SQL Notes
GET – 873 .NET Batch

1. Joins, Where and On

JOINS:

Joins are used for combining one or more tables, based on one related column between them.

There are six types of joins

1. Inner join.

2. Left join.

3. Right join.

4. Full join.

5. Self-join.

6. Cross join.

Syntax and Example :

--creating table for joins

```
--joins
 Create table tblexample1 (id int,firstname varchar(30),lastname varchar(30),phoneno int)

 alter table tblexample1 alter column id int  not null
 alter table tblexample1 add primary key(id)
 insert into tblexample1(id,firstname,lastname,phoneno) values(1,'abhi','kumar',9988776655)
 alter table tblexample1 alter column phoneno nvarchar(40)
  insert into tblexample1(id,firstname,lastname,phoneno)
values(2,'divya','jyothi',9988776656),(3,'kiran','kumar',9988776652),(4,'hema','latha',9988776653)
  select * from tblexample1

create table tblexample2 (id int,employeefirstname varchar(30),employeelastname
varchar(30),Location varchar(20))
insert into tblexample2(id,employeefirstname,employeelastname,Location)
values(101,'kishan','naidu','kakinada'),
(102,'divya','nandhini','rajahmundry'),(103,'ajith','kumar','kapileswarapuram'),(104,'siva','jyothi','manda
peta')

select * from tblexample1
select * from tblexample2

 alter table tblexample2 alter column id int  not null
 alter table tblexample2 add primary key(id)
alter table tblexample1 add eid int references tblexample2(id)

update tblexample1 set eid =101 where id =1
update tblexample1 set eid =102 where id =2
update tblexample1 set eid =103 where id =3
update tblexample1 set eid =104 where id =4

--inner join
-- inner join is used for getting the results which are included(intersection) of  both the tables
```

<u>Syntax:</u>
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;

```
insert into tblexample1(id,firstname,lastname,phoneno)
values(6,'divya','laksmi',9988776656),(7,'kishore','kumar',9988776652)
insert into tblexample2(id,employeefirstname,employeelastname,Location)
values(105,'prakash',null,'kakinada'),
(107,'harshini','ramalashmi',null)

select t.firstname,e.employeefirstname from tblexample1 t
inner join tblexample2 e
on t.eid=e.id
```

<u>Example 2</u>

```
select * from tblexample1 t
inner join tblexample2 e
on t.eid=e.id

--left join(left outer join)
--the left join returns all the values in the left table and matched records in the right table ,if nothing is macthed it will show null values
```

<u>Syntax:</u>
SELECT *column_name(s)*
FROM *table1*
LEFT JOIN *table2*
ON *table1.column_name = table2.column_name*;

```
select * from tblexample1
select * from tblexample2
select * from tblexample1 t
left join tblexample2 e
on t.eid=e.id

--right join(right outer joint)
--the right join returns all the values in the right table am=nd matched records in the left table,if nothing is matched it will show null values
```

<u>Syntax:</u>

SELECT *column_name(s)*
FROM *table1*
RIGHT JOIN *table2*
ON *table1.column_name = table2.column_name*;

```
 select * from tblexample1 t
right join tblexample2 e
on t.eid=e.id

--full join
--full join is nothing but union of two tables,along with null values
```

<u>Syntax:</u>

SELECT *column_name(s)*
FROM *table1*
FULL JOIN *table2*
ON *table1.column_name = table2.column_name*;

```
 select * from tblexample1 t
full join tblexample2 e
on t.eid=e.id

--cross join
--cross join is nothing but the value in each table are matching wwith all the all values in another table
and forms a large result table.


 select * from tblexample1 t
cross join tblexample2 e

--self join(self join is nothing but joined with itself)
```

<u>Syntax:</u>

SELECT *column_name(s)*
FROM *table1 T1, table1 T2*
WHERE *condition*;

```
select * from tblexample1
select * from tblexample2

alter table tblexample1 add Mgrid int
update tblexample1 set Mgrid=1 where id in(2,3)
update tblexample1 set Mgrid=2 where id =4
update tblexample1 set Mgrid=3 where id in(6,7)
```

<u>Where and On</u>

ON should be used to define the join condition and WHERE should be used to filter the data.

It also prevents incorrect data being retrieved when using JOINs.

Can be used to help join data through defining the condition on which the two tables are joined.

Syntax for ON

//select *

From table1

Join table2

On table1.alias name=table2.alias name

Example For ON

```
SELECT *
FROM facebook
JOIN linkedin
ON facebook.name = linkedin.name
```

This example shows the person who is both our facebook friend and our linkedin friend using ON.

Syntax for Where

//**select***

From table1, table2

Where table1.alias name= table2.alias name

Example For where

```
SELECT *
FROM facebook, linkedin
WHERE facebook.name = linkedin.name
```

This example shows the person who is both our facebook friend and our linkedin friend using WHERE.

In the first query we can easily see the tables being joined in the FROM and JOIN clause. We can also clearly see the join condition in the ON clause. In the second query we can easily see the tables being filtered and not joined.

2. Sub – Queries

Subqueries: Query inside a Query, the inner query is enclosed within the parenthesis.

Definition: A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery. The query which is nested is called inner query or inner select and the outer statement which contains the inner query is called outer query or outer select. The inner query should contain only one column in the select clause.

Syntax: select (column names) from TableName Where   (inner query)

Example: Display the name, salary of employee whose salary is greater than the average of salary of the department 101:

Select Ename , Salary from tblEmployee where salary > (select avg(salary) from tblEmployee where Did=101)

Types of subquery:

1. Single row subquery

2. Multirow subquery

3. Correlated Subquery

Single Row Subquery: Subquery with comparison operators

Inner query will return only one row.

Mostly uses the comparison operators such as **>,<,=<,=>,=, !=, ><**

Example: Display the Name, Salary of the employees whose salary is greater than the salary of employee Eid=102:

Select Ename, Salary from tblEmployee where salary> (select salary from tbllEmployee where eid =102)

Multirow Subquery: Subquery with comparison operators modified by **in, All, Any**

Inner query will return more than one row

Example for In**:** Display the Name, Salary of the employees whose salary is equal to one of the salaries in department 102:

Select Ename, Salary from tblEmployee where salary in (select salary from tblEmployee where Did =102)

Example for Any: Display the Name, Salary of the employees whose salary is greater than one of the salaries in department 102:

Select Ename, Salary from tblEmployee where salary > any (select salary from tblEmployee where Did =102)

Example for all: Display the Name, Salary of the employees whose salary is greater than all the salaries in department 102:

Select Ename, Salary from tblEmployee where salary > all (select salary from tblEmployee where Did =102)

Note: In single row and Multi row subquery, the inner query is executed First followed by the Outer query. Here the inner query is executed only once, and the inner query is independent of outer query.

Correlated Subquery:

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the WHERE clause of the outer query. In queries that include a correlated subquery (also known as a repeating subquery), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query.

Example: Display the employee details whose salary is equal to the minimum salary of their departments

```
Select * from tblEmployee as 'E' where salary = (select min(salary) from tblEmployee  where Did= E.Eid)
```

Subquery Rules and Restrictions:

- The DISTINCT keyword cannot be used with subqueries that include GROUP BY
- ORDER BY can only be specified when TOP is also specified.
- A view created by using a subquery cannot be updated.

3. DDL

DATA DEFINITION LANGUAGE (DDL):

- DDL actually consists of the SQL commands that can be used to define the database schema.
- It simply deals with descriptions of the database schema i.e., It defines the column (Attributes) of the table.
- It is used to define the type and structure of the data that will be stored in a database.
- It is used to create and modify the structure of database objects in the database.
- It can be used to define constraints as well.

SQL DDL commands are further divided into the following major categories:

1)CREATE
2)ALTER
3)DROP
4)TRUNCATE

=>CREATE

The CREATE query is used to create a database or objects such as tables, views, stored procedures, index,function and triggers.

--Creating a DataBase:
Syntax: CREATE DATABASE databasename
Example: CREATE DATABASE dbmovies

--Creating a table:
The CREATE query is also used to add tables in an existing database.
Syntax:

```
CREATE TABLE table_name
(
  column data_type [column_constraint],
  column_1 data_type [column_constraint],
  column_2 data_type,
  column_n data_type
)
```

Example:

```
create table sample (ID int constraint pk_id primary key,CollegeName nvarchar(20))
```

here, ID = column_name1;CollegeName = column_name2;int = DataType;pk_id = constraint name;column_constraint =primary key

=>ALTER
The ALTER command in SQL DDL is used to modify the structure of an already existing table.
--Adding a New column:
Syntax:
ALTER TABLE table_name ADD column_name datatype
Example:
ALTER TABLE sample ADD University varchar(25)
--Modifying an existing column:
Syntax:
ALTER TABLE table_name ALTER COLUMN column_name datatype
Example:
ALTER TABLE sample ALTER COLUMN University nvarchar(10)

--Adding Primary Key:
Adding a Primary Key after the table has been made.
Syntax:
ALTER TABLE table_name add Primary key(column_name);

--Adding Foreign Key:
Adding a Foreign Key after the table has been made.
Syntax:
ALTER TABLE table2 Add Foriegn key (column_name2) references table1 (column_name1);
here, column_name2(i.e. foreign key) is column in table2 and column_name1 is primary key of table1

=>DROP
The DROP TABLE statement is used to drop an existing table in a database.
The DROP is also used to drop a column in a table.
--Dropping Table:
Syntax:
DROP TABLE table_name;
--Dropping column:
Syntax:
ALTER TABLE table_name DROP COLUMN column_name
--Dropping a PrimaryKey From Table:
Syntax:
ALTER TABLE table_name DROP PRIMARY KEY
--Dropping a constraint:
Syntax:
ALTER TABLE table_name DROP constraint_name

=>TRUNCATE
The TRUNCATE command in SQL DDL is used to remove all the records from a table but not the structure of the table
Syntax:
TRUNCATE TABLE table_name;

4. DML

Data Manipulation Language (DML)

- In SQL, the data manipulation language comprises the *SQL-data change* statements, which modify stored data but not the schema or database objects. Manipulation of persistent database objects, e.g., tables or stored procedures, via the SQL schema statements, rather than the data stored within them, is considered to be part of a separate data definition language (DDL).
- The DML has four basic commands:

A. Select
B. Insert
C. Update
D. Delete

1. **SELECT**: It retrieves certain records from one or more table.

