SQL

➢ Chapter 1: Constraints:

- **Types of Constraints**
  1) Primary Key constraint (PK)
  2) Foreign Key constraint (FK)
  3) Default constraint (DF)
  4) Check constraint (CK)
  5) Unique Constraint (unique)
  6) Not Null Constraint

1) **Primary Key ->** A primary Key is basically used to identify uniquely each record within that table.
2) **Foreign Key ->** A foreign key is a way to enforce referential integrity with SQL DB. A foreign key means that value in one table must appear in another table.

```
Alter table tblperson
ADD constraint tblperson_GenderID_FK
Foreign key (GenderID) References tblGender(ID)
```

3) **Default Constraints ->** The default constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified, including NULL.

```
Alter table tblPerson
ADD constraint DF_tblPerson_GenderID
DEFAULT 3 for GenderID
```

4) **Check Constraint ->** Check constraint is used to limit the range of the values that can be entered for a column.

```
Alter table tblPerson
ADD Constraint CK_tblPerson_Age
CHECK (Age > 0 AND Age < 150)
```

5) **Unique Constraints ->** We use UNIQUE constraint to enforce uniqueness of a column i.e the column shouldn't allow any duplicate values.

```
Alter table tblPerson
ADD constraint UQ_tblPerson_Email
UNIQUE (Email)
```

6) **NOT NULL Constraint ->** This constraint is useful to stop storing the Null entries in the specified column.

7) **Dropping Constraints**

```
Alter table tblperson
Drop constraint ck_tblperson_Age
```

8) **Adding and Dropping columns**

To Add:

```
Alter table tblperson
ADD “Column_Name” Datatype
```

TO Drop :

```
Alter table tblperson
Drop column “Column_Name”
```

➢ Chapter 2 : Clauses

1) **From :-**
2) **Where: -** is used to apply conditions.
3) **Having: -** this clause works like a where condition when using the aggregate functions.
4) **Group by: -** is used to group a selected set of rows into a set of summary by the values of one or more columns or expressions. It always used in conjunction with one or more aggregate functions.
5) **Order by: -** is used when we need sort data ascending or descending order.

What is the difference between having and Group by Clause?

➔ Having clause works like a where condition when using the aggregate functions and Group by clause is used to Group rows not columns.

➢ Chapter 3 : JOINS

Joins in SQL are used to retrieve data from 2 or more related tables. In general, tables are related to each other using foreign key constraint.

Types of Joins:

1) **INNER JOIN:** Inner join returns only the matching rows between both tables, non-matching rows are eliminated.

```
Select Name, Gender, Salary, Department_Name from tblEmployee
INNER JOIN tblDepartment ON tblEmployee.DepartmentID = tblDepartment.DeptID
```

2) **LEFT JOIN:** Left join returns all the matching rows + non-matching rows from the left table.

```
Select Name, Gender, Salary, Department_Name
From tblEmployee
LEFT JOIN tblDepartment
ON tblEmployee.DepartmentID = tblDepartment.DeptID
```

3) **Right JOIN:** Right join returns all the matching rows + non-matching rows from the right table.

```
Select Name, Gender, Salary, Department_Name
From tblEmployee
RIGHT JOIN tblDepartment
ON tblEmployee.DepartmentID = tblDepartment.DeptID
```

4) **FULL JOIN**: Full Join returns all rows from both the left and right tables, including the non-matching rows.

```
Select Name, Gender, Salary, Department_Name
From tblEmployee
FULL JOIN tblDepartment
ON tblEmployee.DepartmentID = tblDepartment.DeptID
```

5) **CROS JOIN:** Cross join produces the Cartesian product of the 2 tables involved in the join 10 rows x 4 rows = 40 rows.

```
Select Name, Gender, Salary, Department_Name
From tblEmployee
CROSS JOIN tblDepartment
```

NOTE: Cross join shouldn't have "ON" clause.

