','Sector 7 Gurgaun','Gurgaun');

insert into Customer\_tbl values(2,'Madhav','Sherpur Chadni Chauwk','Samastipur');

insert into Customer\_tbl values(3,'Pranav','Jyoti Park Indore','Indore');

insert into Customer\_tbl values(4,'Suraj','Jyoti Park Indore','Indore');

insert into Customer\_tbl values(5,'Anjali','Sector 7 Gurgaun','Gurgaun');

create table [order]

(

Ord\_Id int primary key,

Item varchar(50),

Quantity int,

Price\_Of\_1 int,

C\_id int foreign key references Customer\_tbl(C\_id)

on delete cascade

on update cascade

);

--order is keyword in SQL Server thats why we use square bracket

select \* from [order];

insert into [order] values(111,'HardDisk',2,500,3);

insert into [order] values(222,'Ram',1,300,1);

insert into [order] values(333,'Keyboard',3,400,4);

insert into [order] values(444,'Mouse',2,300,2);

insert into [order] values(555,'USB',2,1000,5);

insert into [order] values(666,'Speaker',1,3000,2);

insert into [order] values(777,'USB',1,1000,6);

delete from [order] where Ord\_Id=777;

select \*from Customer\_tbl;

select \*from [order];

--delete table if we use foreign key then first we need to delete [order] then Customer\_tbl

drop table Customer\_tbl;

drop table [order];

-- These two line will be terminated just because of foreign key (NO ACTION )

delete from Customer\_tbl where C\_id=2;

update Customer\_tbl set C\_id=7 where C\_id=1;

alter table [order] drop constraint FK\_\_order\_\_C\_id\_\_398D8EEE;

alter table [order] add foreign key (C\_id) references Customer\_tbl(C\_id);

**SET NULL:**

* If a user tries to delete or update statement(s) that will affect rows in the foreign key table, then those values will be set to NULL when the primary key record is deleted or updated in the Primary key table.
* The important thing that we need to keep in mind that the foreign key columns affected must allow NULL values.

**SET DEFAULT:**

* If a delete or update statement affects rows in a foreign key table, then all rows containing those foreign keys are set to the default value.
* All foreign key columns in the related table must have default constraints defined on them.

**Important Note:**

* You cannot add ON DELETE CASCADE to an already existing constraint.
* You will have to drop and re-create the constraint.
* The documentation shows that the MODIFY CONSTRAINT clause can only modeify the state of a constraint ( i.e: ENABLE/DISABLED).

**Alter Foreign Key Constraint:**

* ALTER TABLE TABLENAME drop CONSTRAINT FK\_CONSTRAINTNAME.
* ALTER TABLE TABLENAME ADD CONSTRAINT FK\_CONSTRAINT FOREIGN KEY (FID)

REFERENCES OTHERTABLE (ID)

ON DELETE CASCADE ON UPDATE NO ACTION.

**Use of ALTER:**

* ALTER table statement is used to add, delete, or modify columns in an existing table.
* The ALTER table statement is also used to add and drop various constraints on an existing table.

use Batch\_17028;

--update database name using alter

alter database Batch\_17028 modify name = New\_Batch\_17028;

--update database name using store procidure

execute sp\_renamedb 'New\_Batch\_17028', 'Batch\_17028';

--update table name using store procidure

execute sp\_rename 'voter\_list','New\_voter\_list';

--want to add new column in Voter list - Voter\_City

alter table New\_Voter\_list add Voter\_City varchar(50);

select \*from New\_Voter\_list;

--once we execute insert query we can not execute. so, for inserting data into

--Voter\_City we need to execute update query like below

update New\_Voter\_list set Voter\_City='Hydrabad' where Voter\_id=1;

--for droping Voter\_city

alter table New\_Voter\_list drop column Voter\_City;

alter table New\_Voter\_list alter column Voter\_Name nvarchar(50)

**ALTER WITH SQL CONSTRAINTS**

1. NOT NULL
2. UNIQUE
3. PRIMARY KEY
4. FOREIGN KEY
5. CHECK
6. DEFAULT

**QUERY:**

create table Voter\_Table

(

Voter\_Id int,

Voter\_Name varchar(50),

Voter\_Age int

);

select \* from Voter\_Table;

--modify Voter\_Table:modify to Voter\_Name not null

alter table Voter\_Table alter column Voter\_Name varchar(50) not null

alter table Voter\_Table alter column Voter\_Name varchar(50) null

insert into Voter\_Table ( Voter\_Id,Voter\_Age) values(1,28)

--Unique Voter\_Id

alter table Voter\_Table add unique(Voter\_Id)

insert into Voter\_Table values(2,'Keshav',28)

--drop this unique constraint

alter table Voter\_Table drop constraint [UQ\_\_Voter\_Ta\_\_E2882764EA70856C]

--make primary key Voter\_Id but this volter id is null type so for that purpose make it not null

alter table Voter\_Table add primary key(Voter\_Id);

alter table Voter\_Table alter column Voter\_Id int not null

--drop this primary key

alter table Voter\_Table drop constraint [PK\_\_Voter\_Ta\_\_E288276590A3A9FA]

create table Voter\_City

(

C\_Id int,

C\_Name varchar(50),

Voter\_Id int

);

select \*from Voter\_Table;

select \* from Voter\_City;

--make foreign key

alter table Voter\_City add foreign key (Voter\_Id)

references Voter\_Table(Voter\_Id);

--drop foreign key

alter table Voter\_City drop constraint [FK\_\_Voter\_Cit\_\_Voter\_\_5FB337D6]

--Check on Voter Table

alter table Voter\_Table add check(Voter\_Age>=18);

insert into Voter\_Table values(3,'Madhav',18);

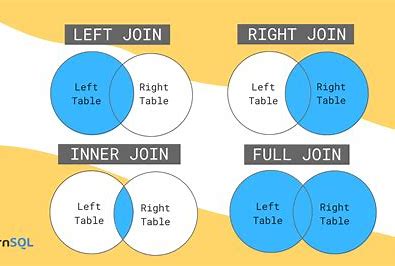
alter table Voter\_Table drop constraint [CK\_\_Voter\_Tab\_\_Voter\_\_60A75C0F]

drop table Voter\_Table;

**SQL JOINS:**

* SQL JOINS is used to combine two or more table data in resultset.

1. **INNER JOIN**
2. **LEFT JOIN**
3. **RIGHT JOIN**
4. **FULL OUTER JOIN**



**INNER JOINING Query:**

* Inner Join is used for fetching the records that are common in both tables.

create table Employee\_Tbl

(

EmpId int unique not null,

Emp\_Name varchar(50) not null,

Email varchar(50) not null,

Designation varchar(50) not null

);

insert into Employee\_Tbl values(11,'Keshav','Keshav@gmail.com','PA');

insert into Employee\_Tbl values(12,'Madhav','Madhav@gmail.com','P');

select \* from Employee\_Tbl;

create table Department

(

Dpt\_Id int unique not null,

Dpt\_Name varchar(50) not null,

Dpt\_Salary varchar(50) not null,

Emp\_Id int unique not null

);

insert into Department values(111,'Administration','33000',12);

insert into Department values(222,'IT','30000',11);

select \* from Employee\_Tbl;

select \*from Department;

--inner join

select \* from Employee\_Tbl as A

inner join Department as B

on A.EmpId=B.Emp\_Id;

select A.Emp\_Name,A.Designation,B.Dpt\_Name,B.Dpt\_Salary from Employee\_Tbl as A

inner join Department as B

on A.EmpId=B.Emp\_Id;

**LEFT JOINS:**

* Left Join is used to fetch the records that are common in the both tables and available in the left side table.

--LEFT JOIN

select A.Emp\_Name,A.Designation,B.Dpt\_Name,B.Dpt\_Salary from Employee\_Tbl as A

left join Department as B

on A.EmpId = B.Emp\_Id;

**RIGHT JOINS:**

* Right Join is used when we want to retrieve the records common in the both tables and available in the right side table.

--RIGHT JOIN

select A.Emp\_Name,A.Designation,B.Dpt\_Name,B.Dpt\_Salary from Employee\_Tbl as A

right join Department as B

on A.EmpId=B.Emp\_Id;

**FULL OUTER JOINS:**

* Full Join is used to fetch all the records of both tables.

--FULL JOIN

select A.Emp\_Name,A.Designation,B.Dpt\_Name,B.Dpt\_Salary from Employee\_Tbl as A

full outer join Department as B

on A.EmpId=B.Emp\_Id;

**SELF JOIN IN SQL SERVER:**

* A self-join is a regular join, but the table is joined with itself.