Syntax: Select * from TableName

Example:

- SQL statement to display all the information of all employees.

select * from employee

- SQL statement to display firstname all employees.

select first_name,last_name from employees

2. **Insert**: Creates a record

Syntax: Insert into TableName (Column1, …, ColumnN) values (Column1 Values, …, ColumnN Values)

Or

Insert into TableName Values (Column1 Values, …, ColumnN Values)

Example:

- Insert record in dept

insert into tblDepartment values(107,'HR','2011-02-01')

- Insert record in dept with column names

insert into tblDepartment(Deptid,Dname,Location) values(102,'Development','Mumbai')

- Multiple Insertion

insert into tblDepartment(Deptid,Dname,YearofEstablishment,Location) values (103,'Sales','2011-02-10','Pune'), (104,'Finance','2012-03-15','Chennai')

- Inserting null values external

insert into tblDepartment(Deptid,Dname,YearofEstablishment,Location) values (105,'Admin',null,'Mumbai')

- Error cant insert duplicate value in primary key column

insert into tblDepartment(Deptid,Dname,YearofEstablishment,Location) values (105,'UX',null,'Mumbai')

- new syntax

insert tblDepartment(Deptid,Dname,YearofEstablishment,Location) values (106,'UX','2012-05-28','Mumbai')

3. **Update**: Modifies the record

Syntax: Update TableName Set Column1 = Value1, Column2 = Value2, …Where Condition

Example:

- updating individual record

update tblDepartment set yearofestablishment='2011-08-20' where Deptid=102

- updating all records in particular column

update tblDepartment set yearofestablishment='2011-08-20'

<u>4.</u> **Delete:** Delete Records

<u>Syntax:</u> delete from TableName

Or

deleting particular record - delete from tablename where condition

<u>Example:</u>

- delete from sample
- delete from tblDepartment where Deptid=106

5. Views

- It is a virtual table whose contents are defined by a query.
- Like a table, a view consists of a set of named columns and rows of data. But, a view does not exist as a stored set of data values in a database.
- The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.
- A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the current or other databases.
- Can't use "Order by" In view
- Best practice to use view is for DQL purpose only
- Better to don't update the view

<u>Why View</u>

- Focus the Data for Users-Important or appropriate data
- Limit access to sensitive data
- Hide complex database design
- Simplify complex queries including distributed queries to Heterogeneous data
- Simplify Management of User Permissions

Create view

```
Create view Viewname

As

Select [Top, Bottom with Number of Rows] Column1, Column2, ……Column N

From Table1 T1

inner join Table2 T2

on T1.Column=T2.Column
```

inner join

.

.

. [Can use join to join Tables or Where for two table]

Where condition

Group by TableName.ColumnName

Having condition

Alter view

Alter view Viewname

As

Select [Top, Bottom with Number of Rows] Column1,Column2,……Column N

From Table1 T1

inner join Table2 T2

on T1.Column=T2.Column

inner join

.

.

. [Can use join to join Tables or Where for two table]

Where condition

Group by TableName.ColumnName

Having condition

Example:

create view v_deptmaxsalary

as

select tblDepartment.Dname ,max(e.salary) 'Max Salary'

from tblDepartment ,tblemployee e

where tblDepartment.Deptid=e.did

group by tblDepartment.Dname

having max(e.salary) >25000

Drop view

```
drop view  Viewname
```

<u>View with out check</u>

```
alter view v_agecheck
as
select name,age from sample
where age>25
```

Insert

```
insert into v_agecheck(name,age) values('hari',23)
```

- Insert takes place into base original table into database

View with check option

```
alter view v_agecheck
as
select name,age from sample
where age>25 with check option
```

Error

```
insert into v_agecheck(name,age) values('vimal',22)
```

Insert

```
insert into v_agecheck(name,age) values('vimal',26)
```

<u>Few things to remember:</u>

A) Define View

1. View can be created to retrieve data from one or more tables.
2. Query used to create view can include other views of the database.
3. We can also access remote data using distributed query in a view.

B) What functions can a view be used to performed?

1. Subset data of a table.
2. Can join multiple tables values into one.
3. They can act as aggregated tables. i.e. a view can be used to store Sum, average of values.
4. Views can be nested and can be used for abstraction.

C) What are the restrictions that views must follow?

1. Since a view is a virtual table – columns of the view cannot be renamed. To change anything in the view, the view must be dropped and create again.
2. The select statement on the view cannot contain ORDER BY.
3. When a table or view is dropped, any views in the same database are also dropped.
4. It is not possible to use DELETE to update a view that is defined as a join.

D) Can we create a view based on other views?

Ans) Yes, we can create a view based on other views. Usually, we create views based on tables, but it is also possible to create views based on views.

E) What are the limitations of a view?

1. We cannot pass parameters to a view.
2. Rules and Defaults cannot be associated with views.
3. The ORDER BY clause is invalid in views unless TOP or FOR XML is also specified.

6. Basic

Introduction to DBMS

SQL(Structured Query Language) is a language to operate databases; it includes database creation, deletion, fetching rows, modifying rows, etc. SQL is an ANSI (American National Standards Institute) standard language, but there are many different versions of the SQL language.

Definition:

- Database can be defined as the storage of interrelated data that has been organized in such a fashion that the process of retrieving data is effective andefficient

- DBMS contains information about a particular enterprise
- Collection of interrelated data
- Set of programs to access the data
- An environment that is both *convenient* and *efficient* touse.

Limitation of Traditional File Based systems:

- Data redundancy
- Inconsistant data
- Difficulty in accessing data
- Limited data sharing
- Concurrent access
- Atomicity

Acid properties:

- a-atomicity
- c-consistency
- i-isolation
- d-durability

Data Models

- According to Hoberman(2009),

"A data model is a way of finding the tools for both business and IT professionals, which uses a set of symbols and text to precisely explain a subset of real information to improvecommunication within the organization and thereby lead to a more flexible and stable application environment"

A data model is an idea which describes how the data can be represented and accessed from software system after its complete implementation

- It is a simple abstractionof complex real world data gathering environment

- It defines data elements and relationships among various data elements for a specified system

- The main purpose of data model is to give an idea that how final system or software will look likeafter development is completed

Types

- Hierarchical DBMS
- Network DBMS
- Relational DBMS
- Object Relational DBMS
- Object Oriented DBMS

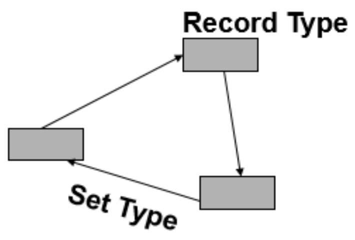
Hierarchical Data Model

- A hierarchical data model an be definied as a model that organizes data in a hierarchical tree structure

- Description
- A hierarchical tree structure is made up of nodes and branches
- The dependent nodes are at lower levels in the tree

Network Data Model

- The network data model can be defined as interconnects the entities of an enterprise into a network



Description

• A block represents an entity or record type. Each record type is composed of zero, one, or more attributes

1:N Relationship

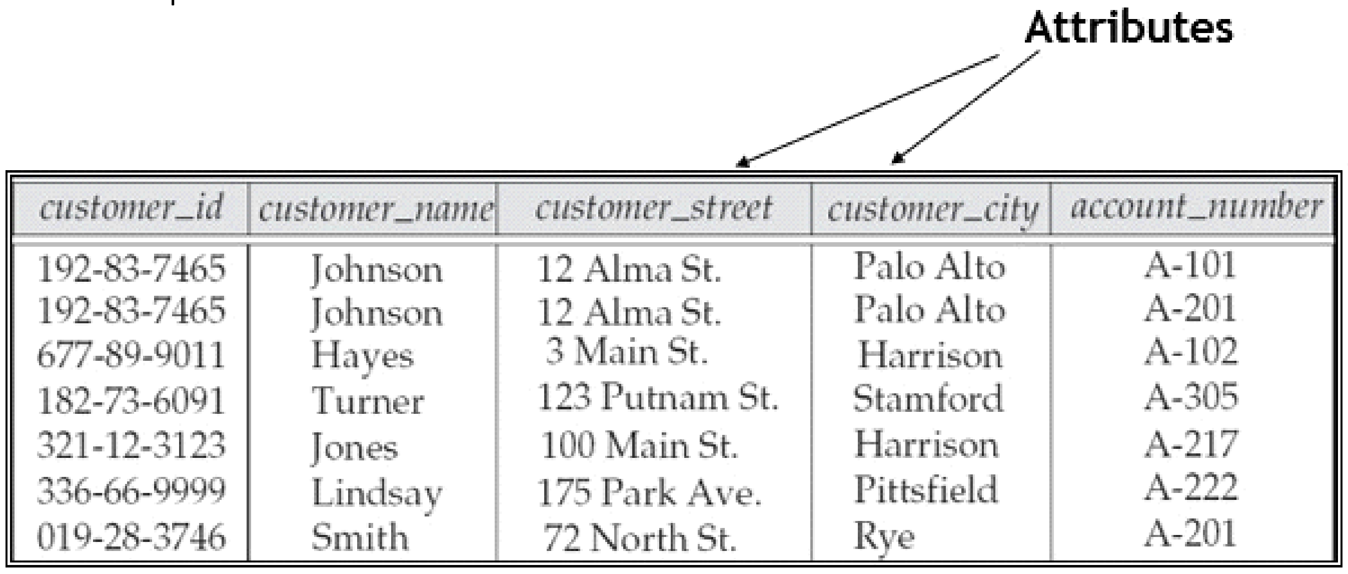
• An owner record type owns zero, one, or many occurrences of a member record type

M:N Relationship :

A many to many relationship can be implemented by creating two one-to-many relationship a third entity type called connector record type has two owner record type

Relational model

• Example of tabular data in the relational model


Attributes

customer_id	*customer_name*	*customer_street*	*customer_city*	*account_number*
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

Database Design

The process of designing the general structure of the database:

• Logical Design – Deciding on the database schema. Database design requires that we find a "good" collection of relation schemas.