➢ Chapter 4 : ADVANCED OR INTELLIGENT JOINS

1) **Left Join:** only retrieve non-matching rows from left table.

```
Select Name,Gender, Salary, Department_Name
From tblEmployee
Left Join tblDepartment
On tblEmployee.DepartmentID = tblDepartment.DeptID
Where tblEmployee.DepartmentID IS NULL
```

2) **Right Join:** only retrieve non-matching rows from right table.

```
Select Name,Gender, Salary, Department_Name
From tblEmployee
Right Join tblDepartment
On tblEmployee.DepartmentID = tblDepartment.DeptID
Where tblEmployee.DepartmentID IS NULL
```

3) **Full Join:** only retrieve Non-Matching rows from right table and non-matching rows from left table.

```
Select Name,Gender, Salary, Department_Name
From tblEmployee
Full Join tblDepartment
On tblEmployee.DepartmentID = tblDepartment.DeptID
Where tblEmployee.DepartmentID IS NULL
```

➢ Chapter 5 : SELF JOIN

Joining a table with itself is called as self-join. It can be classified under any type of Join.

1) **Inner self join:** Inner join returns only the matching rows between both tables. Non-matching rows are eliminated.

```
Select E.Name As Employee, M.Name As Manager
From tblEmployee E
INNER JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
```

2) **Left self-Join:** It retrieves matching rows + non-matching rows from left table.

```
Select E.Name As Employee, M.Name As Manager
From tblEmployee E
LEFT JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
```

3) **Right self-join:** Right join returns all the matching rows + non-matching rows from the right table.

```
Select E.Name As Employee, M.Name As Manager
From tblEmployee E
RIGHT JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
```

4) **Full self-join:** only retrieve Non-Matching rows from right table and non-matching rows from left table.

```
Select E.Name As Employee, M.Name As Manager
From tblEmployee E
```

```
FULL JOIN tblEmployee M
ON E.ManagerID = M.EmployeeID
```

5) **CROSS self-join :** generate Cartesian Product

```
Select E.Name As Employee, M.Name As Manager
From tblEmployee E
CROSS JOIN tblEmployee M
```

➢ Chapter 6 : OPERATORS

01) Union
02) Union All
03) Intersect
04) Minus
05) Except
06) Exists
07) Arithmetic
08) Merge
09) Like
10) IN
11) Between
12) AND
13) OR
14) DISTINCT
15) TOP

➢ Chapter 7 : SQL LANGUAGES

1) **DDL (Data Definition Language)**

   DR.CAT

   - Drop
   - Create
   - Alter
   - Truncate

2) **DML (Data Manipulation Language)**

   UID

   - Update
   - Insert
   - Delete

3) **DQL (Data Query Language**
   - Select
4) **DCL (Data Control Language)**
   - Grant
   - REVOKE
5) **TCL (Transaction Control Language)**
   - Rollback
   - Commit
   - Save Point

➢ Chapter 8 : Sub-Queries

Sub-Queries is simply a “Select statement” that returns a single value and can be nested inside a select, update, insert or Delete statement. It is also possible to nest a sub queries inside another sub query. According to MSDN, sub queries can be nested up to 32 levels.

```
Select ID, Name,[Description] from tblProducts
Where ID NOT IN (Select Distinct Product_ID from tblProduct_Sales)
```

NOTE: the Query that present inside parenthesis that is called sub-query.

➢ Chapter 8 : Common Table Expression (CTE)

A CTE is a temporary result set that can be referenced within a SELECT, INSERT, UPDATE OR DELETE Statement that immediately follows the CTE.

```
With EmployeeCount (DepartmentID,TotalEmployees)
As
(
        Select DepartmentID,Count(*) as TotalEmployees
        From tblEmployee
        Group by DepartmentID
)
Select DepartmentName,TotalEmployees
From tblDepartment
JOIN EmployeeCount
ON tblDepartment.DeptID = EmployeeCount.DepartmentID
Order by TotalEmployees

With Employee_Name_Gender
AS
(
        Select ID,Name,Gender from tblEmployee
)
Update Employee_Name_Gender
Set Gender = 'male' where ID =3

With Employee_Name_Gender
AS
(
        Select ID, Name, Gender from tblEmployee
)
Delete from tblEmployee where Name = 'vikas'
```