BEFORE SELF JOIN:

EMP ID EMP NAME MANAGER ID

1 Keshav 4

2 Madhav 4

3 Parnav 5

4 Pragyan 6

5 Vivek 1

6 Anand 1

AFTER SELF JOIN:

EMPLOYEE MANAGER

Keshav Pragyan

Madhav Pragyan

Pranav Vivek

Pragyan Anand

Vivek Keshav

Anand Keshav

**QUERY FOR SELF JOIN:**

* There is no keyword for self-join. We can perform self-join through inner join.

create table Employee\_Manager (

Empid int primary key,

Emp\_Name varchar(50) not null,

Manager\_Id int not null

);

select \* from Employee\_Manager;

insert into Employee\_Manager values(1,'Keshav',4);

insert into Employee\_Manager values(2,'Madhav',4);

insert into Employee\_Manager values(3,'Pranav',5);

insert into Employee\_Manager values(4,'Pragyan',6);

insert into Employee\_Manager values(5,'Vivek',1);

insert into Employee\_Manager values(6,'Anand',1);

select A.Emp\_Name as Employee, B.Emp\_Name as Manager

from Employee\_Manager as A

inner join Employee\_Manager as B

on A.Manager\_Id=B.Empid;

select \*from Employee\_Manager;

1. Write a query to get the below result:

EmployeeId Name Managerid

 1 Atul 5

2 Prashant 5

3 Asif 1

 4 Abhay 5

5 Shalabh null

employeename managername

Atul Shalabh

QUERY:

create table Employee\_Details1

(

id int primary key,

Name varchar(50) not null,

Manager\_Id int

);

select \* from Employee\_Details1;

insert into Employee\_Details1 values(1,'Atul',5);

insert into Employee\_Details1 values(2,'Prashant',5);

insert into Employee\_Details1 values(3,'Asif',1);

insert into Employee\_Details1 values(4,'Abhay',5);

insert into Employee\_Details1 values(5,'Sulabh',null);

select A.Name as employeename, B.Name as managername

from Employee\_Details1 as A

inner join Employee\_Details1 as B

on A.Manager\_Id=B.id;

select top(1) A.Name as employeename, B.Name as managername

from Employee\_Details1 as A

inner join Employee\_Details1 as B

on A.Manager\_Id=B.id;

**IDENTITY/ AUTO INCREMENT:**

create table Teacher\_tbl

(

T\_id int primary key identity,

T\_Name varchar(50) not null,

T\_Qualification varchar(50) not null,

T\_Salary varchar(50) not null

);

select \* from Teacher\_tbl;

insert into Teacher\_tbl values('Keshav','BTECH','33000');

insert into Teacher\_tbl values('Madhav','Deploma','25000');

insert into Teacher\_tbl values('Pranav','Graphics','27000');

insert into Teacher\_tbl values('Suraj','BSC','30000');

--delete table

drop table Teacher\_tbl;

**The SQL UNION Operator**

The UNION operator is used to combine the result-set of two or more SELECT statements.

* Every SELECT statement within UNION must have the same number of columns.
* The columns must also have similar data types.
* The columns in every SELECT statement must also be in the same order.

UNION Syntax

SELECT *column\_name(s)* FROM *table1*  
UNION/  
SELECT *column\_name(s)* FROM *table2*.

### UNION ALL Syntax

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

SELECT column\_name(s) FROM table1  
UNION ALL  
SELECT column\_name(s) FROM table2;

**Note:** The column names in the result-set are usually equal to the column names in the first SELECT statement.

QUERY:

create table HockeyParticipants

(

id int primary key,

Name varchar(50) not null,

Email varchar(50) not null

);

select \* from HockeyParticipants;

insert into HockeyParticipants values(1,'Keshav','Keshav@gmail.com');

insert into HockeyParticipants values(2,'Madhav','Madhav@gmail.com');

insert into HockeyParticipants values(3,'Pranav','Pranav@gmail.com');

create table FootballParticipants

(

id int primary key,

Name varchar(50) not null,

Email varchar(50) not null

);

insert into FootballParticipants values(1,'abc','Keshav@gmail.com');

insert into FootballParticipants values(2,'xyz','Madhav@gmail.com');

insert into FootballParticipants values(3,'Pranav','Pranav@gmail.com');

--union

select \* from HockeyParticipants

union

select \* from FootballParticipants;

--union all

select \* from HockeyParticipants

union all

select \* from FootballParticipants;

**INTERSECT:**

The INTERSECT clause in SQL is used to combine two [SELECT](https://www.geeksforgeeks.org/sql-select-clause/) statements but the dataset returned by the INTERSECT statement will be the intersection of the data sets of the two SELECT statements.

select \* from HockeyParticipants

intersect

select \* from FootballParticipants;

Except:

The **EXCEPT** operator in SQL is used to retrieve all the unique records from the left operand (query), except the records that are present in the result set of the right operand (query).

select \* from HockeyParticipants

except

select \* from FootballParticipants;

**AGGREGATE FUNCTIONS:**

SQL SERVER aggregate functions perform a calculation on a set of values and return a single value.

SUM  
MAX

MIN

AVG

COUNT

QUERY:

create table Employee\_Details

(

id int primary key identity,

Name varchar(50) not null,

Gender varchar(50) not null,

Salary int not null,

City varchar(50) not null

);

insert into Employee\_Details values('Keshav','Male','33000','Gurgaun');

insert into Employee\_Details values('Madhav','Male','34000','Samastipur');

insert into Employee\_Details values('Pranav','Male','32000','Indore');

insert into Employee\_Details values('Vivek','Male','30000','Noida');

insert into Employee\_Details values('Anand','Male','38000','Gurgaun');

select \* from Employee\_Details;

**--Aggregate function**

select sum(Salary) as Total\_Salary from Employee\_Details;

select max(Salary) as Maximum\_Salary from Employee\_Details;

select min(Salary) as Minimum\_Salary from Employee\_Details;

select avg(Salary) as Average\_Salary from Employee\_Details;

select count(Salary) as Count\_Salary from Employee\_Details;

drop table Employee\_Details;

**GROUP BY COMMAND:**

* A GROUP BY statement is used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
* We can only select those columns which were present in group by command.
* We can use one or more than one column in a GROUP BY clause.

QUERY:

select City, sum(Salary) as [Total Salary According to Cities]

from Employee\_Details group by City;

select Gender, City, sum(Salary) as [Total Salary According to Cities]

from Employee\_Details group by Gender, City;

**HAVING CLAUSE:**

* Having clause is used to specify conditions like where clause.
* But having clause is used with group by command.
* We cannot use where clause with group by command.
* Where clause is used for filtering rows and having clause is used to filtering groups.
* Where clause does not work with aggregate functions like SUM, MIN, MAX, COUNT, AVG.
* We can use having clause with Aggregate functions.
* Where clause can be used before the group by clause but not after that.

QUERY:

select City, sum(Salary) as Total\_Salary

from Employee\_Details

group by City

having City in ('Gurgaun');

--We can not use where clause after group by like below so it will give error.

select City, sum(Salary) as Total\_Salary

from Employee\_Details

group by City

where City in ('Gurgaun');

--But we use where clause before group by clause

select City, sum(Salary) as Total\_Salary

from Employee\_Details

where City in ('Gurgaun')

group by City;

* We can use Where and having together in a single query.

select City, sum(Salary) as Total\_Salary

from Employee\_Details

where City in ('Gurgaun','Samastipur')

group by City

having sum(salary)>25000;

**THE SQL ORDER BY keyword:**

* The ORDER BY keyword is used to sort the result-set in ascending or descending order.
* The ORDER BY keyword sorts the records in ascending order by default.
* To sort the records in descending order, use the DESC keyword.