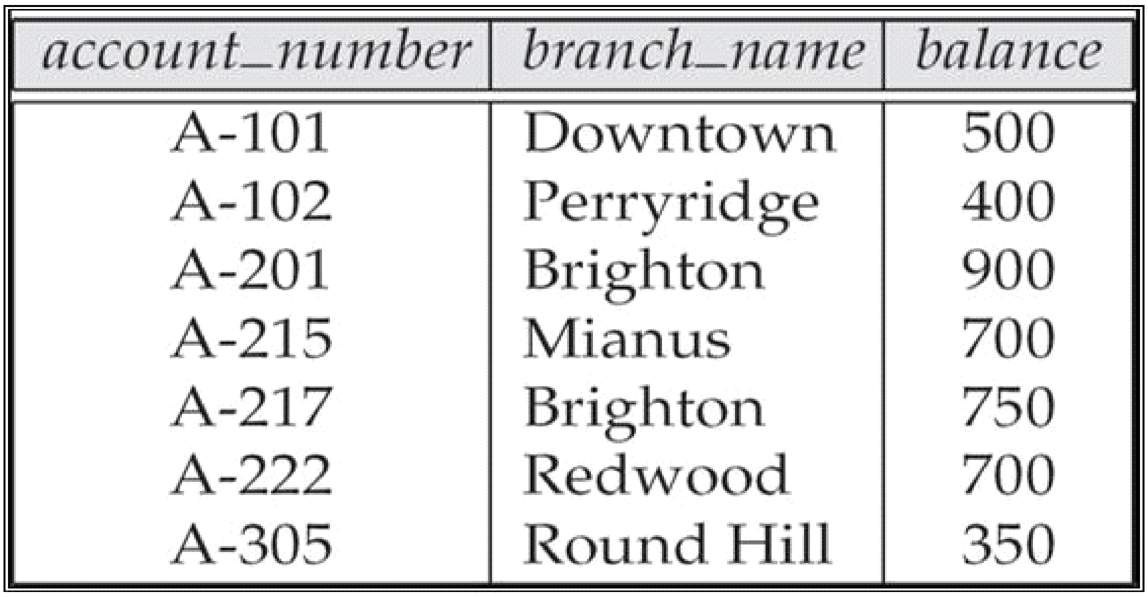
- Business decision – What attributes should we record in the database?
- Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?

- Physical Design – Deciding on the physical layout of the database

Relational Model

- Structure of Relational Databases
- Fundamental Relational-Algebra-Operations
- Additional Relational-Algebra-Operations
- Extended Relational-Algebra-Operations
- Null Values
- Modification of the Database

Example of a Relation



account_number	*branch_name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

7. Stored Procedure

THEORY
QUERY PLAN
A query plan (or query execution plan) is a sequence of steps used to access data in a SQL relational database management system.
Because query optimizers are imperfect, database users and administrators sometimes need to manually examine and tune the plans
produced by the optimizer to get better performance.

STORED PROCEDURE-

1) A stored procedure is a saved set of instructions in sql. If you have a situation, where you write the same
query over and over again, you can save that specific query as a stored procedure and cal it just by it's name.
Stored Procedures are created to perform one or more DML operations on Database.

2) It is nothing but the group of SQL statements that accepts some input
in the form of parameters and performs some task and may or may not returns a value.

For stored procedure it reuses the query plan once defined . Otherwise query plan value creates everytime execution happens.

IMPORTANT BULLETINS FOR STEPS TO USE STORED PROCEDURE:

1) <u>Syntax :</u> Creating a Procedure

```
CREATE or REPLACE PROCEDURE name(parameters)
IS
variables;
BEGIN
//statements;
END;
```

2) Use CREATE PROCEDURE or CREATE PROC statement to create SP

3) It can be called with or without parameter. Using parameter we can call out specific data of that parameter.

4)To execute the stored procedure

a) spGet(spname)
b) EXECspGet(spname)
c) ExecutespGet(spname)

Note- you can also right click on the procedure name, in object explorer in SQL Server Management Studio and select EXECUTE STORED PROCEDURE
------------------CODE WITHOUT PARAMETER--

1)

```
--without parameter
create procedure pro_getsongs
as
begin
	SELECT TOP 3 songs + ' '+ musicname + ' ' + artistname AS songname
	FROM Library.Songs
end
```

```
--execute
exec pro_getsongs
```

STORED PROCEDURE WITH PARAMETER

The most important part is parameters. Parameters are used to pass values to the Procedure. There are 3 different types of parameters, they are as follows:

IN:
This is the Default Parameter for the procedure. It always recieves the values from calling program.
OUT:
This parameter always sends the values to the calling program.
IN OUT:
This parameter performs both the operations. It receives value from as well as sends the values to the calling program.

```
---------------CODE FOR PARAMETER----------------------------------------------------
--With parameter
--Example - 1
create proc proc_Flowers(@id char)
as
begin
select * from tblFlowers where Fname=@id
end


create procedure proc_join(@age int)
as
begin
 select f.fname,b.Age,d.datebought
 from tblflower f
 inner join
 tblDurability b
 on f.Fname=p.Fname
 inner join
 tblDepartment d
 on f.did=d.deptid
 where age<@age
 order by Fname
 end

 proc_join @age=5
```

select * from tblDurability

8. 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.

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

SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

<u>Syntax</u>

```
CREATE TABLE table_name (

column1 datatype constraint,

column2 datatype constraint,

column3 datatype constraint,

.... );
```

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

SQL NOT NULL on CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

```
CREATE TABLE Persons ( ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255) NOT NULL,
```

SQL NOT NULL on ALTER TABLE

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

ALTER TABLE Persons

Age int NOT NULL;

SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

CREATE TABLE Persons (ID int NOT NULL UNIQUE, LastName

varchar(05) NOT NULL, FirstName varchar(20), Age int);

SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

ALTER TABLE Persons

ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

ALTER TABLE Persons

DROP CONSTRAINT UC_Person;

SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

CREATE TABLE Persons (ID int NOT NULL UNIQUE, LastName

varchar(05) NOT NULL, FirstName varchar(20), Age int);

SQL PRIMARY KEY on ALTER TABLE

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Persons

ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);

DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

ALTER TABLE Persons

DROP CONSTRAINT PK_Person;

SQL FOREIGN KEY

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

Look at the following two tables:

"Persons" table:

Person ID Last name First name Age

1 Sobti Barun 32

2 Mehta Nakuul 35

3 Shiek Shaheer 33

"Persons" table:

OrderId OrderNo PersonID

1

77895 3

2 44678 3

3 22456 2

4 24562 1

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

```
CREATE TABLE Orders ( OrderNumber int NOT NULL, PersonID int FOREIGN KEY REFERENCES
Persons(PersonID) );
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Orders (

OrderID int NOT NULL,

OrderNumber int NOT NULL,

PersonID int,

PRIMARY KEY (OrderID),

CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)

REFERENCES Persons(PersonID));
```

SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

```
ALTER TABLE Orders

ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Orders

ADD CONSTRAINT FK_PersonOrder

FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE Orders

DROP CONSTRAINT FK_PersonOrder;
```

SQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

(i)Using CHECK on create table:

Applying CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

```
CREATE TABLE Persons (ID int NOT NULL,LastName varchar(255) NOT NULL,FirstName varchar(255),Age int CHECK(Age>=18));
```

(ii) CHECK constraint on create table:

```
CREATE TABLE Persons (ID int NOT NULL,LastName varchar(255) NOT NULL, FirstName varchar(255),Age int CONSTRAINT CHK_Person CHECK(Age>=18));
```

(iii)CHECK on Alter table:

```
ALTER TABLE Persons ADD CHECK(Age>=18);
```

(iv) CHECK Constraint on Alter table:

```
ALTER TABLE Persons ADD CONSTRAINT CHK_PersonAge CHECK(Age>=18);
```

(v)Drop a CHECK constraint

```
ALTER TABLE Persons DROP CONSTRAINT CHK_PersonAge;
```

SQL DEFAULT Constraint

The DEFAULT constraint is used to provide a default value for a column.

The default value will be added to all new records IF no other value is specified.

(i)DEFAULT on create table:

Applying DEFAULT value for the "City" column when the "Persons" table is created:

```
CREATE TABLE Persons (ID int NOT NULL,LastName varchar(255) NOT NULL, FirstName varchar(255),Age int,City varchar(255) DEFAULT 'Sandnes');
```

(ii) DEFAULT Constraint on alter table:

```
ALTER TABLE Persons ADD CONSTRAINT df_City DEFAULT 'Sandnes' FOR City;
```

(iii) Drop a DEFAULT Constraint:

```
ALTER TABLE Persons ALTER COLUMN City DROP DEFAULT;
```

Integrity Constraints

Integrity constraints are used to ensure accuracy and consistency of the data in a relational database.

Types of integrity constraints include Primary Key, Foreign Key, Unique Constraints and other constraints which are mentioned above.

Disabling and Enabling Constraints

Enabing and disabling constraints works only to check constraints and foreign key constraints.