➢ Chapter 9 : VIEWS

The views are nothing more than a saved SQL Query. A view can also be considered a virtual table.

```
Create view vwEmployeeByDapartment
As
SELECT ID, NAME, SALARY, GENDER, Department_NAME
From tblEmployee
Join tblDepartment
On tblEmployee.DepartmentID = tblDepartment.DeptID

Select * from vwEmployeeByDapartment
```

ROW LEVEL Security

```
Create view vwITEmployees As
Select ID, Name,Salary, Gender, Department_Name
From tblEmployee JOIN tblDepartment
ON tblEmployee.DepartmentID=tblDepartment.DeptID
Where tblDepartment.DepartmentID='IT'
```

Column Level Security

```
Create view vwnonConfidentialData As
Select ID, Name, Gender, DepartmentName
From tblEmployee
JOIN tblDepartment
On tblEmployee.DepartmentID = tblDepartment.DeptID
```

➢ Chapter 10 : STORE PROCEDURE

Store Procedure is a set of SQL statements with an assigned name, which is stored in RDBMS, so it can be reused and shared with multiple programs.

```
Create procedure spGetEmployees
AS
BEGIN
        Select Name, Gender from tblEmployee
END
```

Store Procedure with Input Parameters

```
Create procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(20),
@DepartmentID INT
AS
BEGIN
        Select Name, Gender, DepartmentID from tblEmployee
        From tblEmployee
        Where Gender = @Gender
        And DepartmentID = @DepartmentID
END
```

For Executing
spGetEmployees
Exec spGetEmployees
SpGetEmployeesByGenderAndDepartment[parameter],[parameter]
Sp_helptext Procedure Name

- **DROPPING OR DELETING**
  Drop Proc (ProcName)
  Alter proc (ProcName)

- **Store Procedure with Output Parameter**

```
Create Proc spGetEmployeeCountByGender
@Gender Nvarchar(20),@EmployeeCount INT Output
AS
BEGIN
        Select @EmployeeCount = count(ID) from tblEmployee
        Where Gender = @Gender
END
```

Executing Output Store Procedure

```
Declare @TotalCount int
Execute spGetEmployeeCountByGender 'Male', @Totalcount
Output
Print @TotalCount
```

➢ Chapter 11 : TRIGGERS

Triggers are database Operations which are automatically performed when an action such as INSERT, UPDATE, and DELETE is performed on a table or view directly i.e. each table has its own triggers.

- ➢ **Triggers are Classified into three types as below**
  1) **DML Trigger**
  2) **DDL Trigger**
  3) **Logon Trigger**

1) **DML Trigger : DML Triggers are fired automatically in response to DML events (insert, update, Delete)**

   A. **Inserted → insert table will contain the new data after the update action.(inserted table contains new data)**

```
Create Trigger trg_tblEmployee_FOR_insert
ON tblEmployee
FOR INSERT
AS
BEGIN
    Declare @ID INT
    SELECT @ID = id from inserted
    Insert into tblAudit values ('New Employee with ID = ' + cast(@ID as nvarchar(5)) ' is added at '
    + cast(GETDATE() as nvarchar(20)))
END
```

   B. **Updated → after updated triggers use both after inserted and after deleted tables.**

```
Create trigger trg_tblEmployee_for_Update
ON tblEmployee
As
BEGIN
    Select * from deleted
    Select * from inserted
END
```

   C. **DELETED → Deleted table will contain the data that the table has before the update action.(Deleted table contains the old Data)**

```
Create Trigger trg_tblEmployee_For_Deleted
ON tblEmployee
FOR Deleted
AS
```

```
BEGIN
    Declare @ID INT
    Select @ID = ID from deleted
    INSERT into tblAudit values
    ('An Existing Employees with ID = ' + CAST (@ID as nvarchar(5)) + ' is deleted at ' +
CAST(GETDATE() as nvarchar(20)))
END
```