QUERY:

--By default ascending order

select \* from Employee\_Details

order by Name;

--descending order DESC

select \* from Employee\_Details

order by Name DESC;

Query :

create table Employee\_Details

(

id int primary key identity,

Name varchar(50) not null,

Gender varchar(50) not null,

Salary int not null,

City varchar(50) not null

);

insert into Employee\_Details values('Keshav','Male','33000','Gurgaun');

insert into Employee\_Details values('Madhav','Male','34000','Samastipur');

insert into Employee\_Details values('Pranav','Male','32000','Indore');

insert into Employee\_Details values('Vivek','Male','30000','Noida');

insert into Employee\_Details values('Anand','Male','38000','Gurgaun');

insert into Employee\_Details values('Keshav','Male','33000','Gurgaun');

insert into Employee\_Details values('Keshav','Male','33000','Gurgaun');

insert into Employee\_Details values('Keshav','Male','33000','Gurgaun');

insert into Employee\_Details values('Madhav','Male','34000','Samastipur');

insert into Employee\_Details values('Madhav','Male','34000','Samastipur');

insert into Employee\_Details values('Madhav','Male','34000','Samastipur');

select \* from Employee\_Details;

--Aggregate function

select sum(salary) as Total\_Salary from Employee\_Details;

select max(salary) as Maximu\_Salary from Employee\_Details;

select min(salary) as Minimum\_Salary from Employee\_Details;

select avg(salary) as Average\_Salary from Employee\_Details;

select count(salary) as Count\_Salary from Employee\_Details;

select City, sum(salary) as Total\_Salary from Employee\_Details

group by City having City in ('Gurgaun');

select City, sum(Salary) as [Total Salary] from Employee\_Details

group by City having City in ('Samastipur');

select Gender , City , sum(Salary) as [Total Salary According to Cities]

from Employee\_Details group by Gender, City;

select City, sum(Salary) as Total\_Salary

from Employee\_Details

where City in ('Gurgaun')

group by City;

select City,sum(Salary) as Total\_Salary

from Employee\_Details

group by City having City in ('Gurgaun');

select City, sum(Salary) as Total\_Salary

from Employee\_Details

where City in ('Gurgaun','Samastipur')

group by City

having sum(salary)>25000;

select City, sum(Salary) as Total\_Salary

from Employee\_Details

group by City having City in ('Gurgaun','Samastipur');

select \* from Employee\_Details

order by Name asc

select \* from Employee\_Details

order by Name desc

--write query to find departments where the total salary exceeds 35000

select id,Name, sum(Salary) As Total\_Salary

from Employee\_Details

group by id,Name having sum(Salary)>35000;

select top 1 Salary from

(

Select distinct top 1 Salary from Employee\_Details

order by Salary desc

)

Result

Order by Salary;

select \* from Employee\_Details

-- find Nth highest salary

select min(Salary) as salary from

(

select top 2 Salary from Employee\_Details

order by Salary desc

)

as TopSalaries

-- Highest Salary

select top 1 Salary from Employee\_Details

order by Salary desc

-- second highest salary

select min(Salary) as salary from (

select top 2 Salary from Employee\_Details

order by Salary desc

)

as secondHighestSalary

order by Salary

--duplicate

select A.Name, count(\*) as totalNoOfDuplicate from Employee\_Details as A

group by Name having Name='Keshav'

--find duplicate with where clause

select \* from Employee\_Details

where Name ='Keshav';

--find duplicate using group by

select E.Name,count(\*) as DuplicateCount from Employee\_Details as E

group by Name having count(\*)>1;

select E.Name,count(\*) as DuplicateCount from Employee\_Details as E

group by Name having Name='Keshav'

select E.Name,count(\*) as DuplicateCount from Employee\_Details as E

group by Name having Name='Madhav';

--delete duplicate

delete from Employee\_Details

where id not in(

select min(id) from Employee\_Details

group by Name

);

select \* from Employee\_Details;

**VIEWS IN SQL SERVER:**

* A view in SQL is just a saved query.
* A view can also be considered as a virtual table.
* INSERT WITH VIEWS
* UPDATE WITH VIEWS
* DELETE WITH VIEWS

🡪 Views can be used as a mechanism to implement row and column level security.

🡪SP\_HELPTEXT

🡪ALTER THE VIEW

🡪DROP THE VIEW

QUERY:

create table MyDepartment

(

id int primary key,

Dept\_Name varchar(50) not null

);

select \* from MyDepartment;

insert into MyDepartment values(1,'Accounts');

insert into MyDepartment values(2,'HR');

insert into MyDepartment values(3,'Administration');

insert into MyDepartment values(4,'Counselling');

create table MyEmployees

(

Emp\_Id int not null,

Emp\_Name varchar(50) not null,

Gender varchar(50) not null,

Salary varchar(50) not null,

City varchar(50) not null,

Dept\_Id int not null

);

select \* from MyEmployees;

insert into MyEmployees values(100,'Usman','Male','25000','Hydrabad',2);

insert into MyEmployees values(101,'Anas','Male','45000','Kashi',1);

insert into MyEmployees values(102,'Suman','female','35500','Gurgaun',3);

insert into MyEmployees values(103,'Ali','Male','45500','Hydrabad',4);

insert into MyEmployees values(104,'Aman','Male','46000','Patna',2);

insert into MyEmployees values(105,'Asman','Male','25500','Delhi',1);

select \*from MyDepartment;

select \*from MyEmployees;

--create view for Employee table only

create view vW\_Employee

as select Emp\_Id,Emp\_Name,Gender from MyEmployees;

select \* from vW\_Employee

--create view using employee table and department table

create view vW\_ForEmployees

as

select \* from MyEmployees as A

inner join MyDepartment as B

on B.Id=A.Dept\_Id;

select \* from vW\_ForEmployees;

create view vW\_ForEmployees1

as

select A.\*, B.Dept\_Name from MyEmployees as A

inner join MyDepartment as B

on B.Id=A.Dept\_Id;

create view vW\_ForEmployees2

as

select A.Emp\_Id,A.Emp\_Name,A.Gender,A.City,A.Dept\_Id, B.Dept\_Name

from MyEmployees as A

inner join MyDepartment as B

on B.Id=A.Dept\_Id;

--create view for HR department

create view vW\_ForEmployees3

as

select A.\*, B.Dept\_Name

from MyEmployees as A

inner join MyDepartment as B

on B.Id=A.Dept\_Id

where Dept\_Name='HR';

--update view using alter

alter view vW\_ForEmployees3

as

select A.\*, B.Dept\_Name

from MyEmployees as A

inner join MyDepartment as B

on B.Id=A.Dept\_Id

where Dept\_Name='HR' or Dept\_Name='Accounts';

select \* from vW\_ForEmployees;

select \* from vW\_ForEmployees1;

select \* from vW\_ForEmployees2;

select \* from vW\_ForEmployees3;

sp\_helptext vW\_ForEmployees3;

--delete view

drop view vW\_ForEmployees;

create view vW\_ForMyEmployees

as

select \* from MyEmployees;

select \* from vW\_ForMyEmployees;

--insert view

insert into vW\_ForMyEmployees values(106,'Areeb','Male','35000','New Delhi',3);

--update view

update vW\_ForMyEmployees set Emp\_Name='Sufiyan' where Emp\_Id=102;

--delete from view

delete from vW\_ForMyEmployees where Emp\_Id=106;

select \* from MyEmployees;

**The LIKE Operator:**

* The LIKE Operator is used in WHERE clause to search for a specified pattern in a column.
* There are 3 wildcards used in conjunction with the LIKE operator.

1. % - The percent sign represents zero, one, or multiple characters.
2. \_ - The Underscore represents a single character.
3. [] – for multiple characters

WHERE CustomerName LIKE ‘a% 🡪 find any values that starts with “a”.

WHERE CustomerName LIKE ‘%a’ 🡪 finds any values that ends with "a”.

WHERE CustomerName LIKE ‘%or% 🡪 finds any values that have “or” in any position

WHERE CustomerName LIKE ‘\_r%’ 🡪 finds any values that have “r” in the second position

--Start with a char

select \* from MyEmployees where Emp\_Name like 'a%';

--ends with a char

select \* from MyEmployees where Emp\_Name like '%a';

--in between a char

select \* from MyEmployees where Emp\_Name like '%a%';

--after a having 3 char

select \* from MyEmployees where Emp\_Name like 'a\_\_\_';

--before n having 3 char

select \* from MyEmployees where Emp\_Name like '\_\_\_n';

--shows employee star with a,b,s char

select \* from MyEmployees where Emp\_Name like '[a,b,s]%';

--show all records from a to z

select \* from MyEmployees where Emp\_Name like '[a-z]%';

--check 2 char having s char

select \* from MyEmployees where Emp\_Name like '\_s%';

**SUB QUERY IN SQL SERVER:**

* A Sub query or Inner query or Nested query is a query within another SQL query and embedded within the where clause.
* A sub query is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
* There are a few rules that subqueries must follow:

1. Subqueries must be enclosed within parentheses.
2. A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
3. An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY.
4. Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
5. We can also update and delete with subquery.
6. We can display particular columns in outer query with subquery.

Query:

--show employees which have greater than 30000 salary

select \* from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where Salary >30000);

--in the sub query we pass only one expression

select \* from MyEmployees

where Emp\_Id in

(select Emp\_Id,Emp\_Name from MyEmployees where Salary >30000);

--use particular columns

select Emp\_Id,Emp\_Name,Salary from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where Salary >30000);

select \* from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where City='Gurgaun');

select \* from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where Gender='Female');

--we can use update query in sub query

update MyEmployees set Salary=Salary+2000

where Emp\_Id in

(select Emp\_Id from MyEmployees where City='Hydrabad');

--we can use delete query in sub query

delete from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where City='Kashi');

--subquery used with two table

select \* from MyEmployees

where Dept\_Id in

(select id from MyDepartment where Dept\_Name='Accounts');

--order by is not used in subquery

select \* from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where Salary >30000 order by Emp\_Id asc);

--but we can use order by in outer query

select \* from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where Salary >30000 )

order by Emp\_Id asc;

**TYPES OF SUBQUERIES**

Sub-queries can be divided into two main categories.