(i)Disable all table constraints: ALTER TABLE YourTableName NOCHECK CONSTRAINT ALL

(ii) Enable all table constraints: ALTER TABLE YourTableName CHECK CONSTRAINT ALL (iii)Disable single constraint: ALTER TABLE YourTableName NOCHECK CONSTRAINT YourConstraint

(iv)Enable single constraint: ALTER TABLE YourTableName CHECK CONSTRAINT YourConstraint

SQL AUTO INCREMENT

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

(i)AUTO INCREMENT on create table:

The following SQL statement defines the "Personid" column to be an auto-increment primary key field in the "Persons" table:

CREATE TABLE Persons (Personid int IDENTITY(1,1) PRIMARY KEY,LastName varchar(255) NOT NULL,FirstName varchar(255),Age int);

9. **Where and Have**

Where

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

Syntax:

SELECT *column1*, *column2, ...* FROM *table_name* WHERE *condition*

Operators used in the WHERE clause:

Operator	**Description**

=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	l. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern(Strings)
IN	To specify multiple possible values for a column

Example:

From the EMPLOYEE table

1. Select the Employee name and salary whose salary = 40000
```
select Ename, salary from tblEmployee where salary = 40000
```
2. Select the Employee name and salary whose salary < 40000
```
select Ename, salary from tblEmployee where salary < 40000
```
3. Select the Employee name and salary whose salary < 40000
```
select Ename, salary from tblEmployee where salary < 40000
```
4. Select the Employee name and salary whose salary >= 35000
```
select Ename, salary from tblEmployee where salary >= 35000
```
5. Select the Employee name and salary whose salary <= 40000
```
select Ename, salary from tblEmployee where salary <= 40000
```
6. Select the Employee name and salary whose salary between 35000 and 50000
```
select Ename, salary from tblEmployee where salary between 35000 and 50000
```
7. Select the Employee name and salary whose name has 's'
```
select Ename, salary from tblEmployee where Ename like '%s%'
```
8. Select the Employee name and salary whose salary is not equal to 35000
```
select Ename, salary from tblEmployee where salary !=35000
```

From the DEPARTMENT table

9. Select the Department name and Location whose location is in Chennai, Pune

```
select Dname, Location from tblDepartment where Location in ('chennai','pune')
```

Having:

- Having works with aggregate function which allows to use aggregate function in conditional part

<u>Example:</u>

Select the Department ID and Minimum salary from Employee table whose minimum salary is >40000

From EMPLOYEE table

```
select did,min(salary)
from tblEmployee
group by did
having min(salary)>40000
```

<u>Example for WHERE and HAVING:</u>

Display the department id, minimum salary whose minimum salary is >40000 and employee name ends with 'i'

```
select did, min(salary) from tblEmployee where ename like '%i'
group by did having min(salary)>40000
```

Points to ponder:

1. WHERE cannot be used with aggregate functions whereas HAVING can be used with aggregate functions. This means WHERE is used for filtering individual rows on a table whereas HAVING is used to filter groups.

2. WHERE comes before GROUP BY. HAVING comes after GROUP BY.

3. Performance point of view, HAVING is slower than WHERE

4. WHERE and HAVING can be used together in a SELECT query. In this case WHERE is applied first to filter individual rows. The rows are then grouped and aggregate calculations are performed and then the HAVING filters the groups.

5. WHERE can be used with-Select, Insert and update statements whereas HAVING can only be used with the Select statement.

10. Aggregate Functions

An aggregate function performs a calculation on a set of values, and returns a single value. Except for COUNT(*), aggregate functions ignore null values. Aggregate functions are often used with the GROUP BY clause of the SELECT statement. We cannot use aggregate functions in WHERE clause.
SQL provides the following aggregate functions:

1. AVG
2. COUNT
3. MAX
4. MIN
5. SUM
6. STDEV

Id	Name	Salary	Location	Department
1	Swaroop	33000	Hyderabad	Admin
2	Saketh	35000	Mumbai	Hr
3	Madhav	32000	Mumbi	Payroll
4	Satweek	34000	Hyderabad	Hr
5	Manoj	36000	Chennai	Manager
6	Sharif	45000	Pune	engineer

7. VAR

Table name : tblemployee

AVG: This function returns the average of the values in a group. It ignores null values.

SYNTAX: select AVG(**column_name**) *from* **table_name**.

Example: select AVG(**Salary**) *from* **tblemployee**

Output: 35833

AVERAGE WITH GROUP BY CLAUSE

```
Select Location, AVG(Salary) as [avg salary] from tblemployee
  Group by Location
```

Output:

	location	Avg salary
1	Chennai	36000
2	Hyd	33500
3	Mumbai	33500
4	pune	45000

COUNT: This function returns the number of items found in a group.COUNT always returns an int data type value. Count excludes null value. Count will not give distinctive information.

SYNTAX: select COUNT(**column_name**) *from* **table_name**

EXAMPLE: select COUNT(**Id**) *from* **tbltable**

Output: 6

MAX: Returns the maximum value in the expression.

SYNTAX: select MAX(**column_name**) *from* **table_name**

EXAMPLE: select MAX(**salary**) *from* **tbltable**

Output: 45000

MIN: Returns the minimum value in the expression.

SYNTAX: select MIN(**column_name**) *from* **table_name**.

EXAMPLE: select MIN(**salary**) *from* **tbltable**

Output: 32000

SUM: Returns the sum of all the values, or only the DISTINCT values, in the expression. SUM can be used with numeric columns only. Null values are ignored.

SYNTAX: select SUM(**column_name**) *from* **table_name**

EXAMPLE : select SUM(**salary**) *from* **tbltable**

Output: 215000

STDEV: Returns the statistical standard deviation of all values in the specified expression.

If STDEV is used on all items in a SELECT statement, each value in the result set is included in the calculation. STDEV can be used with numeric columns only. Null values are ignored

SYNTAX: select STDEV(**column_name**) *from* **table_name**

EXAMPLE : select STDEV(**salary**) *from* **tbltable**

OUTUT : 4708.1489

VAR : Returns the statistical variance of all values in the specified expression

If VAR is used on all items in a SELECT statement, each value in the result set is included in the calculation. VAR can be used with numeric columns only. Null values are ignored.

SYNTAX: select VAR(**column_name**) *from* **table_name**

EXAMPLE : select VAR(**salary**) *from* **tbltable**

OUTPUT : 22166666.66666

Use aggregate functions as expressions only in following situation

- The select list of a SELECT statement (either a subquery or an outer query).

QUESTION: display salary of employee where salary is greater than average salary of all employees?

Select salary from **tbltable** where **salary**> avg**(salary**)

The above expression will not execute in SQL and shows error like

''An aggregate may not appear in the WHERE clause unless it is in a subquery contained in a HAVING clause or a select list, and the column being aggregated is an outer reference''.

To avoid that we use following expression
select salary from **tbltable** where **salary**>(select AVG(**salary**) from **tbltable**)

11. Operators (Transact – SQL)

An operator is a symbol specifying an action that is performed on one or more expressions.

Arithmetic Operators:

Arithmetic operators run mathematical operations on two expressions of one or more data types

Operator	Meaning	Syntax
+ (Add)	Addition	expression + expression
- (Subtract)	Subtraction	expression - expression
* (Multiply)	Multiplication	expression * expression
/ (Divide)	Division	dividend / divisor
% (Modulo)	eturns the integer remainder of a division. For mple, 12 % 5 = 2 because the remainder of 12 divided by 5 is 2.	dividend % divisor

The plus (+) and minus (-) operators can also be used to run arithmetic operations on **datetime** and **smalldatetime** values.

Logical Operators :

Logical operators test for the truth of some condition. Logical operators, like comparison operators, return a **Boolean** data type with a value of TRUE, FALSE, or UNKNOWN.

Operator	Meaning	Syntax
ALL	JE if all of a set of comparisons are TRUE.	pression { = | <> | != | > | >= | | < | <= | !< } ALL (subquery)
AND	JE if both Boolean expressions are TRUE.	boolean_expression AND boolean_expression
ANY	ny one of a set of comparisons are TRUE.	_
BETWEEN	TRUE if the operand is within a range.	st_expression [NOT BETWEEN pression AND end_expression
EXISTS	TRUE if a subquery contains any rows.	EXISTS (subquery)
IN	if the operand is equal to one of a list of expressions.	test_expression [NOT] IN subquery | expression [,...n])
LIKE	TRUE if the operand matches a pattern.	ax for Azure Synapse Analytics and Parallel Data Warehouse xpression [NOT] LIKE pattern
NOT	the value of any other Boolean operator.	[NOT] boolean_expression

OR	RUE if either Boolean expression is TRUE.	boolean_expression OR boolean_expression
SOME	f some of a set of comparisons are TRUE.	pression { = | < > | ! = | > | > = | ! > | < | < = | ! < } { SOME | ANY } (subquery)

Comparison Operators :

Comparison operators test whether two expressions are the same. Comparison operators can be used on all expressions except expressions of the **text, ntext, or image** data types

Operator	Meaning	Syntax
= (Equals)	Equal to	pression = expression
> (Greater Than)	Greater than	pression > expression
< (Less Than)	Less than	pression < expression
reater Than or Equal To)	eater than or equal to	pression >= expression
(Less Than or Equal To)	ess than or equal to	pression <= expression
<> (Not Equal To)	Not equal to	pression <> expression
!= (Not Equal To)	Not equal to	_
!< (Not Less Than)	Not less than	pression !< expression
> (Not Greater Than)	Not greater than	pression !> expression

Compound Operators :

Compound operators execute some operation and set an original value to the result of the operation. For example, if a variable @x equals 35, then @x += 2 takes the original value of @x, add 2 and sets @x to that new value (37).

Operator	Meaning	Syntax
signment)	amount to the original value ie original value to the result.	+= expression
t Assignment)	ome amount from the lue and sets the original value ilt.	-= expression
ly Assignment)	by an amount and sets the lue to the result.	*= expression
ssignment)	an amount and sets the lue to the result	/= expression
s Assignment	an amount and sets the lue to the modulo.	%= expression
AND Assignment)	bitwise AND and sets the lue to the result	&= expression
Exclusive OR	bitwise exclusive OR and iginal value to the result.	^= expression
OR Assignment)	bitwise OR and sets the lue to the result.	|= expression

Unary Operators :

Returns the negative of the value of a numeric expression (a unary operator). Unary operators perform an operation on only one expression of any one of the data types of the numeric data type category.

Operator	Meaning	Syntax
+ (Positive)	meric value is positive.	numeric_expression
- (Negative)	neric value is negative.	numeric_expression
~ (Bitwise NOT)	he ones complement of the number.	~ expression

Bit Operators

Operator	Meaning	Syntax
&	Bitwise AND	_
|	Bitwise OR	_
^	itwise Exclusive OR	_

Set Operators

Operator	Meaning	Syntax

Union	wo or more result sets into a set, without duplicates.	uery_specification> | (query_expression>) } { UNION uery_specification> | (query_expression>) }
Union All	wo or more result sets into a et, including all duplicates.	uery_specification> | (query_expression>) } { ALL uery_specification> | (query_expression>) }
Intersect	e data from both result sets hich are in common.	uery_specification> | (query_expression>) } { INTERSECT } uery_specification> | (query_expression>) }
Except	data from first result set, but not the second	uery_specification> | (query_expression>) } { EXCEPT } uery_specification> | (query_expression>) }

12. INDEX

Q How will the database engine retrives the information from a table:

Whenever the database engine want to retrive the information from the table it will adopt two different mechanism for searching the data.

