1. **How to create a db?**

   The `CREATE database` command creates a new Database in the Instance server.

```
create database onlineclass
```

2. **How to create a table?**
   The `CREATE TABLE` command creates a new table in the database.

```
create table empdetails
(empid int,
empname varchar (20),
address varchar (20),
joindate Date,
salary money)
```

3. **How to insert data into table?**

```
Create TAble EMPLoyee
(
EMPID int ,
EMPNAME varchar(100),
Salary  Money,
deptid int
)
--Insert the data for all columns
insert into EMPLoyee values
(1,'vsds',40000,25)
--Insert the data for selected columns

insert into EMPLoyee(EMPID, EMPNAME) values
(1,'vsds')
```

4. **How to alter command use?**

***Alter syntax:(one column add)**

```
ALTER TABLE table_name
ADD column_name datatype
```

***Alter syntax:(one column** delete**)**

```
ALTER TABLE table_name
DROP COLUMN column_name
```

What is the Difference between DELETE and TRUNCATE and DROP Statements?

***Difference between DELETE and TRUNCATE Statements:**

DELETE Statement:

This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the

table if no condition is specified. But it does not free the space containing the table.

syntax:

```
DELETE FROM table_name WHERE some_column=some_value;
```

TRUNCATE statement: This command is used to delete all the rows from the table and free the space containing the table.

syntax:

```
TRUNCATE TABLE table_name;
```

SQL DROP Statement:

The SQL DROP command is used to remove an object from the database. If you drop a table, all the rows in the table is deleted and the table structure

is removed from the database. Once a table is dropped, we cannot get it back, so be careful while using DROP command.

When a table is dropped all the references to the table will not be valid.

```
* DELETE FROM table_name [WHERE condition]
* DROP TABLE table_nam
```

SQl CONSTRAINTS:

Constraints are some rules that enforce on the data to be enter into the database table.

Basically, constraints are used to restrict the type of data

that can insert into a database table.

Constraints can be defined in two ways:

1. **Column Level**

The constraints can be specified immediately after the column definition with the CREATE TABLE statement. This is called column-level constraints.

2. **Table Level**

The constraints can be specified after all the columns are defined with the ALTER TABLE statement. This is called table-level constraints.

SQL CREATE TABLE + CONSTRAINT Syntax:

```
CREATE TABLE table_name
            (
                  column_name1 data_type(size) constraint_name,
                  column_name2 data_type(size) constraint_name,
                  column_name3 data_type(size) constraint_name,
                  ....
```

```
);
```

In SQL, we have the following constraints:

1 NOT NULL - This constraint ensures that all rows in the database table must contain value for the column which is specified as not null means a null value is not allowed in that column.

The NOT NULL constraint enforces a column to NOT accept NULL values.

The NOT NULL constraint enforces a field to always contain a value. This means that you cannot insert a new record, or update a record without adding a value to this field.

Example:

Column Level:

```
CREATE TABLE PersonsNotNull
	(
		P_Id int NOT NULL,
		LastName varchar(255) NOT NULL,
		FirstName varchar(255),
		Address varchar(255),
		City varchar(255)
	)
```

Table Level:

```
ALTER TABLE Persons
ADD CONSTRAINT UC_lastname UNIQUE (LastName);
```

2 UNIQUE

Unique key is a set of one or more fields/columns of a table that uniquely identify each record/row

in database table. It is like Primary key but it can accept only one null value and it can not have duplicate values

[Note that you can have many UNIQUE constraints per table.]

Example:

Column Level:

```
CREATE TABLE Persons
	(
	P_Id int NOT NULL UNIQUE,
	LastName varchar(255) NOT NULL,
	FirstName varchar(255),
	Address varchar(255),
	City varchar(255)
	)
```

Table Level:

```
ALTER TABLE Persons
ADD CONSTRAINT UC_lastname UNIQUE (LastName);
```

3 PRIMARY KEY –

A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have an unique identity which helps to find a particular record in a table more easily and quickly.

* Primary keys must contain UNIQUE values.

* A primary key column cannot contain NULL values.

* Most tables should have a primary key, and each table can have only ONE primary key.