* **SCALAR SUBQUERIES**🡪 Subqueries that return one row to the outer SQL statement.

Operators: = > >= <= !=

* **MULTIVALUED SUBQUERIES** 🡪Subqueries that return more than one row to the outer SQL statement.

Operators: in, any and all

**Query:**

**--Scalar subqueries =,>,<,>=,<=**

select \* from MyEmployees

where Dept\_Id =

(select id from MyDepartment where Dept\_Name ='Accounts');

**--multivalued subqueries- in all any**

select \* from MyEmployees

where Emp\_Id in

(select Emp\_Id from MyEmployees where Gender ='Male');

--it will give all the records which have salary less than 47500 means Ali salary

select \* from MyEmployees

where Salary <any

(select Salary from MyEmployees where Emp\_Name ='Ali'

or Emp\_Name='Aman');

--it will give output which have salary leass than 46000 and also less than 47500

select \* from MyEmployees

where Salary <all

(select Salary from MyEmployees where Emp\_Name ='Ali'

or Emp\_Name='Aman');

* **Sub-queries can be further divided into two categories:**

**SELF-CONTAINED SUBQUERIES:**

* These queries are written as standalone queries, without any dependencies on the outer query.
* A self-contained subquery is processed once when the query runs and passes its results to the outer query.

**CORRELATED SUBQUERIES:**

* There queries reference one or more columns from the outer query and therefore, depend on the outer query.
* Correlated subqueries cannot be run separately from the outer query.

Note: We can use scalar aggregate functions in subqueries because where clause don’t allow us to use aggregate functions with where clause.

**Self-contained or Co-related subqueries**

QUERY:

select \* from MyEmployees;

select \* from MyDepartment;

**--SELF-CONTAINED SUBQUERIES**

select \* from MyEmployees

where Dept\_Id in

(select id from MyDepartment where Dept\_Name='HR');

**--CO-RELATED SUBQUERIES**

select \*from MyEmployees as e

where e.Dept\_Id in

(select d.id from MyDepartment as d where e.Gender='Male');

--CO\_RELATED SUBQUERIES

select \*from MyEmployees as e

where e.Dept\_Id in

(select d.id from MyDepartment as d where e.Salary>35000);

--We can use aggrigate function in sub queries

select \* from MyEmployees

where Salary in

(select min(Salary) from MyEmployees);

--**But we can not use aggregate function with where clause**

select \* from MyEmployees

where Salary= min(Salary);

**QUERY FOR BETWEEN:**

select \* from [dbo].[Employee\_Details];

select \* from Employee\_Details

where Salary between 33000 and 35000;

**QUERY FOR SHOW TOP 1**:

select top 1 \* from Employee\_Details;

**QUERY FOR NOT IN:**

select \*from Employee\_Details where City not in ('Gurgaun','Samastipur');

**The SELECT INTO statement:**

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

* Mostly it used to create backups of the tables.
* We can copy all the rows and columns from an existing table into a new table.
* We can copy SELECTED COLUMNS into a new table.
* Copy columns from 2 or more tables into a new table (using joins), all columns or selected columns.
* We can copy data from one database table and the paste it to another database new table.

QUERY:

select \* from MyEmployees;

select \* from MyDepartment;

**-- it will create a new table name as Employee\_Backup and insert values in it**

select \* into Employee\_Backup from MyEmployees;

select \* from Employee\_Backup;

drop table Employee\_Backup;

select Emp\_Id, Emp\_Name, Salary

into Employee\_Backup1 from MyEmployees;

select \* from Employee\_Backup1;

drop table Employee\_BackUp2;

--• We can copy SELECTED COLUMNS into a new table.

select \* into Employee\_BackUp2

from MyEmployees

where City='Hydrabad';

select \* from Employee\_BackUp2;

drop table Employee\_BackUp2;

--• Copy columns from 2 or more tables into a new tables (using joins),

select \* into Employee\_Backup

from MyEmployees as A

inner join MyDepartment as B

on B.Id=A.Dept\_Id;

select \* from Employee\_Backup;

drop table Employee\_Backup;

**INSERT INTO SELECT Statement:**

* In SQL we can copy the data rows from one table to another existing table.
* On the other hand, select into statement can copy the data rows from one table to another new table.
* Copy from source table and paste to target table both actions are executed in a single SQL statement.
* The Source table and target table must have similar table definition (same columns with same data type).
* Any existing rows or records in target table remain un-affected.

create table Employee

(

Empid int primary key,

EmpName varchar(50) not null,

EmpDpt varchar(50) not null,

Salary varchar(50) not null

);

select \* from Employee;

insert into Employee values(1,'ABC','xyz','12000');

create table Students

(

Empid int primary key,

EmpName varchar(50) not null,

EmpDpt varchar(50) not null,

Salary varchar(50) not null

);

select \* from Students;

insert into Students values(8,'ABC','xyz','12000');

truncate table Students;

insert into Employee select \* from Students;

Stored Procedure:

A Stored Procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs.

**Types of Stored Procedures:**

1. System Stored Procedures
2. User-Defined Stored Procedures

WORK TO DO

1. Stored Procedure example
2. Store Procedure with single parameter.
3. Store Procedure with multiple parameters, changing the parameters order.
4. Alter with Stored Procedure
5. Seeing the text of the sp. sp\_helptext
6. Drop with Stored Procedure

Microsoft uses sp\_prefix for system stored procedures.

7.Using WITH ENCRYPTION in stored procedures.

sp\_help

View the information about the stored procedure like parameter names, their datatypes etc. sp\_help can

**QUERIES:**

**--STORE PROCEDURE WITH INPUT PARAMETER**

create table Employee\_Details

(

id int primary key not null,

Name varchar(50) not null,

Gender varchar(50) not null,

Salary varchar(50) not null,

City varchar(50) not null

);

insert into Employee\_Details values(1,'Keshav','Male','33000','Gurgaun');

insert into Employee\_Details values(2,'Madhav','Male','34000','Samastipur');

select \* from Employee\_Details;

--Stored Procedure example

**--create store procedure**

create procedure spGetEmployees

as

begin

select Name,Gender from Employee\_Details;

end

--for view the Store procedure method 1

spGetEmployees;

--for view the Store procedure method 2

execute spGetEmployees;

---Store Procedure with single parameter

create procedure spGetEmployeeById

@id int

as

begin

select \* from Employee\_Details where id=@id;

end

spGetEmployeeById 2;

**--Store Procedure with multiple parameters, changing the parameters**

create procedure spGetEmployeeByIdAndName

@id int,

@Name varchar(50)

as

begin

select \* from Employee\_Details

where id=@id and Name=@Name;

end

spGetEmployeeByIdAndName 1,'Keshav';

spGetEmployeeByIdAndName 2,'Madhav';

**--Alter with Stored Procedure**

alter procedure spGetEmployeeByIdAndName

@id int,

@Name varchar(50)

as

begin

select Name,Salary from Employee\_Details

where id=@id and @Name=Name;

end

spGetEmployeeByIdAndName 1,'Keshav';

--Seeing the text of the sp. sp\_helptext

sp\_helptext spGetEmployeeByIdAndName

--Drop with Stored Procedure

drop procedure spGetEmployeeByIdAndName;

--Using WITH ENCRYPTION in stored procedures.

create procedure spGetEmployeeByIdAndName1

@id int,

@Name varchar(50)

with encryption

as begin

select Name,Salary from Employee\_Details

where id=@id and Name=@Name;

end

spGetEmployeeByIdAndName1 1,'Keshav'

--not showing because of encryption

sp\_helptext spGetEmployeeByIdAndName1;

--decryption

alter procedure spGetEmployeeByIdAndName1

@id int,

@Name varchar(50)

as begin

select Name,Salary from Employee\_Details

where id=@id and Name=@Name;

end

spGetEmployeeByIdAndName1 1,'Keshav'

sp\_helptext spGetEmployeeByIdAndName1;

**-- STORE PROCEDURE WITH OUTPUT PARAMETER:**

--store procedure with output parameter

create procedure spGetEmployeeByGender

@Gender varchar(50),

@EmployeeCount int Output

as

begin

select @EmployeeCount=Count(id) from Employee\_Details

where Gender=@Gender;

end

Declare @TotalEmployee int

execute spGetEmployeeByGender 'Male',@TotalEmployee output

select @TotalEmployee as Male\_Employee;

Declare @TotalEmployee1 int

execute spGetEmployeeByGender 'Female',@TotalEmployee1 output

select @TotalEmployee1 as Female\_Employee;

How do you display columns as rows vs. rows as columns?