→ Full page scan
→ Index scan

→ In the first case sql server will search for the required information in each and every page to collect the information. So, if the table has more rows it will be taking lot of time for scanning all the data. so it is a time consuming process.

→ In the second case sql server without searching into each and every datapage for retriving the information it will make use of an index for retriving the information, where an index is a pointer to the information what we are retrive which can reduce the disk I/O operation saving the time, But if we want to use index scan for searching the data first the index has to be created.

Syntax:- Create [unique] [clustered / Non-clustered],
Index <Index Name> on <TableName>
(<collist>)

Note:- Whenever an index is created on a column or columns of a table internally an index table gets created maintaining the information of a column on which the index is created as well as address (pointer to the row corresponding to a column)

Index Table

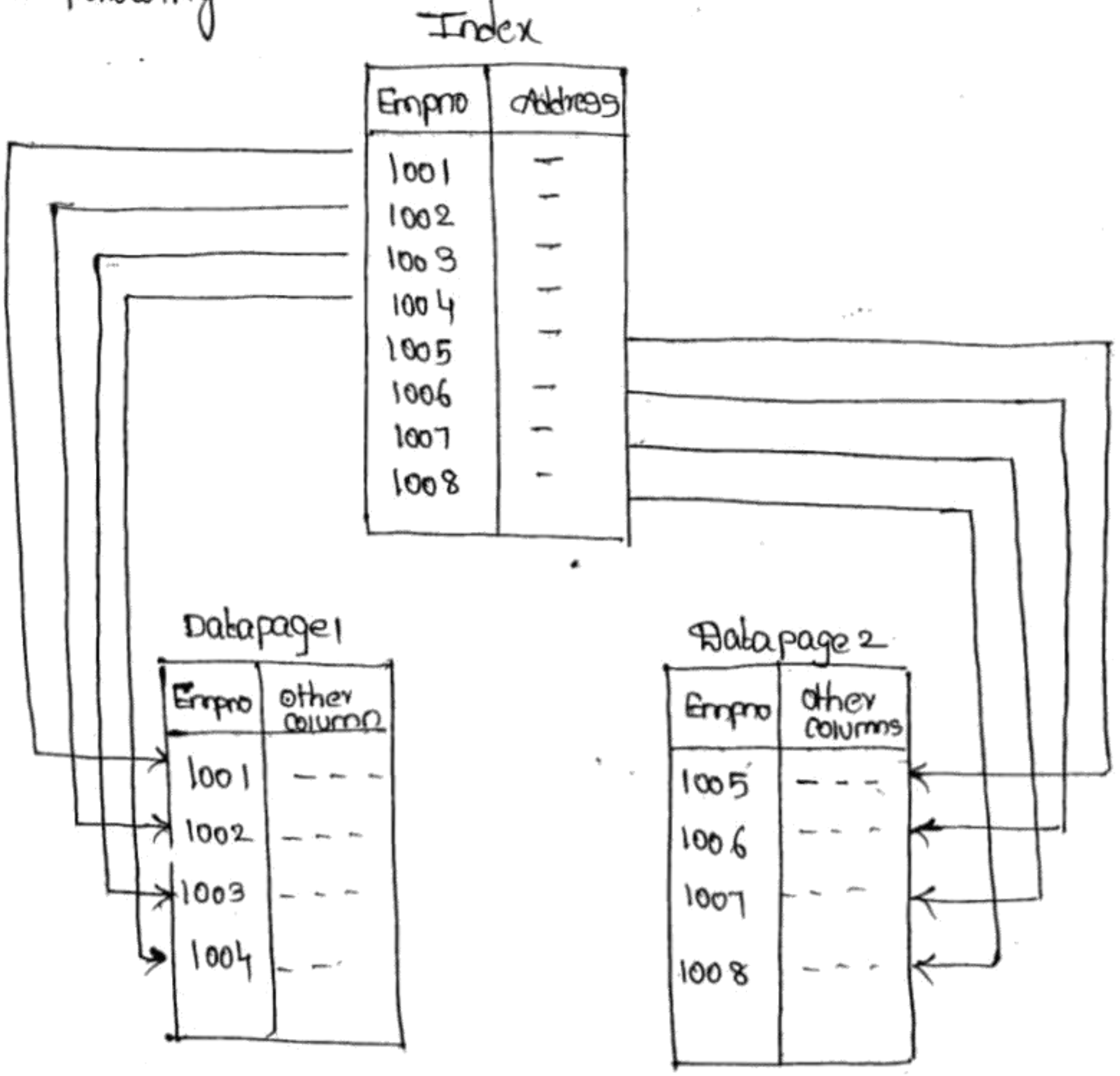
<collist>	Address

__CLUSTERED__ __INDEX__ :- In this case the arrangement of the data in the index table will be same as arrangement of the data of actual table.

Eg:- The index we find in the start of a book.

__Note__ :- Indexes will arrange the information under them by adopting a structure called B-tree structure.

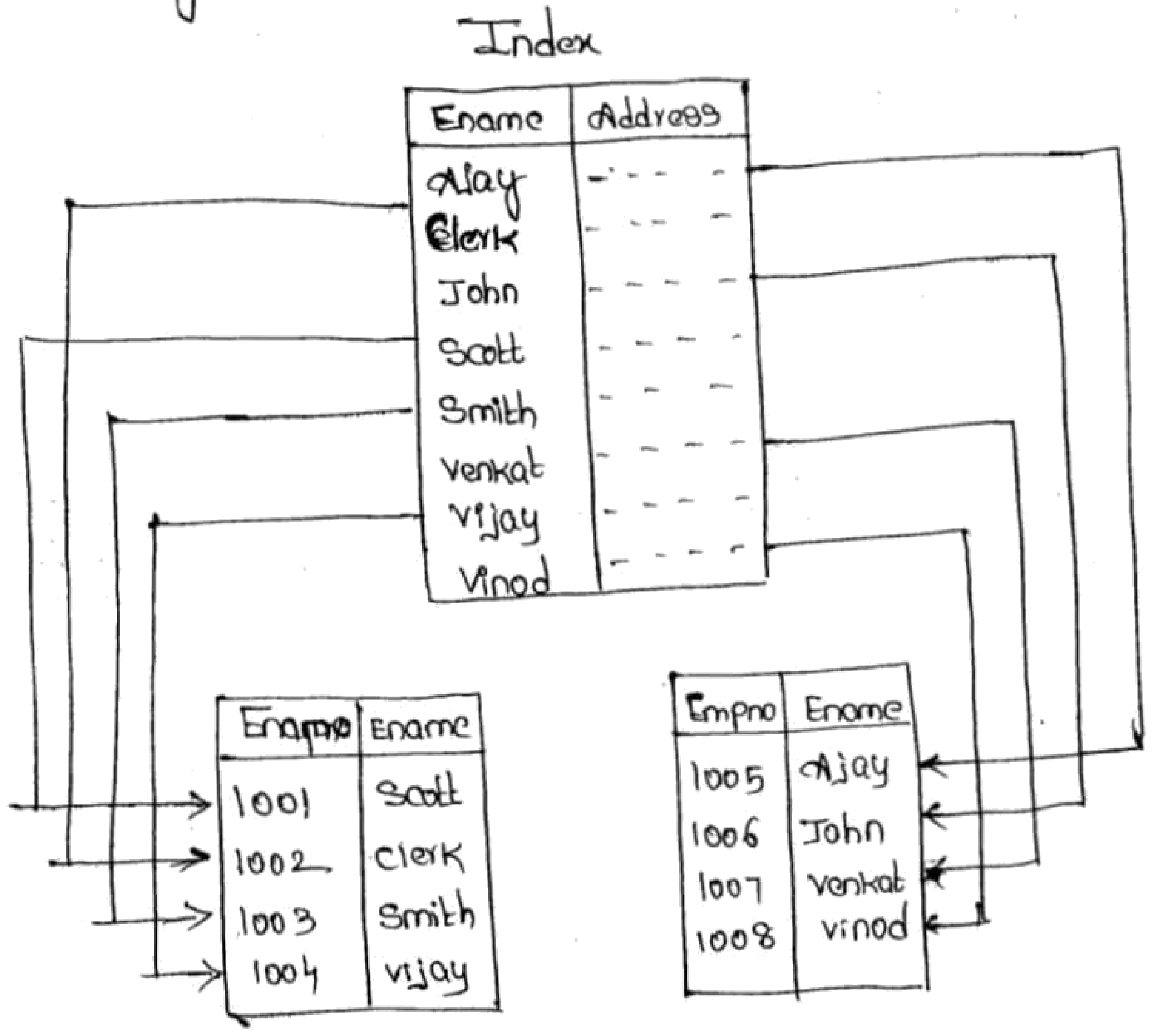
→ Suppose if a clustered index is created on the Employee no. column of Emp table internally we will find the information as following.



NON CLUSTERED INDEX: In this case the arrangement of data in the index page or table will be different from the arrangement of in actual table.

Eg:- The index we find in the end of the book.

Suppose a nonclustered index is created on Ename column of Emp table the arrangement of the data will be as following.



Ename	Address
Ajay	----
Clerk	----
John	----
Scott	----
Smith	----
Venkat	----
Vijay	----
Vinod	----

Empno	Ename
1001	Scott
1002	Clerk
1003	Smith
1004	Vijay

Empno	Ename
1005	Ajay
1006	John
1007	Venkat
1008	Vinod

Unique Index:- If the index is created by using "unique" option that column on which the index is created will not allow duplicate values i.e it works as a "unique constraint". unique constraint can be either unique clustered or unique non-clustered also.

Note:- While creating an index if clustered or non-clustered is not specified. Default is non-clustered.

Q:- How many Indexes a table can have?

We can create a maximum of 250 indexes in which only one can be clustered and remaining all are nonclustered

Are indexes created Implicitly or should be create them Explicitly.