Example:

Column Level:

```
CREATE TABLE Persons
(
P_Id int NOT NULL PRIMARY KEY,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar (255),
City varchar (255)
)
```

Table Level:

```
CREATE TABLE Persons2
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar (255),
City varchar (255)
)

ALTER TABLE Persons2
ADD CONSTRAINT Pk_P_Id PRIMARY KEY  (P_Id);
```

4 FOREIGN KEY –

Ensure the referential integrity of the data in one table to match values in another table

A foreign key is a column (or columns) that references a column (most often the primary key) of another table. The purpose of the foreign key is to ensure referential

integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

Column Level:

```
CREATE TABLE Department
       (
        DeptID int PRIMARY KEY, --define primary key
        Name varchar (50) NOT NULL,
        Address varchar(100) NULL
       )
Create Relation Table

  CREATE TABLE Employee
              (
               EmpID int PRIMARY KEY, --define primary key
               Name varchar (50) NOT NULL,
               Salary int NULL,
               --define foreign key
              DeptID int FOREIGN KEY REFERENCES Department(DeptID)
              )
```

select *from Department

select * from Employee

insert into Department (DeptID,Name,Address) values (2,'emp','hyd')

insert into Employee values(2,'murali',2000,1)

```
CREATE TABLE CITY
  (      ID int not null PRIMARY KEY,
         CITYNAME VARCHAR(100)
  )

   CREATE TABLE subjects
         (      subject_id int not null PRIMARY KEY,
                subject_NAME VARCHAR(100)
         )


  CREATE TABLE student
         (
         id INT PRIMARY KEY,
         first_name VARCHAR(100) NOT NULL,
         last_name VARCHAR(100) NOT NULL,
         city_id INT FOREIGN KEY REFERENCES city(id)
         );
```

```
CREATE TABLE student1
       (
       id INT PRIMARY KEY,
       first_name VARCHAR(100) NOT NULL,
       last_name VARCHAR(100) NOT NULL,
       city_id INT,
       FOREIGN KEY (city_id) REFERENCES city(id)
       );


       CREATE TABLE student2 (
       id INT PRIMARY KEY,
       first_name VARCHAR(100) NOT NULL,
       last_name VARCHAR(100) NOT NULL,
       city_id INT,
           CONSTRAINT fk_student_city_id
         FOREIGN KEY (city_id) REFERENCES city(id)
       );
```

TABLE Level:

```
ALTER TABLE student
ADD FOREIGN KEY (city_id) REFERENCES city(id);

ALTER TABLE student
 ADD CONSTRAINT fk_student_city_id
    FOREIGN KEY (city_id) REFERENCES city(id)
```

5 CHECK - Ensures that the value in a column meets a specific condition

This constraint defines a business rule on a column in the database table that each row of the table must follow this rule.

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.

Example:

Column Level:

```
CREATE TABLE Persons
       (
              P_Id int NOT NULL CHECK (P_Id>0),
              LastName varchar(255) NOT NULL,
              FirstName varchar(255),
              Address varchar(255),
              City varchar(255)
       )
```

Table Level:

```
Create the table
CREATE TABLE TABLEcheck1
(ID int,
age int)

Add Check constraint
alter table TABLEcheck1
```

```
add constraint df_checkage  check (age>30)
```

6 DEFAULT - Specifies a default value when specified none for this column

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.

Example:

Column Level:

```
CREATE TABLE Persons
(
        P_Id int NOT NULL,
        LastName varchar(255) NOT NULL,
        FirstName varchar(255),
        Address varchar(255),
        City varchar(255) DEFAULT 'Sandnes'
        )
--Example2:

--The DEFAULT constraint can also be used to insert system values, by using functions
like GETDATE():
   CREATE TABLE Orders
   (
        O_Id int NOT NULL,
        OrderNo int NOT NULL,
        P_Id int,
        OrderDate date DEFAULT GETDATE()
   )
```

Table Level:

```
Create the table
 CREATE TABLE TABLE2
       (      ID int,
              Sysydate datetime
       )

Add default constraint
 alter table TABLE2
 add constraint df_Sysydate default Getdate() for [Sysydate]
```

example:

Column Level:

```
CREATE TABLE Persons (
        PersonID int NOT NULL PRIMARY KEY,
        PersonNAME int NOT NULL,
          );

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

SELECT Department.Name, Employee.Name, Employee.Salary

FROM Department

INNER JOIN Employee

ON Department.DeptID=Employee.EmpID;

SQL Clause:

1. **DISTINCT:**

In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.

The DISTINCT keyword can be used to return only distinct (different) values.

Syntax: SELECT DISTINCT column_name,column_name FROM table_name; +

2. **WHERE Clause:**

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

Syntax:

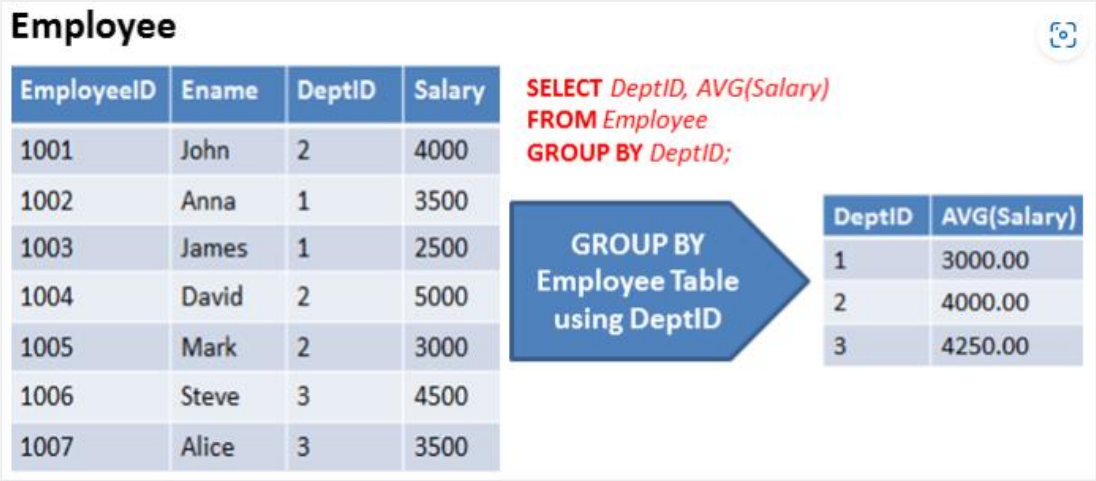
```
SELECT column_name,column_name
 FROM table_name WHERE column_name operator value;
(select * from tablename WHERE COLUMNNAME='SOMEVALUE')
```

3. **Group BY**

The GROUP BY Clause is utilized in SQL with the SELECT statement to organize similar data into groups. It combines the multiple records in single or more columns using some functions. Generally, these functions are aggregate functions such as min(),max(),avg(), count(), and sum() to combine into single or multiple columns. It uses the **split-apply-combine** strategy for data analysis.'

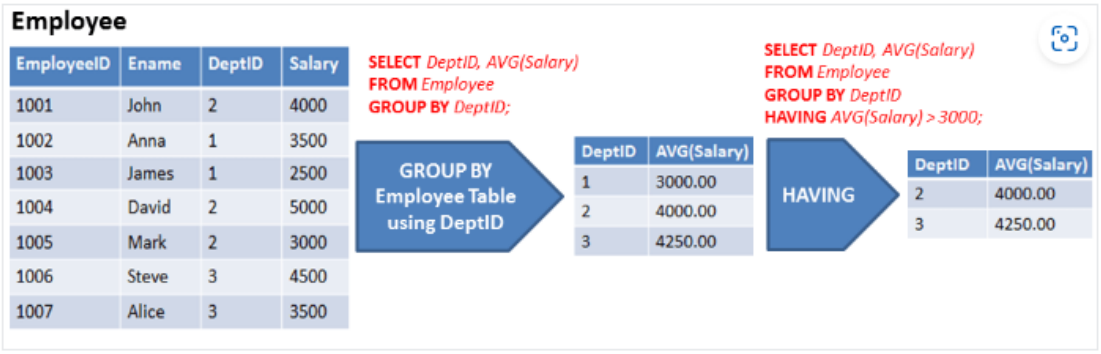
Synatx:

```
SELECT SUM (SAL) AS SUMOFSALARIES, DEPTNO
FROM EMPLOYEES
 GROUP BY DEPTNO;
```



4. Having Clause

HAVING Clause utilized in SQL as a conditional Clause with GROUP BY Clause. This conditional clause returns rows where aggregate function results matched with given conditions only. It added in the SQL because WHERE Clause cannot be combined with aggregate results, so it has a different purpose. The primary purpose of the WHERE Clause is to deal with non-aggregated or individual records.



In above example, Table is grouped based on DeptID column and these grouped rows filtered using HAVING Clause with condition AVG(Salary) > 3000.

5. TOP Clause:

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause can be very useful on large tables with thousands of records.

Returning a large number of records can impact on performance

Syntax: `SELECT TOP 5 * FROM TABLENAME;`

6. SQL AND & OR Operators:

The AND & OR operators are used to filter records based on more than one condition.

The AND operator displays a record if both the first condition AND the second condition are true.

The OR operator displays a record if either the first condition OR the second condition is true.

Syntax:
```
SELECT * FROM Tablename WHERE column='somevalue' AND column='somevalue';
```

SQL JOINS

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them.

The most common type of join is: SQL INNER JOIN (simple join). An SQL INNER JOIN return all rows from multiple tables where the join condition is met.

There are many types of join.

1. **Inner Join**
2. **Equi-join**
3. **Outer Join**

   Left outer Join

   Right outer join

   Full outer join

4. **Cross Join**
5. **Self Join**

Join is very useful to fetching records from multiple tables with reference to common column between them.

To understand join with example, we have to create two tables in SQL Server database.

Employee