Queries:

--How to convert columns as rows

create table Emp\_Compensation

(

emp\_id int,

salary\_component\_type varchar(50),

val int

);

insert into Emp\_Compensation values(1,'salary',10000);

insert into Emp\_Compensation values(1,'bonus',5000);

insert into Emp\_Compensation values(1,'hike\_percent',10);

insert into Emp\_Compensation values(2,'salary',15000);

insert into Emp\_Compensation values(2,'bonus',7000);

insert into Emp\_Compensation values(2,'hike\_percent',8);

select \* from Emp\_Compensation;

select

emp\_id,

sum(case when salary\_component\_type='salary' then val end) as salary,

sum(case when salary\_component\_type='bonus' then val end) as bonus,

sum(case when salary\_component\_type='hike\_percent' then val end) as hike\_percent

from Emp\_Compensation

group by emp\_id;

--create a new table using select for above

select

emp\_id,

sum(case when salary\_component\_type='salary' then val end) as salary,

sum(case when salary\_component\_type='bonus' then val end) as bonus,

sum(case when salary\_component\_type='hike\_percent' then val end) as hike\_percent

into Emp\_Compensation\_pivot

from Emp\_Compensation

group by emp\_id;

select \*from Emp\_Compensation\_pivot;

--un pivot rows to columns

select \* from(

select emp\_id, 'salary' as salary\_component\_type,salary as val from Emp\_Compensation\_pivot

union all

select emp\_id, 'bonus' as salary\_component\_type,bonus as val from Emp\_Compensation\_pivot

union all

select emp\_id, 'hike\_percent' as salary\_component\_type,hike\_percent as val from Emp\_Compensation\_pivot

) a

order by emp\_id

**Functions In SQL Server:**

* SQL Server functions are useful objects in SQL Server database.
* A function is a set of SQL statements that perform a specific task.
* Functions faster code reusability.
* If you have to repeatedly write large SQL scripts to perform the same task, you can create a function that performs that task.
* Next time instead of rewriting the SQL, you can simply call that function.
* A function accepts inputs in the form of parameters and returns a value.
* SQL Server comes with a set of built-in functions that perform a variety of tasks.
* In SQL Server, a function is a stored program that we can pass parameters into and return a value.
* A SQL Server function is a code snippet that can be executed on a SQL Server.
* Of course, we could create a stored procedure to group a set of SQL statements and execute them, however, stored procedure cannot be called within SQL statements.
* Therefore, if you are using functions with large data sets, you can hit performance issues.
* In T-SQL (Transact SQL), a function is considered an object. Here are some of the rules when creating functions in SQL Server.
* A function must have a name and a function name can never start with a special character such as @, $, #, and so on.
* Functions only work with select statements.
* Functions can be used anywhere in SQL, like AVG, COUNT, SUM, MIN, DATE and so on with select statements.
* Function compiles every time.
* Functions must return a value or result.
* Functions only work with input parameters.
* Try and catch statements are not used in functions.

**SQL Server Function Types:**

* SQL Server supports two types of functions – user defined and system.
* User Defined function: User defined functions are created by a user.
* System Defined function: System functions are built in database functions.

There are three types of User-defined functions in SQL Server:

1. Scalar Functions
2. Inline Table Valued Functions
3. Multi-Statement Table Valued Functions

What are Scalar Functions?

* SQL Server scalar function takes one or more parameters and **returns a single(scalar)** value.
* The returned value can be of any data type, except text, ntext, image, cursor and timestamp.

The following are some key takeaways of the scalar functions:

* Scalar functions can be used almost anywhere in T-SQL statements.
* Scalar functions accept one or more parameters but return only one value, therefore, they must include a return statement.
* Scalar functions can use logic such as IF blocks or WHILE loops.
* Scalar functions cannot update data. They can access data, but this is not a good practice.
* Scalar functions call other functions.

QUERIES:

create database StudentDB;

--Create a function without parameter

create function ShowMessage()

returns varchar(100)

as

begin

return 'Welcome to function'

end

--dbo - database owner

select dbo.ShowMessage();

-- create a function with single parameter

create function TakeANumber(@num as int)

returns int

as

begin

return (@num \* @num)

end

select dbo.TakeANumber(2);

--alter function

alter function TakeANumber(@num as int)

returns int

as

begin

return (@num\*@num\*@num)

end

select dbo.TakeANumber(2);

-- drop a function

drop function dbo.TakeANumber;

-- create a function with multiple parameter

create function Addition(@num1 as int , @num2 as int )

returns int

as

begin

return (@num1 + @num2)

end

select dbo.Addition(1,2) as Addition;

select dbo.Addition(3,2)as Addition;

--create function using if else Condition

create function CheckVotersAge(@age as int)

returns varchar(100)

as

begin

declare @str varchar(100)

if @age >= 18

begin

set @str = 'You are eligible to vote'

end

else

begin

set @str= 'You are not eligible to vote'

end

return @str

end

select dbo.CheckVotersAge(20) ;

select dbo.CheckVotersAge(10) ;

-- create function using gettime

create function GetMyDate()

returns datetime

as

begin

return getdate()

end

select dbo.GetMyDate();

**INLINE TABLE VALUED FUNCTIONS- USER DEFINED FUNCTIONS IN SQL SERVER**

* Contains a single TSQL statement and returns a Table Set.

Scalar Functions:

* It returns a Scalar value.

Inline Table Valued Function:

* It returns a Table.

Follow the steps to create Inline Table Valued Function:

Step 1:

* We have to specify TABLE as the return type, instead of any scalar data type like int, varchar etc.

Step 2:

* There is no BEGIN and END Blocks.

Step 3:

* The table that gets returned, is determined by the SELECT command within the function.

use StudentDB;

create table Teacher\_tbl

(

T\_id int primary key,

T\_name varchar(50) not null,

T\_Qualification varchar(50) not null,

T\_Age int not null

);

insert into Teacher\_tbl values(10,'Sri Abhinav','MTECH',35);

insert into Teacher\_tbl values(15,'Pushpendra','MTECH',36);

insert into Teacher\_tbl values(20,'Raghavendra','PHD',37);

insert into Teacher\_tbl values(25,'Jithesh','BTECH',31);

insert into Teacher\_tbl values(30,'Mahesh','MSC',32);

select \* from tbl\_Student;

-- create Inline function without parameter

create function fnGetStundents()

returns table

as

return (select \* from tbl\_Student)

select \* from fnGetStundents();

select \* from dbo.fnGetStundents();

--create Inline function with single parameter

create function fn\_GetStudentById(@id int)

returns table

as

return (select \* from tbl\_Student where id =@id)

select \* from fn\_GetStudentById(1);

create function fn\_GetStudentWithAge(@age int)

returns table

as

return (select \* from tbl\_Student where age >=@age)

select \* from fn\_GetStudentWithAge(26);

--function using join

select \* from fn\_GetStudentWithAge(18) as A

inner join Teacher\_tbl as B

on A.Teacher\_id= B.T\_id;

**Multi Statement Table Valued Functions- User Defined functions in SQL Server**

* A multi-statement table-valued function is a table-valued function that returns the result of multiple statements.
* The multi-statement table-valued function is very useful because you can execute multiple queries within the function and aggregate results into the returned table.
* To define a multi-statement table-valued function, you use a table variable as the return value, inside the function, you execute one or more queries and insert data into this table variable.

Difference between Inline table valued function and Multi-statement table valued function:

|  |  |
| --- | --- |
| Inline table valued functions | Multi-statement table valued functions |
| In this, the return clause cannot contain the structure of the table. | In this, we specify the structure of the table with returns clause. |
| In this, there are no BEGIN and END blocks. | In this, we have to use BEGIN and END blocks. |
| Inline table-valued functions are better in performance as compared to multi statement table-valued functions. | There is no performance advantage in multi-statement table-valued functions. |
| Internally, Inline table-valued function much like it would a view. | Internally, multi statement table-valued function much like it would a stored procedure. |

Similarities:

* Inline statement table-valued function and multi statement table-valued functions both are table-valued functions.
* Inline statement table- valued function and multi statement table-valued functions both are located in Table-valued functions folder in SSMS.
* Both are the types of user-defined functions in SQL Server.