When ever we impose a primary key constraint on a tables column internally a "unique clustered" index gets created. Where as if a unique constraint is impose on any column internally a "unique nonclustered" index gets created.

→ Creating indexes on Emptable:-

As the Empno column is imposed with a PK it will contain a unique clustered index implicitly Apart from that we can create any number of nonclustered indexes on the table as following.

Eg: create Nonclustered index Ename-Ind on Emp (Ename)
    Create index Sal-ind on Emp (sal)

Eg:- Create table Student (sid int, sname varchar(50))
    Create unique clustered index Sid_Ind on
    Student (sid)

Note:- In the above case the index will not provide the functionality of Primary key for Sid column but still we will get the functionality of unique constraint.

When SQL Server uses Indexes:-

Sqlserver uses indexes of a table provided the select or update or delete statement contained "where" condition in them and moreover the where condition column must be a indexed column.

If the select statement contain an "order by" clause also indexes will use.

Note:- when sqlserver is searching for information under the database first it verifies the best execution plan for retriving the data and uses that plan which can be either a full page scan or an index scan also.

```
Eg:- Select * From Emp ✗
     Select * From Emp Where Empno = 1003 ✓
     Select * From Emp where Ename = 'scott' ✓
     Select * From Emp Where Sal >3000 ✓
     Select * From Emp where ~~Soundex~~ job='manager' ✗
     Select * From Emp Where Soundex (Ename) = soundex('Smyth')
     Select * From Emp order By Sal Desc                      ✗
```

When Should we create Indexes on a Table:

We need to create index on a table columns provided those columns are frequently used in "where condition" or "order By clause".

-> It is not adviced creating an index on each and every column because more number of indexes can degrade the performance of database also because every modification we make on the data should be reflected into all the index tables.

13. KEYS

DEFINITION: A key is a single or combination of multiple fields in a table. It is used to fetch or retrieve records/data-rows from data table according to the condition/requirement. Keys are also used to create a relationship among different database tables or views.

Different keys in SQL SERVER: SQL Server supports various types of keys, which are listed below:

a. Candidate Key
b. Primary Key
c. Unique Key
d. Composite Key
e. Foreign Key

a) Candidate Key: Candidate key is a key of a table which can be selected as a primary key of the table. A table can have multiple candidate keys, out of which one can be selected as a primary key. Example: Employee_Id, License_Number and Passport_Number are candidate keys
b) Primary Key: Primary key is a candidate key of the table selected to identify each record uniquely in table.Primary key does not allow null value in the column and keeps unique values throughout the column.

A table can have only one primary key.

Example -Employee_Id is a primary key of Employee table.

SYNTAX FOR ADDING PRIMARY KEY:

```
CREATE TABLE Employee( employee_id int PRIMARY KEY,employee_name char(50) , contact_name char(50));
```

Adding Primary after table creation:

```
  Alter Table (table name) alter  column (columnname) not null

  Alter table  (table name) add primary key(Column name)
```

SYNTAX FOR DELETING PRIMARY KEY:

```
 alter table (table name) drop constraint pk_id

alter table (table name) drop column id
```

c) Foreign Key: In a relationship between two tables, a primary key of one table is referred as a foreign key in another table. Foreign key can have duplicate values in it and can also keep null values if column is defined to accept nulls.
SYNTAX FOR ADDING FOREIGN KEY:

CREATE TABLE (table2 name)

ADD FOREIGN KEY (Column name) REFERENCES (table 1)(column(table1));

SYNTAX FOR DROPING FOREIGN KEY:

ALTER TABLE table_name

DROP CONSTRAINT fk_name;

d) Unique Key: Unique key is similar to primary key and does not allow duplicate values in the column. It allows one null value in the column.
SYNTAX FOR ADDING UNIQUE KEY:

CREATE TABLE tablename

ADD UNIQUE (ID);

SYNTAX FOR DROPING UNIQUE KEY:

ALTER TABLE Persons

DROP INDEX UC_Person;

e) Composite Key: Composite key (also known as compound key or concatenated key) is a group of two or more columns that identifies each row of a table uniquely. Individual column of composite key might not able to uniquely identify the record. It can be a primary key or candidate key also.

14. GROUP BY AND ORDER BY CLAUSE

Order By Clause :

Syntax :

```
/*SELECT column-list
FROM table name
[WHERE condition]
[ORDER BY col1, co12,co13,…] [ASC | DESC];  Default is Ascending order unless specified.
*/
```

Order the result of a query by the specified column list. The order in which rows are returned in a result set are not guaranteed unless an “Order by” clause is specified.

Order_by_expression:

Specifies a column or expression on which to sort query result set. The column to be sort can be specified as a name or alias name or a non negative number representing the position of column in select list.

Multiple columns can be specified in sort list. Ambiguity in name or alias names must be excluded. The sequence of columns in sort list defines the organization of the sorted result set.

The result set is sorted by the first column and then then that ordered list is sorted by the second column so on.

Best Practices:

Avoid specifying integers in order by clause as positional representations of columns in select list. Although they are correct, any change in column selections or order of selections total column positional representations in integers must be changed.

In “select Top (N)” kind of statement, always include order by clause. This is only way to predictably indicate which rows are affected by using TOP.

Limitations:

Columns of type ntext, text, image, geography, geometry cannot be used in an Order by clause.

If a table name is aliased in the FROM clause, then only alias names is used to qualify its columns in Order by clause.

<u>Examples.</u>

1. Specifying a single column in defined select list

```
SELECT ProductID, Name FROM Production.Product

WHERE Name LIKE 'Lock Washer%'

ORDER BY ProductID
```

2. Specifying column not in the select list

```
SELECT ProductID, Name, Color

FROM Production.Product

ORDER BY ListPrice
```

1. Specifying an alias as sort column

```
SELECT name, SCHEMA_NAME(schema_id) AS SchemaName

FROM sys.objects

WHERE type = 'U'

ORDER BY SchemaName
```

2. Specifying an expression as the sort column

```
SELECT BusinessEntityID, JobTitle, HireDate

FROM HumanResources.Employee
```

```
ORDER BY DATEPART(year, HireDate)
```

Group By clause:

```
/*
Groupby
--------
SELECT column1,.., columnN
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
*/
```

A select statement clause that divides the query result into groups of rows , usually for performing one or more aggregations on each group.

Column - expression

Specifies a column or non-aggregate calculation on a column. This column can belong to a table or view. The column must appear in the From clause of the select statement, but is not required to appear in the select list.

The column expression cannot contain the following :

1. A column type of text, ntext or image.
2. A subquery

Group by interaction with select statement :

1. If aggregates are included in select list. Group by calculates summary values of each group.
2. SQL removes rows that do not meet the conditions given in Where clause before any grouping operation is performed.
3. SQL uses Having clause to filter out groups in the result set.
4. Use order by to sort the result set. Group by does not perform sorting 5.
5. If a grouping column contains null values they are considered as equal and all null values are collected as single group.

<u>Examples.</u>

1. Using simple group by clause.

```
SELECT SalesOrderID, SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail AS sod
GROUP BY SalesOrderID
ORDER BY SalesOrderID
```

2. Using group by clause with multiple tables.

```
SELECT a.City, COUNT(bea.AddressID) EmployeeCount
FROM Person.BusinessEntityAddress AS bea
   INNER JOIN Person.Address AS a
      ON bea.AddressID = a.AddressID
GROUP BY a.City
ORDER BY a.City
```

1. Using group by clause with an expression

```
SELECT DATEPART(yyyy,OrderDate) AS N'Year'
   ,SUM(TotalDue) AS N'Total Order Amount'
FROM Sales.SalesOrderHeader
GROUP BY DATEPART(yyyy,OrderDate)
ORDER BY DATEPART(yyyy,OrderDate)
```

2. Using group by with having clause

```
SELECT DATEPART(yyyy,OrderDate) AS N'Year'
   ,SUM(TotalDue) AS N'Total Order Amount'
FROM Sales.SalesOrderHeader
GROUP BY DATEPART(yyyy,OrderDate)
HAVING DATEPART(yyyy,OrderDate) >= N'2003'
ORDER BY DATEPART(yyyy,OrderDate)
```

15. Built-In Functions (SQL Server)

What are Built-In Functions?

- In SQL a built-in function is a piece for programming that takes zero or more inputs and returns a value.
- Built-In functions are used in SQL SELECT expressions to calculate values and manipulate data.
- These functions can be used anywhere expressions are allowed. Common uses of functions include changing a name to all upper case and so on...

There are several things to note regarding functions...!!!

a. The inputs to a function are called parameters. Not all function has parameters, and some functions have more than one.

b. Parameters are enclosed in parenthesis.

c. We use functions in the SELECT clause as well as the WHERE filter condition. A function can be used anywhere in a SELECT statement that you can use an expression.

d. Functions are reserved words. I would avoid using them as column or table names. If you do, then expect to qualify your names with brackets [].

How do Functions Behave?

A. Some functions return the same value each time you call them. These are said to be deterministic functions. For a given input, these functions return the same value each time they are called.

For Example SQRT (), which is used to return the square root of a number, is deterministic. No matter how many times you run it gives square root value of that selected parameter.

B. Non-deterministic functions' return value may change from execution to execution.

For Example GETDATE (), which returns the current date and time and returns a different value every second.

Here Come's Our Function Categories...!

There are over a hundred built-in functions in SQL Server. These functions are categorized into some major categories.

Math/Number Functions – perform advanced calculations and round numbers.

String/Character Functions – change text values to all upper case, or remove the trailing spaces from values.

Date Functions – add days or months to a date. Calculate the day of the week from the date.

Aggregate Functions-These are kind of Numeric functions used for aggregating the attributes in a table.

Advanced Functions- These are used for special purposes while dealing with a table data.

Some Commonly Used Math/Number Functions

ABS (): Returns the absolute value of a number

Syntax: SELECT ABS (-243.5) AS AbsNum;

Output is "243.5"

CEILING (): Return the smallest integer value that is greater than or equal to a number.

Syntax: SELECT CEILING (25.75) AS Ceil Value;

Output is "26"

FLOOR (): Returns the largest integer value that is less than or equal to a number.