```
create table Employee(
id int identity(1,1) primary key,
Username varchar(50),
FirstName varchar(50),
 LastName varchar(50),
 DepartID int
 )
 Departments
```

```
create table Departments(
id int identity(1,1) primary key,
DepartmentName varchar(50)
)
```

1. Inner Join

The join that displays only the rows that have a match in both the joined tables is known as inner join.

syntax:

```
select e1.Username,e1.FirstName,e1.LastName,e2.DepartmentName
from Employee e1
inner join Departments e2
on e1.DepartID=e2.id
```

It gives matched rows from both tables with reference to `DepartID` of first table and id of second table like this.

2. Equi-Join

Equi join is a special type of join in which we use only equality operator. Hence, when you make a query for join using equality operator, then that join query comes under Equi join.

Equi join has only (=) operator in join condition.

Equi join can be inner join, left outer join, right outer join.

Check the query for equi-join:

syntax:

```
SELECT * FROM Employeee e1 JOIN Departmentss e2 ON e1.DepartID = e2.id
```

3. Outer Join

Outer join returns all the rows of both tables whether it has matched or not.

We have three types of outer join:

Left outer join

Right outer join

Full outer join

a) Left Outer join

Left join displays all the rows from first table and matched rows from second table like that..

syntax:

```
 SELECT * FROM Employee e1 LEFT OUTER JOIN Departments e2
ON e1.DepartID = e2.id
```

b) Right outer join

Right outer join displays all the rows of second table and matched rows from first table like that.

```
SELECT * FROM Employee e1 RIGHT OUTER JOIN Departments e2
ON e1.DepartID = e2.id
```

C) Full outer join

Full outer join returns all the rows from both tables whether it has been matched or not.

```
SELECT * FROM Employee e1 FULL OUTER JOIN Departments e2
ON e1.DepartID = e2.id
```

4. Cross Join

A cross join that produces Cartesian product of the tables that are involved in the join.

The size of a Cartesian product is the number of the rows in the first table multiplied by the number of rows in the second table like this.

```
SELECT * FROM Employee cross join Departments e2
```

You can write a query like this also

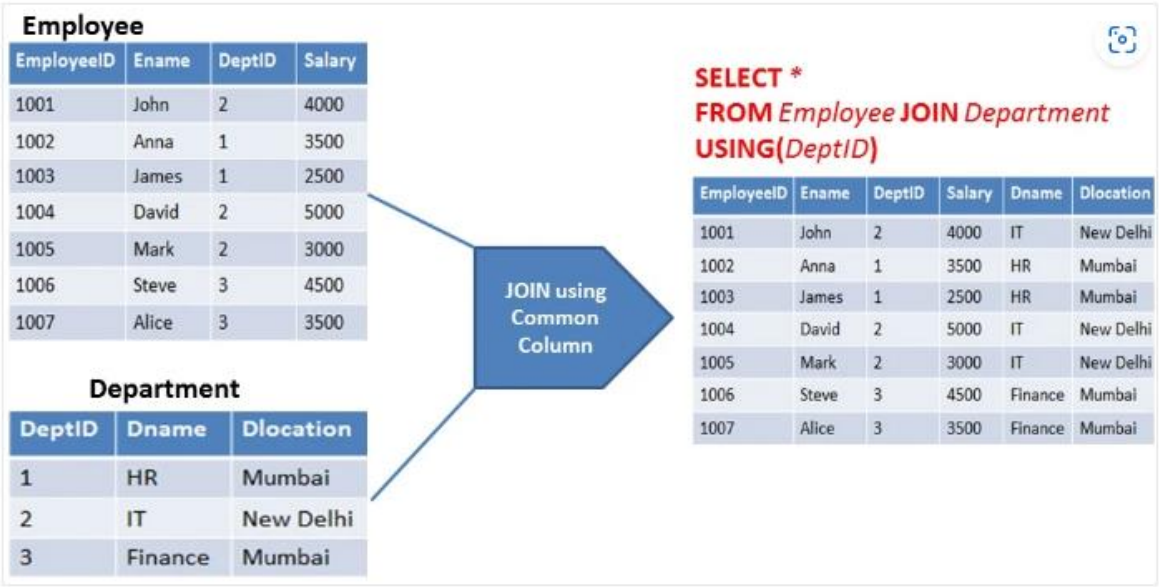
```
SELECT * FROM Employee , Departments e2
```

5. Self Join

Joining the table itself called self join. Self join is used to retrieve the records having some relation or similarity with other records in the same table. Here, we need to use aliases for the same table to set a **self join** between single table and retrieve records satisfying the condition in where clause.