➢ Chapter 12 : INDEXES

Indexes are used by queries to find data from tables quickly. 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. In fact, the existence of the right indexes, can drastically improve the performance of the query. If there is no index to help the query, then the query engine, checks row in the table from the beginning to the end. This is called as table scan and the table scan is bad for performance. Two types of index clustered and non-clustered.

```
Create Index IX_tblEmployee_Salary
ON tblEmployee (Salary ASC)

Sp_helpIndex tblEmployee

Drop index tblEmployee. IX_tblEmployee_Salary
```

Types of Indexes

1) Clustered
2) Non-clustered
3) Unique and Non unique
4) Filtered
5) XML
6) Full text
7) Spatial
8) Column Store
9) Index with Included columns
10) Index on computed column

- Clustered Index → A clustered index determines the physical order of data in a table. For this reason, a table can have only one clustered index.

➢ Chapter 13 : Types of Tables

➢ Table Variables

Table variables are used as an alternative of temporary tables.
Table variable begin with Declare @T
Insert into @T

➢ Temporary Tables

Temporary tables are very similar to the permanent tables, permanent tables get created in the database you specify, and remain in the database permanently, until you delete (drop) them. On the other hand, temporary tables get created in the Temp DB and are automatically deleted, when they are no longer used.
Different Types of Temp tables

1) Local Temporary Tables
2) Global Temporary Tables
3) Local temp table begins with Create table #Table name
4) Global temp table begins with Create table ##tablename

➢ Derived Table

A table which is derived from select statement is known as Derived Table.

```
Select * from
(
        Select Ename, Salary, Commission, Salary + commission as TotalSalary from
tblEmp
) as tblSalary
Where TotalSalary > 3000;
```

➢ Magic Table

Magic Tables are nothing but inserted and deleted which temporary objects are created by the server internally to hold recently inserted values in case of insert and to hold recently deleted values in case of delete, to hold before updating values or after updating values in case of update you see these tables with the help of triggers

```
Create trigger trg_for_Insert_Magic
On update
For INSERT
AS
BEGIN
        Select * from inserted
END
```

➢ Bulk copy

➢ Back up table

insert into

➢ Clone table

insert into tblname select statement

➢ Pivot Table

Pivot is a SQL Server Operator that can be used to turn unique values from one column, into multiple columns in the output, thereby effectively rotating a table.

```
Select SalesAgent, INDIA, US, UK
From tblProductSales
PIVOT
(
        Sum(SalesAmount)
        For SalesCountry
        IN ([INDIA],[US],[UK])
) AS PivotTable

Select SalesAgent, INDIA,US,UK
From
(
        Select SalesAgent,SalesCountry,SalesAmount
        From tblProductSales
) as SourceTable
PIVOT
(
        SUM(SalesAmount) for SalesCountry
        IN (India,US,UK)
) AS PivotTable
```

➢ Chapter 14 : TRANSACTIONS

Transactions is like an event which occurs whenever something changes data in your database. A transaction is a group of commands that change the data stored in a database.

```
BEGIN Transaction
Update tblProduct set QtyAvailable = 200
Where productID = 1
Commit Transaction --- To complete the transaction
Rollback Transaction – Undo Transaction
```

➢ Chapter 15 : Cursor

Cursor is nothing but pointer to a row

- **Types of Cursors**
  1) **Local**
  2) **Global**
- **Record set types of Cursors**
  1) **Static**
  2) **Dynamic**
  3) **Keyset**
  4) **Fast_forward**
- **Lock Type Cursor**
  1) **Read_only**
  2) **Scroll_locks**
  3) **Optimistic**

```
DECLARE FilmCursor CURSOR
FOR
SELECT
        FilmID,
        FilmName,
        FilmReleaseDate
FROM
        tblFilm
OPEN FilmCursor
        FETCH NEXT FROM FilmCursor – To move the cursor to first record
                While @@FETCH_STATUS=0 -- To fetch records continuously
        FETCH NEXT FROM filmCursor
CLOSE FilmCursor
DEALLOCATE FilmCursor;
```

➢ Chapter 16 : Functions

What is Function?