Syntax: SELECT FLOOR (25.75) AS Floor Value;

Output is "25"

PI (): Returns the value of PI

Syntax: SELECT PI ();

Output is "value of Pi"

POWER (): Returns the value of a number raised to the power of another number.

Syntax: SELECT POWER (4,2);

Output is "16"

RAND (): Return a random decimal number (no seed value - so it returns a completely random number >= 0 and <1.

Syntax: SELECT RAND ();

Output is "some random decimal number"

ROUND (): Rounds a number to a specified number of decimal places

Syntax: SELECT ROUND (235.415, 2) AS Round Value;

Output is "235.41"

SQRT (): Returns the square root of a number

Syntax: SELECT SQRT (64);

Output is "8"

SQUARE (): Returns the square of a number

Syntax: SELECT SQUARE (8);

Output is "64"

Some Commonly Used String/Character Functions

ASCII: Returns the ASCII value for the specific character. If more than one character is entered, it will only return the value for the first character

Syntax: ASCII (character)

Usage: SELECT ASCII (<column name>) AS <some name> FROM <Table name>;

CHAR: Returns the character based on the ASCII code.

Syntax: CHAR (code)

Usage: SELECT CHAR(<code>) AS <code_to_character>;

CHARINDEX: Returns the position of a substring in a string

<u>Syntax:</u> SELECT CHARINDEX('<substring>', '<string>') AS <some_name>;

Usage: CHARINDEX (substring, string, start)

CONCAT: Adds two or more strings together

<u>Syntax:</u> CONCAT (string1, string2, ..., string_n)

Usage: SELECT CONCAT('<String1>', '<string2>');

LEFT: Extracts a number of characters from a string (starting from left)

<u>Syntax:</u> LEFT (string, number_of_chars)

Usage: SELECT LEFT('<string>', <number of strings>) AS <somename>;

LEN: Returns the length of a string

<u>Syntax:</u> LEN (string)

Usage: SELECT LEN('<string_name>');

LOWER: Converts a string to lower-case

<u>Syntax:</u> LOWER (text)

Usage: SELECT LOWER('<some_Text>');

LTRIM: Removes leading spaces from a string

<u>Syntax:</u> LTRIM (string)

Usage: SELECT LTRIM '   <string>") AS <some_name>;

RTRIM: Removes trailing spaces from a string

<u>Syntax:</u> TRIM (string)

Usage: SELECT RTRIM '   <string>") AS <some_name>;

STR: Returns a number as string

<u>Syntax:</u> STR (number, length, decimals)

Usage: SELECT STR(<some_number>);

SUBSTRING: Extracts some characters from a string

<u>Syntax:</u> SUBSTRING (string, start, length)

Usage: SELECT SUBSTRING('<string>', <start from number>, <up to number>) AS <some_name>;

Some Commonly Used Date/Time Functions

CURRENT_TIMESTAMP: Returns the current date and time

Syntax: CURRENT_TIMESTAMP;

Usage: SELECT CURRENT_TIMESTAMP;

DATEADD: Adds a time/date interval to a date and then returns the date

Syntax: DATEADD (interval, number, date)

Usage: SELECT DATEADD (year, 1, '2017/08/25') AS DateAdd;

DATEDIFF: Returns the difference between two dates

Syntax: DATEDIFF (interval, date1, date2)

Usage: SELECT DATEDIFF (year, '2017/08/25', '2011/08/25') AS DateDiff;

DATENAME: Returns a specified part of a date (as string)

Syntax: DATENAME (interval, date)

Usage: SELECT DATENAME (year, '2017/08/25') AS DatePartString;

DATEPART: Returns a specified part of a date (as integer)

Syntax: DATEPART (interval, date)

Usage: SELECT DATEPART (year, '2017/08/25') AS DatePartInt;

DAY: Returns the day of the month for a specified date

Syntax: DAY (date)

Usage: SELECT DAY ('2017/08/25') AS DayOfMonth;

GETDATE: Returns the current database system date and time

Syntax: GETDATE ()

Usage: SELECT GETDATE ();

ISDATE: Checks an expression and returns 1 if it is a valid date, otherwise 0

Syntax: ISDATE (expression)

Usage: SELECT ISDATE ('2017-08-25');

MONTH: Returns the month part for a specified date (a number from 1 to 12)

Syntax: MONTH (date)

Usage: MONTH (date)

SYSDATETIME: Returns the date and time of the SQL Server

Syntax: SYSDATETIME ()

Usage: SELECT SYSDATETIME () AS SysDateTime;

YEAR: Returns the year part for a specified date

Syntax: YEAR (date)

Usage: SELECT YEAR ('2017/08/25') AS Year;

Some Commonly Used Aggregate Functions

These are kind of Numeric functions which are used in aggregations of parameters in a table.

AVG: Calculates the average of non-NULL values in a set.

Syntax: AVG (expression)

Usage: SELECT AVG(<column_name>) AS <some_name> FROM <Table_Name>;

COUNT: Returns the number of rows in a group, including rows with NULL values.

Syntax: COUNT (expression)

Usage: SELECT COUNT (<column_name>) AS <some_name> FROM <Table_Name>;

MAX: Returns the highest value (maximum) in a set of non-NULL values.

Syntax: MAX (expression)

Usage: SELECT MAX(<column_name>) AS <some_name> FROM <Table_name>;

MIN: Returns the lowest value (minimum) in a set of non-NULL values.

Syntax: MIN (expression)

Usage: SELECT MIN(<column_name>) AS <some_name> FROM <Table_name>;

SUM: Returns the summation of all non-NULL values a set.

Syntax: SUM (expression)

Usage: SELECT SUM(<column_name>) AS <some_name> FROM <Table_name>;

Some Commonly Used Advanced Functions

CAST (): Converts a value (of any type) into a specified datatype.

Syntax: CAST (expression AS datatype(length))

Usage: SELECT CAST (25.65 AS int);

Output is "25"

COALESCE (): Returns the first non-null value in a list.

Syntax: COALESCE (val1, val2, ..., val_n)

Usage: SELECT COALESCE (NULL, NULL, NULL, 'Sunil', NULL, 'Chandra');

Output is "Sunil"

CURRENT_USER: Returns the name of the current user in the SQL Server database.

Syntax: CURRENT_USER

Usage: SELECT CURRENT_USER;

IIF (): Returns a value if a condition is TRUE, or another value if a condition is FALSE.

Syntax: IIF (condition, value_if_true, value_if_false)

Usage: SELECT IIF (500<1000, 'YES', 'NO');

Output is "YES"

ISNUMERIC (): Tests whether an expression is numeric. This function returns 1 if the expression is numeric, otherwise it returns 0.

Syntax: ISNUMERIC (expression)

Usage: SELECT ISNUMERIC (4567);

Output is "1"

IMPLEMENTING FUNCTIONS IN SQL SERVER

The SQL server user defined functions help you to simplify our development by encapsulating complex business logic and make them available for reuse in every query.

USER DEFINED FUNCTIONS:

In SQL Server user-defined functions, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

Why we use user defined functions?

- They allow modular programming
- They allow faster execution
- They can reduce network traffic

In SQL server, user defined functions are of three types

- Scalar functions
- Inline table valued functions
- Multi-statement table valued functions

Scalar functions

Scalar functions takes one or more parameters and returns a single value. In other words, it returns a single data value of the type defined in the RETURNS clause. The return type can be any data type except **text**, **ntext**, **image**, **cursor**, and **timestamp**

Syntax:

```
CREATE FUNCTION function_name
RETURNS
AS ss
BEGIN
Statements
RETURN
END
```

Example:

```
Create function functionGetEmpFullName
(
@FirstName varchar(50),
@lastName varchar(50)
)
Returns varchar(101)
As
Begin
return (Select @ FirstName + ‘ ‘ +@LastName);
end
```

Inline table-valued functions

The user-defined inline table-valued function returns a table variable as a result of actions performed by the function.

Example:

```
Create function functionGetEmpFullName ()
Returns Table
As
Return (Select * from Employee)
```

Multi-statement table-valued functions

A user-defined multi-statement table-valued function returns a table variable as a result of actions performed by the function. In this, a table variable must be explicitly declared and defined whose value can be derived from multiple SQL statements

Example:

```
Create function functionGetEmpFullName ()
returns @Emp Table
(
EmpID int,
FirstName varchar(50),
Salary int
)
As
begin
```

16. T-SQL BASICS

T-SQL (Transact-SQL) is a set of programming extensions from Sybase and Microsoft that add several features to the Structured Query Language (SQL), including transaction control, exception and error handling, row processing and declared variables.

Exception and Error handling

An error condition during a program execution is called an exception and the mechanism for resolving such an exception is known as an exception handler. SQL Server provides TRY, CATCH blocks for exception handling

Types of SQL Server Exceptions

SQL Server contains the following two types of exceptions:

1. System Defined

In a System Defined Exception the exceptions (errors) are generated by the system.

2. User Defined

This type of exception is user generated, not system generated

Syntax of Exception Handling

```
BEGIN TRY
/* T-SQL Statements */
END TRY
BEGIN CATCH
- Print Error OR
- Rollback Transaction
END CATCH
```

The following are system functions and the keyword used within a catch block

- @@ERROR
- ERROR_NUMBER()
- ERROR_STATE()
- ERROR_LINE()
- ERROR_MESSAGE()
- ERROR_PROCEDURE()
- ERROR_SEVERITY()
- RAISERROR()

Example

```
BEGIN TRY
SELECT SALARY + First_Name From Employee Where Emp_IID=5
END TRY
BEGIN CATCH
SELECT ERROR_STATE() AS Error_Stat,ERROR_SEVERITY() AS ErrorSeverity, ERROR_LINE() as ErrorLine,
ERROR_NUMBER() AS ErrorNumber, ERROR_MESSAGE() AS ErrorMsg;
END CATCH;
```

Output

Error_Stat	ErrorSeverity,	ErrorLine	ErrorNumber	ErrorMsg
1	16	2	245	ion failed when converting the nvarchar……

SQL Variable declaration

In SQL Server (Transact-SQL), a variable allows a programmer to store data temporarily during the execution of code

Before using any variable in batch or procedure, you need to **declare the variable**

Local variable names have to start with an at (@) sign because this rule is a syntax necessity

Assigning a value to a VARIABLE

You can assign a value to a variable in the following **three** ways**:**

1. During variable declaration using DECLARE keyword.
2. Using SET
3. Using SELECT

1) During variable declaration using DECLARE keyword

Syntax is

Declare { @Local_Variable [AS] Datatype [= value] }

Example:

DECLARE @COURSE_ID AS INT = 5

PRINT @COURSE_ID

2) Using SET

Sometimes we want to keep declaration and initialization separate.