Query:

use StudentDB;

select \* from tbl\_Student;

--Multi statement table

create function fn\_GetStudentByGender(@gender varchar(20))

returns @myTable table(Std\_id int , Std\_Name varchar(50), Std\_gender varchar(50))

as

begin

insert into @myTable

select id,name,gender from tbl\_Student where gender=@gender

return

end

select \* from[dbo].[fn\_GetStudentByGender]('Male')

select \* from[dbo].[fn\_GetStudentByGender]('Female')

--inline statement table

create function fn\_GetStudentByGender2(@gender varchar(20))

returns table

as

return

(select id,name,gender from tbl\_Student where gender=@gender)

select \* from [dbo].[fn\_GetStudentByGender2]('Male')

Difference between functions and stored procedures in SQL Server:

|  |  |
| --- | --- |
| FUNCTIONS | STORED PROCEDURES |
| Function must return a value. | Stored Procedure may or may not return values. |
| Create function fn\_Addtion\_Of\_Number(@int1 as int)  Returns int  As  begin  return (@int1+@int1)  end | Create procedure spGetData  as  begin  select \* from student  end |
| Function can have only input parameters. | Stored Procedures can have both input or output parameters. |
| Functions can be called from Stored Procedures. | Stored Procedures cannot be called from a Function. |
| Create function fn\_Addtion\_Of\_Number(@int1 as int)  Returns int  As  begin  return (@int1+@int1)  end | Create procedure spAddition  As begin  Selectv dbo.fn\_Addition\_Of\_Numbers(5)  End  Execute spGetAddition |
| Functions allows only SELECT statement in it. | The Stored Procedure allows SELECT as well as DML (INSERT/UPDATE/DELETE) statement in it. |
| Function can be embedded in a select statement. | Stored Procedures cannot be utilized in a SELECT statement. |
| Functions can be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section. | Stored Procedures cannot be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section. |
| Try-catch block cannot be used in a Function. | An exception can be handled by try-catch block in a Stored Procedure. |
| We cannot use Transactions in Function. (Transactions: Commit, Rollback concepts) | We can use Transactions in Stored Procedure. |
| We use SELECT command to execute a function. | We use EXEC or EXECUTE keywords to execute a stored procedure. |
| Functions are computed values and cannot perform permanent environmental changes to SQL Server (i.e., no INSERT or UPDATE statement allowed). | Stored Procedures can perform permanent environmental changes to SQL Server (i.e. INSERT or UPDATE statement are allowed). |
| A Function can be used in join clause as a result set. | Procedures cannot be used in Join clause. |

**Normalization:**

* Database Normalization is the process of organizing data to minimize data redundancy (Data Duplication) which leads to data inconsistency.

Disadvantages of Data Redundancy:

* Disk Space Wastage
* Data Inconsistency
* DML (Data Manipulation language) insert update delete queries can become slow.

🡪Database Normalization is a step-by-step process. There are 6 Normal forms.

🡪First Normal Form (1NF) to Sixth Normal Form (6NF). Most database are in third normal form (3NF). There are certain rules, that each normal form should follow.

FISRT NORMAL FORM:

1. The Data in each column should be atomic, no multiple values separated by comma.
2. The Table does not contain any repeating column groups.
3. Identify each record uniquely using Primary Key.

|  |  |
| --- | --- |
| DEPARTMENT NAME | EMPLOYEE NAME |
| IT | ANAS, OSAMA, AMIR |
| HR | RAGHAV |

* It is not possible to select, insert, update and delete just one employee.

No repeating column groups.

|  |  |  |  |
| --- | --- | --- | --- |
| DEPT NAME | EMPLOYEE 1 | EMPLOYEE 2 | EMPLOYEE 3 |
| IT | ANAS | OSAMA | AMIR |
| HR | RAGHAV |  |  |

* More than 4 employees, table structure change required. Less than 3 employees, waste disk space

SECOND NORMAL FORM (2NF):

* The table is said to be in 2NF, If
* 1. The table meets all the conditions of first normal form.
* 2. There are no partial dependencies in the table.
* Partial dependency means a non-key attribute should not be partially dependent on more than one key attribute.
* 3. Move redundant data to a separate table.
* 4. Create relationships between these tables using foreign keys.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| EMP ID | DEPT ID | NAME | AGE | SALARY | DEPNAME | DEPHEAD | DEPLOC |
| 1 | 1 | ABC | 27 | 45000 | IT | XYZ | GURGAON |
| 2 | 2 | GWDH | 25 | 34000 | HR | JHG | DELHI |
| 3 | 1 | VCX | 24 | 25000 | IT | LKJ | PATNA |

TABLE DESIGN IN SECOND NORMAL FORM

|  |  |  |  |
| --- | --- | --- | --- |
| DEPT ID | DEPT NAME | DEPT HEAD | DEPT LOCATION |
| 1 | IT | XYZ | GURGAON |
| 2 | HR | JHG | DELHI |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EMP ID | EMP NAME | GENDER | SALARY | DEPT ID |
| 1 | XYZ | MALE | 45000 | 1 |
| 2 | GWDH | FEMALE | 34000 | 2 |

THIRD NORMAL FORM OF NORMALIZATION:

* A table is said to be in 3rd NF
* 1. Meets all the conditions of first normal form and second normal form.
* 2. The Tables should not have TRANSITIVE DEPENDENCIES in them.
* Does not contain columns (attributes) that are not fully dependent on the primary key.
* If an attribute can be determined by another non-key attribute, it is called a transitive dependency.
* To make it simpler, every non key attribute should be determined by the key attribute only.
* If a non-key attribute can be determined by another key attribute, it needs to put into another table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| EMP ID | EMP NAME | GENDER | SALARY | ANNUAL SALARY | DEPT ID |
| 1 | AMIR | MALE | 45000 | 540000 | 1 |
| 2 | SANA | FEMALE | 34000 | 408000 | 2 |

Here annual salary is dependent on salary, so we have to put it on another table.

|  |  |  |
| --- | --- | --- |
| Salary | Annual salary | Emp id |
| 45000 | 540000 | 1 |
| 34000 | 408000 | 2 |

What is Normalization?

* Normalization is a database design technique that reduces data redundancy.

How to apply Normalization?

* Normalization is applied by breaking up the table into 2 or we can say that by splitting the table, first table having primary key and second table having foreign key.

What is De-Normalization?

* De-Normalization is used to combine multiple table data into one so that it can be queried quickly.

**Primary Key VS Unique key:**

SIMILARITY:

* They both ensure that no duplicate values are present in a column.
* Index is automatically added with Primary key or unique key.

DIFFERENCE:

|  |  |
| --- | --- |
| PRIMARY KEY | UNIQUE KEY |
| Primary key cannot have nulls. | Unique key can have nulls. |
| We can have one primary key in a table. | We can have more than one Unique keys in a table. |

**How to optimize a Stored Procedure or SQL Query?**

* Use SET UNCOUNT ON
* Specify column names instead of using \* in SELECT statement.
* Use schema name before objects or tablenames.

Ex: SELECT EmpID , NAME from dbo.Employee.

* Do not use DYNAMIC QUERIES. They aee vulnerable to SQL Injection.
* Use EXISTS() instead of COUNT ().

Ex: SELECT Count(1) from dbo.Employee. (not like this).

Ex: IF(Exists(select 1 from dbo.Employee)).(use like this)

* Use Transaction when required only

**Trigger:**

A trigger is a special kind of stored procedure that automatically executes when an event occurs in the database server.

There are three types of triggers.

* DML Triggers (Data Manipulation Language) INSERT, UPDATE, DELETE
* DDL Triggers (Data Definition Language) CREATE, ALTER
* LOGON Triggers

DML Triggers are fired automatically in response to DML events. (INSERT, UPDATE AND DELETE)

* DML Triggers can be of 2 types.
* AFTER Triggers (Also called for Triggers)
* Instead of Triggers

QUERY:

* AFTER Triggers (Also called for Triggers)

use Batch\_17028;

create table tbl\_Student(

Id int primary key,

Name varchar(50) not null,

Gender varchar(50) not null,

Class int not null,

fees int not null

);

select \* from dbo.tbl\_Student;

insert into tbl\_Student values(1,'Keshav','Male',10,4000);

insert into tbl\_Student values(2,'Madhav','Male',9,5000);

insert into tbl\_Student values(3,'Anand','Male',8,7000);

insert into tbl\_Student values(4,'Anup','Male',10,6000);

insert into tbl\_Student values(5,'Anjali','Female',7,3500);

insert into tbl\_Student values(6,'Shivam','Male',5,3000);

create trigger tr\_Student\_forinsert

on tbl\_Student

after insert

as

begin

print 'Something happened to the student table';

end

insert into tbl\_Student values(7,'Rohit','Male',11,8600);

alter trigger tr\_Student\_forinsert

on tbl\_Student

after insert

as

begin

select \* from inserted

end

insert into tbl\_Student values(8,'Ranjeet','Male',12,8600);

insert into tbl\_Student values(9,'Ramesh','Male',11,8600);

insert into tbl\_Student values(10,'Rakesh','Male',10,8600);

insert into tbl\_Student values(11,'Rohan','Male',10,8600);

insert into tbl\_Student values(12,'Rohn','Male',10,8600);

select \* from dbo.tbl\_Student;

create trigger tr\_Student\_forDelete

on tbl\_Student

after delete

as

begin

select \* from deleted

end

delete from tbl\_Student where id=8;

create table tbl\_Student\_Audit

(

Audit\_id int primary key identity,Audit\_Info varchar(max)