```
 SELECT e1.Username,e1.FirstName,e1.LastName from Employee e1
inner join Employee e2 on e1.id=e2.DepartID
```

Group BY with join Exaple:



Functions:

- Function is a database object in Sql Server. Basically, it is a set of sql statements that accepts only input parameters, perform actions and return the result.
- Function can return only single value or a table.
- We can't use function to Insert, Update, Delete records in the database table(s)

Types of Functions:

1. System Defined Function

2. User Defined Function

1. **<u>System Defined Function:</u>**

   These functions are defined by Sql Server for different purpose.

   We have two types of system defined function in Sql Server.

- **Scalar Function:**

  Scalar functions operate on a single value and returns a single value. Below is the list of some useful

  Sql Server Scalar functions.

  Scalar Function Description:

abs(-10.67)	This returns absolute number of the given number means 10.67.
rand(10)	This will generate random number of 10 characters.
round(17.56719,3)	This will round off the given number to 3 places of decimal means 17.567
upper('dotnet')	This will returns upper case of given string means 'DOTNET'
lower('DOTNET')	This will returns lower case of given string means 'dotnet'
ltrim(' dotnet')	This will remove the spaces from left hand side of 'dotnet' string.
convert(int, 15.56)	This will convert the given float value to integer means 15.

- **Aggregate Function**

Aggregate functions operate on a collection of values and returns a single value. Below is the list of some useful Sql Server Aggregate functions.

Aggregate Functions	Description
max()	This returns maximum value from a collection of values.
min()	This returns minimum value from a collection of values.
avg()	This returns average of all values in a collection.
count()	This returns no of counts from a collection of values.

- **string functions:**

ASCII(CHARACTER_EXPRESSION)	Returns the ASCII code of the given character expression.
CHAR(INTEGER_EXPRESSION)	Converts an int ASCII code to a character. The INTEGER_EXPRESSION should be between 0 and 255.
LTRIM (CHARACTER_EXPRESSION)	Removes blanks on the left hand side of the given character expression.
RTRIM(CHARACTER_EXPRESSION)	Removes blanks on the right hand side of the given character expression.

LOWER (CHARACTER_EXPRESSION):	Converts all the characters in the given CHARACTER_EXPRESSION to lowercase letters.
UPPER (CHARACTER_EXPRESSION)	Converts all the characters in the given CHARACTER_EXPRESSION to uppercase letters.
REVERSE('Any String Expression')	Reverse all the characters in the given string expression.
LEN(String Expression) :	Returns the count of total character in the given string expression excluding the blanks at the end of the expression.

Built in string functions in sql server:

There are several built-in functions. In this session, we will look at the most common string functions.

ASCII(Character_Expression) - Returns the ASCII code of the given character expression.

To find the ACII Code of capital letter 'A'

Example: Select ASCII('A')

Output: 65

CHAR(Integer_Expression) - Converts an int ASCII code to a character. The Integer_Expression, should be between 0 and 255.

The following SQL, prints all the characters for the ASCII values from o thru 255

```
 Declare @Number int
 Set @Number = 100
 While(@Number <= 255)
 Begin
 Print CHAR(@Number)
 Set @Number = @Number + 1
End
```

Note: The while loop will become an infinite loop, if you forget to include the following line.

Set @Number = @Number + 1

Printing uppercase alphabets using CHAR() function:

```
Declare @Number int
Set @Number = 65
While(@Number <= 90)
Begin
Print CHAR(@Number)
 Set @Number = @Number + 1
End
```

Printing lowercase alphabets using CHAR() function:

```
Declare @Number int
       Set @Number = 97
       While(@Number <= 122)
       Begin
       Print CHAR(@Number)
        Set @Number = @Number + 1
End
```

Another way of printing lower case alphabets using CHAR() and LOWER() functions.