SET can be used to assign values to the variable, post declaring a variable.

Syntax:

Case 1:

DECLARE @Local_Variable <Data_Type>

SET @Local_Variable = <Value>

<u>Example:</u>

DECLARE @COURSE_ID AS INT

SET @COURSE_ID = 5

PRINT @COURSE_ID

<u>Case 2:</u>

Assign a value to multiple variables using SET.

DECLARE @Local_Variable _1 <Data_Type>, @Local_Variable_2 <Data_Type>,

SET @Local_Variable_1 = <Value_1>

SET @Local_Variable_2 = <Value_2>

3) USING SELECT

Just like SET, we can also use SELECT to assign values to the variables, post declaring a variable using DECLARE. Below are different ways to assign a value using SELECT

<u>Case 1</u>

<u>Syntax:</u>

DECLARE @LOCAL_VARIABLE <Data_Type>

SELECT @LOCAL_VARIABLE = <Value>

<u>Example:</u>

DECLARE @COURSE_ID INT

SELECT @COURSE_ID = 5

PRINT @COURSE_ID

<u>Case2:</u>

Assigning a value to multiple variable using SELECT

Syntax:

DECLARE @Local_Variable _1 <Data_Type>, @Local_Variable _2 <Data_Type>,SELECT @Local_Variable _1 = <Value_1>, @Local_Variable _2 = <Value_2>

If else

Example:

```
declare @var1 int =20,@var2 int=60
begin
  if(@var1>@var2)
              begin
   print 'var1 is greater'
    end
   else
    begin

    print 'var2 is greater'
              end
End

--Case
/*CASE <Case_Expression>
     WHEN Value_1 THEN Statement_1
     WHEN Value_2 THEN Statement_2
     .
     .
     WHEN Value_N THEN Statement_N
     [ELSE Statement_Else]
END AS [ALIAS_NAME]
*/
```

a) Case statement in select queries along with where, order by, group by.

b) It can also be used in Insert statement

TCL (Transaction Control Language)

4 modes Local transaction

1.Autocommit Transaction

2.Explicit transaction

3.Implicit Transaction

4.Batch-Scoped Transaction

Explicit transaction

  commit , Rollback, Save (Check point/Book Mark)

Commit Command: It is the transactional command used to save changes invoked by a transaction to the database. This command saves all the transactions to the database since the last commit or rollback command.

<u>Example:</u>

```
begin transaction t1
update sample set gender='m' where name='hari'
select * from sample
commit
rollback transaction
```

→For commit don't use save transaction

Rollback Command: The Rollback Command is the transactional command used to undo transactions that have not already been saved to the database.This command can only be used to undo transactions since the last commit or rollback command was issued.

SavePoint Command: A SavePoint Command is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

The Rollback command is used to undo a group of transactions.

<u>Example:</u>

```
select * from TestEntity
select * from sample
begin transaction
insert into sample (name,age,gender) values('divya',33,'f')
save transaction s1
update sample set gender='m' where name='hari'
save transaction s2
delete from sample
select * from sample
save transaction s3
rollback transaction s1
select * from TestEntity
insert into TestEntity values(105,'NeuralNetwork')

begin transaction
save transaction s1
insert into TestEntity values(107,'Purify')
update TestEntity set Testid=114 where Testid=4
save transaction s2
select * from TestEntity
rollback transaction s1
```

While

Example:

```
 declare @count int
 set @count=1
 while @count <=5
  Begin
  Print @count
  set  @count= @count+1
  End
```

While break and Continue

Example:

```
DECLARE @count1 INT = 0
WHILE @count1 < 10
BEGIN
   SET @count1 = @count1 + 1

   IF @count1 = 5
      CONTINUE

   IF @count1 = 7
      BREAK

   PRINT @count1
END
```

17. TRIGGERS

A trigger in SQL is a procedural code that is automatically executed in response to certain events on a specified table.

Triggers are the SQL codes that are automatically executed in response to certain events on a particular table. These are used to maintain the integrity of the data. A trigger in SQL works similar to a real-world trigger.

For example, John is the marketing officer in a company. When a new customer data is entered into the company's database he has to send the welcome message to each new customer. If it is one or two customers John can do it manually, but what if the count is more than a thousand? Well in such scenario triggers come in handy.

Thus, now John can easily create a trigger which will automatically send a welcome email to the new customers once their data is entered into the database.

Always remember that there cannot be two triggers with similar action time and event for one table. For example, we cannot have two BEFORE UPDATE triggers for a table. But we can have a *BEFORE UPDATE* and a *BEFORE INSERT* trigger, or a *BEFORE UPDATE* and an *AFTER UPDATE* trigger.

Syntax and Example

//// Create Trigger Trigger_Name

(Before | After) [Insert | Update | Delete]

on [Table_Name]

[for each row | for each column]

[trigger_body]

Break down of syntax

- **Create Trigger**

These two keywords are used to specify that a trigger block is going to be declared.

Example-

```
create trigger t_InsertSample
on Sample
for Insert
as
begin
print 'Insert Operation Not Allowed!!'
rollback transaction
end

select * from sample
insert into sample(name,age,gender) values('Sivam',22,'Male')
```

- **Trigger_Name**

It specifies the name of the trigger. Trigger name has to be unique and shouldn't repeat.

- **(Before|After)**

This specifies when the trigger will be executed. It tells us the time at which the trigger is initiated, i.e, either before the ongoing event or after.

- Before Triggers are used to update or validate record values before they're saved to the database.
- After Triggers are used to access field values that are set by the system and to effect changes in other records. The records that activate the after trigger are read-only. We cannot use After trigger if we want to update a record because it will lead to read-only error.

- **[Insert|Update|Delete]**

These are the DML operations and we can use either of them in a given trigger.

Example on Insert Trigger

```
create trigger t_E_New_desc
on tblemployee
for Insert
as
begin
declare @t_id int
set @t_id=(select Eid from inserted)
insert into tbl_Emp_New_Desc(Eid,Description) values(@t_id,'NewEmployeeJoined')
end

insert into tblemployee(Eid,Ename,Salary,Gender) values(1011,'Gayathri',40000,'F')

select * from tblEmployee
select * from tbl_Emp_New_Desc


alter table tbl_Emp_New_Desc add Joineddate date ,Resigneddate date
 -----
alter trigger t_E_New_desc
on tblemployee
for Insert
as
begin
declare @t_id int
set @t_id=(select Eid from inserted)
insert into tbl_Emp_New_Desc(Eid,Description,Joineddate)
values(@t_id,'NewEmployeeJoined', getdate())
end

insert into tblemployee(Eid,Ename,Salary,Gender) values(1012,'Radha',40000,'F)
```

Example on Delete Trigger

```
create trigger t_delete_emp
on tblEmployee
for delete
as
begin
declare @t_id int
```

```
set @t_id=(select Eid from deleted)
delete from tbl_Emp_New_Desc where eid=@t_id
--update tbl_Emp_New_Desc set resigneddate=GETDATE() where eid=@t_id
end


delete from tblemployee where eid=1012
```

Example on Update Trigger

```
CREATE TABLE tblEmployeeLeave
(
EmployeeID int CONSTRAINT fkEmployeeID
FOREIGN KEY REFERENCES tblEmployee(Eid),
LeaveStartDate datetime NOT NULL ,
LeaveEndDate datetime NOT NULL,
LeaveReason varchar(100),
LeaveType char(2) CONSTRAINT chkLeave
CHECK(LeaveType IN('CL','SL','PL'))
CONSTRAINT chkDefLeave DEFAULT 'PL',
Status nvarchar(30) ,LeaveApprovedDate datetime,
)

select * from tblEmployeeLeave

alter trigger t_Leave
on tblEmployeeLeave
for update
as
begin
declare @t_eid int
declare @t_status varchar(15)
set @t_eid=(select EmployeeID from inserted)
set @t_status=(select Status from inserted)
update tblEmployeeLeave set  LeaveApprovedDate=GETDATE() where EmployeeID=@t_eid and
Status='approved'
end

update tblEmployeeLeave set status='Approved' where Employeeid=1001
```

- **ONTable_Name**

We need to mention the table name on which the trigger is being applied. Don't forget to use **on** keyword and also make sure the selected table is present in the database.

- **[for each row | for each column]**

1.Row-level trigger gets executed before or after any column value of a row changes

2.Column Level Trigger gets executed before or after the specified column changes

- **[trigger_body]**

It consists of queries that need to be executed when the trigger is called.

we can also create a nested trigger that can do multi-process. Also handling it and terminating it at the right time is very important. If we don't end the trigger properly it may lead to an infinite loop.

Operations in Triggers

- **DROP A Trigger**

DROP TRIGGER trigger name;

- **Display A Trigger**

SHOW TRIGGERS(code will display all the triggers that are present)

SHOW TRIGGERS IN database_name;(code will display all the triggers that are present in a particular database.)

Before and After of Trigger

BEFORE triggers run the trigger action before the triggering statement is run.

AFTER triggers run the trigger action after the triggering statement is run.

Advantages and Disadvantages of Triggers

Advantages

- Forcing **security** approvals on the table that are present in the database
- Triggers provide another way to check the **integrity of data**
- **Counteracting invalid** exchanges
- Triggers **handle errors** from the database layer
- Normally triggers can be useful for **inspecting the data** changes in tables
- Triggers give an alternative way to run s**cheduled tasks**. Using triggers, we don't have to wait for the scheduled events to run because the triggers are invoked automatically before or after a change is made to the data in a table

Disadvantages

- Triggers can only provide extended **validations**, i.e., not all kind validations. For simple validations, you can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints
- Triggers may increase the **overhead** of the database
- Triggers can be difficult to **troubleshoot** because they execute automatically in the database, which may not invisible to the client applications.