);

select \* from dbo.tbl\_Student\_Audit;

create trigger tr\_Student\_audit\_forinsert

on tbl\_Student

after insert

as

begin

Declare @id int

Select @id=id from inserted

insert into tbl\_Student\_Audit

values('Student with id '+ Cast(@id as varchar(50)) + ' is added at ' + Cast(GETDATE() as varchar(50)));

end

alter trigger tr\_Student\_audit\_forinsert

on tbl\_Student

after insert

as

begin

Declare @id int

Select @id=id from inserted

insert into tbl\_Student\_Audit

values('Student with id '+ Cast(@id as varchar(50)) + ' is added at ' + Cast(GETDATE() as varchar(50)));

end

create trigger tr\_Student\_audit\_fordelet

on tbl\_Student

after delete

as

begin

Declare @id int

Select @id=id from deleted

insert into tbl\_Student\_Audit

values('Existing Student with id '+ Cast(@id as varchar(50)) + ' is deleted at ' + Cast(GETDATE() as varchar(50)));

end

alter trigger tr\_Student\_audit\_fordelet

on tbl\_Student

after delete

as

begin

Declare @id int

Select @id=id from deleted

insert into tbl\_Student\_Audit

values('Existing Student with id '+ Cast(@id as varchar(50)) + ' is deleted at ' + Cast(GETDATE() as varchar(50)));

end

delete from tbl\_Student where id =11;

select \* from dbo.tbl\_Student\_Audit;

create trigger tr\_Student\_forUpdate

on tbl\_Student

after update

as

begin

select \* from inserted

select \* from deleted

end

update tbl\_Student set Name='Hari' where id = 10;

sp\_helptext tr\_Student\_audit\_forinsert

DML INSTEAD OF TRIGGERS:

* Instead of triggers is executed in place of the insert, update or delete operations.

AS we know that there are two types of DML TRIGGERS:

1. After Triggers
2. Instead of Triggers

There are 3 types of Instead Triggers

1. Instead of Insert Trigger
2. Instead of Update Trigger
3. Instead of Delete Trigger

* Are executed before constraint checks are performed on the table and after the creation of the inserted and deleted tables.
* We can view the definition of triggers by using SP\_HELPTEST System
* We can use alter statement with triggers.
* Modifying DML Triggers🡪 1. Drop and Recreate & Alter trigger statement.
* A DML Trigger can be encrypted to hide its definition or query by using with encryption command.

QUERY:

create table Tbl\_Customer

(

Id int primary key,

Name varchar(50),

Gender varchar(50),

City varchar(50),

ContactNo varchar(50)

);

insert into dbo.Tbl\_Customer values(1,'Keshav','Male','Samastripur','9162232443');

insert into dbo.Tbl\_Customer values(2,'Madhav','Male','Samastripur','9162232444');

insert into dbo.Tbl\_Customer values(3,'Rakesh','Male','Begusarai','9162232445');

insert into dbo.Tbl\_Customer values(4,'Aman','Male','Patna','9162232446');

insert into dbo.Tbl\_Customer values(5,'Rohan','Male','Dalsinghsarai','9162232447');

insert into dbo.Tbl\_Customer values(6,'Sandeep','Male','Bhagalpur','9162232448');

select \* from dbo.Tbl\_Customer;

--instead of insert

create trigger tr\_Customer\_InsteadOf\_Insert

on Tbl\_Customer

instead of insert

as

begin

print 'You are not allowed to insert data in this table!!'

end

-- not inserted

insert into dbo.Tbl\_Customer values(7,'Sandi','Male','Bhagalpur','9162232448');

drop trigger tr\_Customer\_InsteadOf\_Insert;

--instead of update

create trigger tr\_Customer\_InsteadOf\_Update

on Tbl\_Customer

instead of update

as

begin

print 'You are not allowed to update data in this table!!'

end

--not updated

update Tbl\_Customer set Name='Amar' where id =6;

drop trigger tr\_Customer\_InsteadOf\_Update;

--instead of delete

create trigger tr\_Customer\_InsteadOf\_Delete

on Tbl\_Customer

instead of delete

as

begin

print 'You are not allowed to delete data in this table!!'

end

--not deleted

delete from Tbl\_Customer where id = 6;

drop trigger tr\_Customer\_InsteadOf\_Delete;

select \* from dbo.Tbl\_Customer;

create table Customer\_Audit\_table

(

Audit\_Id int primary key identity,

Audit\_Information varchar(max)

);

select \*from Customer\_Audit\_table;

-- cast is used for change the data type

--we only perform one instead of trigger in a table

create trigger tr\_Customer\_InsteadOf\_Insert\_Audit

on Tbl\_Customer

instead of insert

as

begin

insert into Customer\_Audit\_table values('Someone tries to insert data in customer table at ' + cast(GETDATE() as varchar(50)) );

end

insert into dbo.Tbl\_Customer values(7,'Sandi','Male','Bhagalpur','9162232448');

select \* from Tbl\_Customer;

select \* from Customer\_Audit\_table;

create trigger tr\_Customer\_InsteadOf\_Update

on Tbl\_Customer

instead of update

as

begin

insert into Customer\_Audit\_table values('Someone tries to update data in customer table at ' + cast(GETDATE() as varchar(50)));

end

update Tbl\_Customer set Name='Amar' where id =6;

select \* from Tbl\_Customer;

select \* from Customer\_Audit\_table;

create trigger tr\_Customer\_InsteadOf\_Delete

on Tbl\_Customer

instead of delete

as

begin

insert into Customer\_Audit\_table values('Someone tries to delete data in customer table at ' + cast(GETDATE() as varchar(50)));

end

delete from Tbl\_Customer where id = 6;

select \* from Tbl\_Customer;

select \* from Customer\_Audit\_table;

USING INSTEAD OF TRIGGERS WITH VIEWS:

* It can be specified on tables as well as views and provides a wider range and types of updates that the user can perform against a view.

QUERY:

create table Employee\_Personal\_Details

(

EmpID int primary key,

FirstName varchar(50),

LastName varchar(50),

Address varchar(100)

);

insert into Employee\_Personal\_Details values(1,'Keshav','Kumar','Madanpuri Sector 7 Gurgaon');

insert into Employee\_Personal\_Details values(2,'Rakesh','Kumar','Madanpuri Sector 8 Gurgaon');

insert into Employee\_Personal\_Details values(3,'Ramesh','Kumar','Madanpuri Sector 9 Gurgaon');

insert into Employee\_Personal\_Details values(4,'Ranjit','Kumar','Madanpuri Sector 10 Gurgaon');

select \* from Employee\_Personal\_Details;

create table Employee\_Salary\_Details

(

EmpID int primary key,

Designation varchar(50),

Salary varchar(50)

);

insert into Employee\_Salary\_Details values(1,'PA','33000');

insert into Employee\_Salary\_Details values(2,'PAT','26000');

insert into Employee\_Salary\_Details values(3,'HR','25000');

insert into Employee\_Salary\_Details values(4,'AC','23000');

select \* from Employee\_Personal\_Details;

select \* from Employee\_Salary\_Details;

create view vW\_Employees

as

select A.EmpID,A.FirstName,A.LastName,

B.Designation,B.Salary

from Employee\_Personal\_Details as A

inner join Employee\_Salary\_Details as B

on A.EmpID=B.EmpID;

select \* from vW\_Employees;

delete from vW\_Employees where EmpID =4;

--delete particular employees and salary details from both table using trigger

create trigger tr\_InsteadOf\_EmployeeDetails

on vW\_Employees

instead of delete

as

begin

delete from Employee\_Personal\_Details where EmpID in

(select EmpID from deleted)

delete from Employee\_Salary\_Details where EmpID in

(select EmpID from deleted)

end

delete from vW\_Employees where EmpID =4;

select \* from Employee\_Personal\_Details;

select \* from Employee\_Salary\_Details;

**DDL TRIGGERS IN SQL SERVER:**

Data Definition Language (DDL) triggers execute stored procedures when DDL events such as CREATE, ALTER, and DROP statements occur in the database or the server.

What is the use of DDL Triggers?

* DDL triggers can be used to prevent modifications in the database schema.
* A schema is a collection of objects such as tables, views, and so forth in a database.
* If you want to execute some code or message in response to a specific DDL event.
* To prevent certain changes to your database schema.
* Audit or check the changes that the users are making to the users are making to the database structure.