```
  Declare @Number int
  Set @Number = 65
  While(@Number <= 90)
Begin
 Print LOWER(CHAR(@Number))
Set @Number = @Number + 1
End
```

LTRIM(Character_Expression) - Removes blanks on the left handside of the given character expression.

Example: Removing the 3 white spaces on the left hand side of the '   Hello' string using LTRIM() function.

```
Select LTRIM('   Hello')
```

Output: Hello

RTRIM(Character_Expression) - Removes blanks on the right hand side of the given character expression.

Example: Removing the 3 white spaces on the left hand side of the 'Hello   ' string using RTRIM() function.

```
Select RTRIM('Hello   ')
```

Output: Hello

Example: To remove white spaces on either sides of the given character expression, use LTRIM() and RTRIM() as shown below.

```
Select LTRIM(RTRIM('   Hello   '))
```

Output: Hello

LOWER(Character_Expression) - Converts all the characters in the given Character_Expression, to lowercase letters.

Example:
```
Select LOWER('CONVERT This String Into Lower Case')
```

Output**:** convert this string into lower case

UPPER(Character_Expression) - Converts all the characters in the given Character_Expression, to uppercase letters.

Example:
```
Select UPPER('CONVERT This String Into upper Case')
```

Output**:** CONVERT THIS STRING INTO UPPER CASE

REVERSE('Any_String_Expression') - Reverses all the characters in the given string expression.

Example:
```
Select REVERSE('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

Output: ZYXWVUTSRQPONMLKJIHGFEDCBA

LEN(String_Expression) - Returns the count of total characters, in the given string expression, excluding the blanks at the end of the expression.

Example:
```
Select LEN('SQL Functions   ')
```

Output: 13

LEFT, RIGHT, CHARINDEX and SUBSTRING functions -

LEFT(Character_Expression, Integer_Expression) - Returns the specified number of characters from the left hand side of the given character expression.

Example:
```
Select LEFT('ABCDE', 3)
```

Output: ABC

RIGHT(Character_Expression, Integer_Expression) - Returns the specified number of characters from the right hand side of the given character expression.

Example:
```
Select RIGHT('ABCDE', 3)
```

Output: CDE

CHARINDEX('Expression_To_Find', 'Expression_To_Search', 'Start_Location') - Returns the starting position of the specified expression in a character string. Start_Location parameter is optional.

Example: In this example, we get the starting position of '@' character in the email string 'sara@aaa.com'.

```
Select CHARINDEX('@','sara@aaa.com',1)
```

Output: 5

SUBSTRING('Expression', 'Start', 'Length') - As the name, suggests, this function returns substring (part of the string), from the given expression. You specify the starting location using the 'start' parameter and the number of characters in the substring using 'Length' parameter. All the 3 parameters are mandatory.

Example: Display just the domain part of the given email 'John@bbb.com'.

```
Select SUBSTRING('John@bbb.com',6, 7)
```

Output: bbb.com

In the above example, we have hardcoded the starting position and the length parameters. Instead of hardcoding we can dynamically retrieve them using CHARINDEX() and LEN() string functions as shown below.

Example:

```
Select SUBSTRING('John@bbb.com',(CHARINDEX('@', 'John@bbb.com') + 1),
   (LEN('John@bbb.com') - CHARINDEX('@','John@bbb.com')))
```

Output: bbb.com

DateTime functions in SQL Server:

1. DateTime data types

2. DateTime functions available to select the current system date and time

3. Understanding concepts - UTC time and Time Zone offset

There are several built-in DateTime functions available in SQL Server.

All the following functions can be used to get the current system date and time, where you have

sql server installed.

Function	Date Time Format	Description
GETDATE()	2012-08-31 20:15:04.543	Commonly used function
CURRENT_TIMESTAMP	2012-08-31 20:15:04.543	ANSI SQL equivalent to GETDATE
SYSDATETIME()	2012-08-31 20:15:04.5380028	More fractional seconds precision
SYSDATETIMEOFFSET()	2012-08-31 20:15:04.5380028 + 01:00	More fractional seconds precision + Time zone offset

GETUTCDATE()	2012-08-31 19:15:04.543	UTC Date and Time
SYSUTCDATETIME()	2012-08-31 19:15:04.5380028	UTC Date and Time, with More fractional seconds precision

Note: UTC stands for Coordinated Universal Time, based on which, the world regulates clocks and time.

There are slight differences between GMT and UTC, but for most common purposes, UTC is synonymous with GMT.