A function is a sequence of statements that accepts inputs, process them to perform a specific task and provide the output. Function must have a Name but the function name can be never start with a special character such as @,$,# and so on.

Functions: Broadly divided into 2 Categories

1) **System Functions**
2) **User Defined Functions**

➢ **System Functions**

1) **Aggregate Functions**
2) **Configuration functions**
3) **Cursor functions**
4) **Date & Time functions**
5) **Hierarchy ID functions**
6) **Rowset functions**
7) **String functions**
8) **System statistical functions**
9) **Text and Image functions**

➢ **Commonly used String functions :**

ASCII (Character Expression): Returns the ASCII code of the given character expression.

```
Select ASCII ('A')
Select CHAR(65)
Select LRTIM('      Hello') – Remove leading space
Select RTRIM('Hello      )  -- Remove trailing spaces.
Select LOWER('HELLO')
Select UPPER('hello')
Select REVERSE('  Hello ')
Select LEN('Hello')
Select LEFT('ABCDF',3)
Select CHARINDEX('@', 'sara@aaa.com',1)
Select substring('sara@aaa.com',6,7)
Select REPLICATE('Vikas',3)
Select SPACE(5)
```

Select PATINDEX('%@aaa.com','sara@aaa.com')
Select REPLACE('@gamil.com','@amdocs.com','vikas.bansode@amdocs.com')

- **Date and Time Functions**

Select ISDATE('vikas')
Select ISDATE(GetDate())
Select ISDATE('2012-08-07')
Select ISDATE('2012-09-01 11:34:21.1918447')
Select Day(GetDate())
Select MONTH(GetDate())
Select YEAR(GetDate())
Datename()
Datepart()
DateAdd()
Datediff()
Cast()
Convert()

- **Mathematical Functions**

Abs()
Ceiling()
Floor()
Power()
Rand()
Square()
Sqrt()
Round()

- **USER DEFINED Functions**

Types of UDF:

1) **Scalar Functions ->** A function may or may not have parameters, but always return a single (Scaler) value. The returned value can be of any data type, except text, ntext, image, cursor and timestamp.

Function Syntax:

```
Create FUNCTION Function_name(@parameter1 Datatype,@parameter2 Datatype,
@parameterN Datatype)
RETURNS Return_Datatype
AS
BEGIN
    Function Body
    Return Return_Datatype
END
```

Example :

```
Create Function CalculateAge (@DOB Date)
RETURN INT
AS
BEGIN
        DECLARE @Age INT
        SET @Age = DATEDIFF(year,@DOB,GetDate()) –
        Case
        WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR
        (MONTH(@DOB)= MONTH(GETDATE()) AND DAY(@DOB) > DAY(GETDATE()))
        THEN 1
        ELSE 0
        END
RETURN @Age
END
```

To Invoke the function use below statement

```
Select dbo.CalculateAge('11/03/1984') as Age
```

2) Inline table-valued functions

Inline table valued function returns a table

1. We specify table as the return type, instead of any scalar data type.
2. The function body is not enclosed between BEGIN and END block.
3. The structure of the table that gets returned, is determined by the SELECT statement within the function.

- Creating Inline valued table functions **syntax :**

```
Create function fn_EmployeesByGender
(@Gender nvarchar(10))
RETURNS Table
AS
RETURN
(Select ID, Name, Gender, DepartmentID from tblEmployee
Where Gender = @Gender)
```

- To call the function

```
Select * from EmployeesByGender('Male')
```

Where can we use inline table valued functions?

1) Inline table valued functions can be used to achieve the functionality of parameterized views.

2) The table returned by the table valued functions, can also be used in joins with other tables.

3) Multi-statement table-valued functions

Multi statement table valued functions are very similar to inline values functions, with a few differences.

```
Create function fn_MSTVF_GetEmployees()
RETURN @Table Table (ID INT,Name nvarchar (20),DOB Date)
AS
BEGIN
    Insert into @Table
Select ID,Name Cast(DateOfBirth AS Date) from tblEmployees
END
```

- To call the function

```
Select * from fn_MSTVF_GetEmployees()
```