CREATE TRIGGER <trigger\_name>

ON {ALL SERVER | DATABASE}

[WITH ENCRYPTION]

{FOR |AFTER} {event\_type}

AS <sql\_statement>

Syntax:

Where:

🡪ALL SERVER: Specifies that the DDL trigger executes when DDL events occur in the current server.

🡪DATABASE: Specifies that the DDL trigger executes when DDL events occur in the current database.

🡪event\_type: Specifies the name of the DDL event that invokes the DDL trigger. For example, CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE

* DDL event type CREATE\_TABLE, ALTER\_TABLE and DROP\_TABLE.
* So many events are available for DDL Triggers. Msdn
* DDL triggers scope: DDL triggers can be created in a specific database or at the server level.
* We can use ROLLBACK command to prevent for creating, altering and dropping something in the database or server.
* We can use With Encryption with DDL triggers to hide the code definition.
* We can use drop and alter command with DDL triggers.
* We can Enable or Disable triggers for a time being as per requirements.
* Rename event is used in the DDL trigger when we invoke the system stored procedure sp\_rename.

QUERY:

create database Trigger\_DB;

use Trigger\_DB;

--create trigger for CREATE\_TABLE

create trigger tr\_ddl\_table\_create

on database

for CREATE\_TABLE

as

begin

rollback

print 'You cannot create a table in this database !!';

end

--disable tr\_ddl\_table\_create

disable trigger tr\_ddl\_table\_create on database;

create table Test2(id int)

--enable tr\_ddl\_table\_create

enable trigger tr\_ddl\_table\_create on database;

create table Test3(id int);

--create trigger for ALTER\_TABLE

create trigger tr\_ddl\_Table\_Alter

on database

for ALTER\_TABLE

as

begin

print 'YOU HAVE JUST ALTERED A TABLE !!';

end

--create trigger for DROP\_TABLE

create trigger tr\_ddl\_table\_Drop

on database

for DROP\_TABLE

as begin

print 'YOU HAVE JUST DROPPED A TABLE !!';

end

create table Test\_table(id int);

alter table Test\_table add Name varchar(50);

alter table Test\_table add City varchar(50);

drop table Test\_table;

select \* from Test\_table;

--create trigger for all ddl in one

create trigger tr\_ddl\_table

on database

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

as

begin

print 'YOU HAVE JUST CREATED, ALTERED, OR DROPPED A TABBLE !!';

end

--create trigger for stored procedure to prevent create store procedure

create trigger tr\_ddl\_SP\_CREATE

on database

for CREATE\_PROCEDURE

as

begin

ROLLBACK

print 'YOU ARE NOT ALLOWED TO CREATE A STORED PROCEDURE'

end

create procedure sp\_myProcedure

as

begin

print 'This is my store procedure!!';

end

alter trigger tr\_ddl\_SP\_CREATE

on database

with encryption

for CREATE\_PROCEDURE

as

begin

ROLLBACK

print 'YOU ARE NOT ALLOWED TO CREATE A STORED PROCEDURE'

end

--sp\_helptext is only used for show the code of table not database

--rename

create trigger tr\_ddl\_Rename

on database

for RENAME

as

begin

print 'You have just renamed a table or table column !!';

end

select \* from Test2;

--rename the table name

sp\_rename 'Test2','Test3';

select \* from Test3;

sp\_rename 'Test3.id', 'Student\_Id';

--droping the trigger

drop trigger tr\_ddl\_Rename on database;

**INDEXES IN SQL SERVER:**

* To facilitate quick retrieval of data from a database, sql server 2012 provides the indexing features.
* An index in SQL Server database contains information that allows you to find specific data without scanning through the entire table.
* Indexes are created on tables and views.
* Index on a table or a view, is very similar to an index that we find in a book.
* If you don’t have an index in a book, and I ask you to locate a specific chapter in the book, you will have to look at every page starting from the first page of the book.
* On the other hand, if you have the index in a book, you lookup the page number of the chapter in the index, and then directly go to that page number to locate the chapter.
* Obviously, the book index is helping to reduce the time it takes to find the chapter.
* In fact, the existence of the right indexes, can improve the performance of the query. If there is no index to help the query, then query engine, checks every row in the table from the beginning to end. This is called as table scan. Tabe scan is very bad for performance.
* Create indexes on columns that will be frequently searched against.
* An Index is a pointer to data in a table.
* An Index helps to speed up select queries and where caluses, but it slows down data input, with the update and the insert statements.
* Indexes can be created or dropped with no effect on the data.
* Indexes are automatically created when primary key and unique constraints are defined on a table.
* SP\_HELPINDEX system store procedure used to find indexes on a table.
* Dropping index.
* Creating index using designer.

Single-Column Indexes:

* A single-column index is created based on only one table column.

Implicit Indexes:

* Implicit Indexes are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

When should indexes be avoided?

* Although indexes are intended to enhance a database’s performance, there are times when they should be avoided.
* The following guidelines indicate when the use of an index should be reconsidered.
* Indexes should not be used on small tables.
* Tables that have frequent, large batch updates or insert operations.
* Indexes should not be used on columns that contain a high number of Null values.
* Columns that are frequently manipulated should not be indexed.

**CLUSTERED AND NON-CLUSTERED INDEX IN SQL SERVER:**

Clustered Index:

* A Clustered Index clauses records to be physically stored in a stored or sequential order.
* A Clustered Index determines the actual order in which data is stored in the database. Hence you can create only one clustered index in a table.
* Uniqueness of a value in a clustered index is maintained explicitly using the unique keyword or implicitly using an internal unique identifier.
* Clustered index is as same as dictionary where the data is arranged by alphabetical order.
* SP\_HELPINDEX system stored procedure to view the index on a table.
* We can have only one clustered index in one table, but we can have on clustered index on multiple columns, and that type of index is called composite index.

Query:

use Batch\_17028;

create table FullTimeEmployees

(

Id int ,

Name varchar(50),

Gender varchar(50),

Designation varchar(50),

Salary int

);

--drop table FullTimeEmployees;

insert into FullTimeEmployees values(2,'Keshav','Male','PA',33000);

insert into FullTimeEmployees values(4,'Madhav','Male','HR',20000);

insert into FullTimeEmployees values(1,'Rakesh','Male','Manager',22000);

insert into FullTimeEmployees values(5,'Ramesh','Male','Operator',15000);

insert into FullTimeEmployees values(3,'Rohan','Male','Account',23000);

select \* from FullTimeEmployees;

drop index FullTimeEmployees.PK\_\_FullTime\_\_3214EC07D502AD86;

create index IX\_FTE\_Salary

on FullTimeEmployees (Salary asc)

sp\_helpindex FullTimeEmployees;

select \* from FullTimeEmployees

where Salary>10000 and Salary <22000;

drop index FullTimeEmployees.IX\_FTE\_Salary;

--If table has primary key than by default table has clustered index

--create clustered index it will arrange in asc order

create clustered index IX\_FTE\_Id\_Clustered

on FullTimeEmployees(Id asc);

select \* from FullTimeEmployees;

sp\_helpindex FullTimeEmployees

--we cannot create more than one clustered index in a table so it will give error

--if we want to execute this index so we have to delete clustered index from database then it will execute

create clustered index IX\_FTE\_Gender\_Salary\_Clustered

on FullTimeEmployees(Gender asc,Salary asc);

sp\_helpindex FullTimeEmployees;

**NON-CLUSTERED INDEX:**

* A non-clustered index is as same as to an index of a book.
* The data is stored in one place, and index is stored in another place.
* Since, the non-clustered index is stored separately from the actual data, a table can have more than one non cluster index.
* Just like how book can have index by chapters at the beginning and another index by common terms at the end.

QUERY:

use Batch\_17028;

create table FullTimeEmployees

(

Id int primary key ,

Name varchar(50),

Gender varchar(50),

Designation varchar(50),

Salary int

);

--drop table FullTimeEmployees;

insert into FullTimeEmployees values(2,'Keshav','Male','PA',33000);

insert into FullTimeEmployees values(4,'Madhav','Male','HR',20000);

insert into FullTimeEmployees values(1,'Rakesh','Male','Manager',22000);

insert into FullTimeEmployees values(5,'Ramesh','Male','Operator',15000);

insert into FullTimeEmployees values(3,'Rohan','Male','Account',23000);

select \* from FullTimeEmployees;

--create nonclustered index we can use or not use nonclustered keyword

--when we use index keyword it is by default nonclustered index

create nonclustered index NIX\_FTE\_Name

on FullTimeEmployees (Name asc);

sp\_helpindex FullTimeEmployees;

create nonclustered index NIX\_FTE\_Salary

on FullTimeEmployees (Salary asc);

create nonclustered index NIX\_FTE\_Name\_Salary

on FullTimeEmployees (Name asc, Salary asc);

drop index FullTimeEmployees.[NIX\_FTE\_Name\_Salary]