To practically understand how the different date time datatypes available in SQL Server, store data, create the sample table tblDateTime.

```
CREATE TABLE [tblDateTime]
(
[c_time] [time](7) NULL,
 [c_date] [date] NULL,
 [c_smalldatetime] [smalldatetime] NULL,
 [c_datetime] [datetime] NULL,
 [c_datetime2] [datetime2](7) NULL,
 [c_datetimeoffset] [datetimeoffset](7) NULL
)
```

To Insert some sample data, execute the following query.

```
INSERT INTO tblDateTime VALUES
  (GETDATE(),GETDATE(),GETDATE(),GETDATE(),GETDATE(),GETDATE())
```

Now, issue a select statement, and you should see, the different types of datetime datatypes, storing the current datetime, in different formats.

DatePart, DateAdd and DateDiff functions in SQL Server -

DatePart(DatePart, Date) - Returns an integer representing the specified DatePart.

This function is similar to DateName(). DateName() returns nvarchar, where as DatePart() returns an integer.

The valid DatePart parameter values are shown below.

Examples:

```
Select DATEPART(weekday, '2012-08-30 19:45:31.793') -- returns 5

Select DATENAME(weekday, '2012-08-30 19:45:31.793') -- returns Thursday
```

DATEADD (datepart, NumberToAdd, date) - Returns the DateTime, after adding specified NumberToAdd, to the datepart specified of the given date.

Examples:

```
Select DateAdd(DAY, 20, '2012-08-30 19:45:31.793')
```

-- Returns 2012-09-19 19:45:31.793

```
Select DateAdd(DAY, -20, '2012-08-30 19:45:31.793')
```

-- Returns 2012-08-10 19:45:31.793

DATEDIFF(datepart, startdate, enddate) - Returns the count of the specified datepart boundaries crossed between the specified startdate and enddate.

Examples:

```
Select DATEDIFF(MONTH, '11/30/2005','01/31/2006') -- returns 2
Select DATEDIFF(DAY, '11/30/2005','01/31/2006') -- returns 62
```

Cast and Convert functions in SQL Server:

To convert one data type to another, CAST and CONVERT functions can be used.

Syntax of CAST and CONVERT functions from MSDN:

CAST (expression AS data_type [(length)])

CONVERT (data_type [(length)] , expression [, style])

From the syntax, it is clear that CONVERT() function has an optional style parameter, where as CAST() function lacks this capability.

The following are the differences between the 2 functions.

1. Cast is based on ANSI standard and Convert is specific to SQL Server. So, if portability is a concern and if you want to use the script with other database applications, use Cast().

2. Convert provides more flexibility than Cast. For example, it's possible to control how you want Date Time datatypes to be converted using styles with convert function.

Mathematical functions in sql server:

we will understand the commonly used mathematical functions in sql server like, Abs, Ceiling, Floor, Power, Rand, Square, Sqrt, and Round functions

ABS (numeric_expression) - ABS stands for absolute and returns, the absolute (positive) number.

Example:

```
Select ABS(-101.5) -- returns 101.5, without the - sign.
```

CEILING (numeric_expression) and FLOOR (numeric_expression)

CEILING and FLOOR functions accept a numeric expression as a single parameter.

CEILING() returns the smallest integer value greater than or equal to the parameter,

whereas FLOOR() returns the largest integer less than or equal to the parameter.

Examples:

```
Select CEILING(15.2) -- Returns 16
Select CEILING(-15.2) -- Returns -15
Select FLOOR(15.2) -- Returns 15
Select FLOOR(-15.2) -- Returns -16
```

Power(expression, power) - Returns the power value of the specified expression to the specified power.

Example: The following example calculates '2 TO THE POWER OF 3' = 2*2*2 = 8

```
Select POWER(2,3) -- Returns 8
```

RAND([Seed_Value]) - Returns a random float number between 0 and 1. Rand() function takes an optional seed parameter. When seed value is supplied the

RADN() function always returns the same value for the same seed.

Example:

```
Select RAND(1) -- Always returns the same value
```

If you want to generate a random number between 1 and 100, RAND() and FLOOR() functions can be used as shown below. Every time, you execute this query, you get a random number between 1 and 100.

```
Select FLOOR(RAND() * 100)
```

The following query prints 10 random numbers between 1 and 100.

```
 Declare @Counter INT
 Set @Counter = 1
 While(@Counter <= 10)
 Begin
 Print FLOOR(RAND() * 100)
 Set @Counter = @Counter + 1
End
```

SQUARE (Number) - Returns the square of the given number.

Example:

```
Select SQUARE(9) -- Returns 81
```

SQRT (Number) - SQRT stands for Square Root. This function returns the square root of the given value.

Example:

```
Select SQRT(81) -- Returns 9
```

ROUND (numeric_expression , length [,function]) - Rounds the given numeric expression based on the given length. This function takes 3 parameters.

1. Numeric_Expression is the number that we want to round.

2. Length parameter, specifies the number of the digits that we want to round to. If the length is a positive number, then the rounding is applied for the decimal part, where as if the length is negative, then the rounding is applied to the number before the decimal.

3. The optional function parameter, is used to indicate rounding or truncation operations. A value of 0, indicates rounding, where as a value of non zero indicates truncation. Default, if not specified is 0.

Examples:

-- Round to 2 places after (to the right) the decimal point

```
Select ROUND(850.556, 2) -- Returns 850.560
```

-- Truncate anything after 2 places, after (to the right) the decimal point

```
Select ROUND(850.556, 2, 1) -- Returns 850.550
```

-- Round to 1 place after (to the right) the decimal point

```
Select ROUND(850.556, 1) -- Returns 850.600
```

-- Truncate anything after 1 place, after (to the right) the decimal point

```
Select ROUND(850.556, 1, 1) -- Returns 850.500
```

-- Round the last 2 places before (to the left) the decimal point

```
Select ROUND(850.556, -2) -- 900.000
```

-- Round the last 1 place before (to the left) the decimal point

```
Select ROUND(850.556, -1) -- 850.000
```

The general guideline is to use CAST(), unless you want to take advantage of the style functionality in CONVERT().

2. User Defined Function:

These functions are created by user in system database or in user defined database.

We three types of user defined functions.

(1).Scalar Function

(2).Inline Table-Valued Function

(3).Multi-Statement Table-Valued Function

Scalar Function:

User defined scalar function also returns single value as a result of actions perform by function.

We return any datatype value from function.

--Create a table

```
CREATE TABLE Employee
(
 EmpID int PRIMARY KEY,
FirstName varchar(50) NULL,
 LastName varchar(50) NULL,
 Salary int NULL,
Address varchar(100) NULL,
)
```

--Insert Data

```
Insert into Employee(EmpID,FirstName,LastName,Salary,Address)
Values(1,'Mohan','Chauahn',22000,'Delhi');
Insert into Employee(EmpID,FirstName,LastName,Salary,Address)
Values(2,'Asif','Khan',15000,'Delhi');
Insert into Employee(EmpID,FirstName,LastName,Salary,Address)
Values(3,'Bhuvnesh','Shakya',19000,'Noida');
Insert into Employee(EmpID,FirstName,LastName,Salary,Address)
Values(4,'Deepak','Kumar',19000,'Noida');
```

--See created table

Select * from Employee

--Create function to get emp full name

```
Create function fnGetEmpFullName
(
 @FirstName varchar(50),
 @LastName varchar(50)
)
returns varchar(101)
As
Begin return (Select @FirstName + ' '+ @LastName);
end
```

--Calling the above created function

Select dbo.fnGetEmpFullName(FirstName,LastName) as Name, Salary from Employee

(2). Inline Table-Valued Function

User defined inline table-valued function returns a table variable as a result of actions perform by function.

The value of table variable should be derived from a single SELECT statement.

--Create function to get employees

```
Create function fnGetEmployee()
returns Table
As
  return (Select * from Employee)
```

--Now call the above created function

Select * from fnGetEmployee()

(3). Multi-Statement Table-Valued Function

User defined multi-statement table-valued function returns a table variable as a result of actions

perform by function. In this a table variable must be explicitly declared and defined whose value can be derived

from a multiple sql statement.

--Create function for EmpID,FirstName and Salary of Employee

```
Create function fnGetMulEmployee()
returns @Emp Table
(
EmpID int,
FirstName varchar(50),
Salary int
)
As
begin
 Insert into @Emp Select e.EmpID,e.FirstName,e.Salary from Employee e;
--Now update salary of first employee
 update @Emp set Salary=25000 where EmpID=1;
--It will update only in @Emp table not in Original Employee table
return
end
```

--Now call the above created function

Select * from fnGetMulEmployee()

--Now see the original table. This is not affected by above function update command

Select * from Employee

Note:

*Function returns only single value.

*Function accepts only input parameters.

*Function is not used to Insert, Update, Delete data in database table(s).

*Function can be nested up to 32 level.

*User Defined Function can have upto 1023 input parameters while a Stored Procedure can have upto 2100 input parameters.

*User Defined Function can't returns XML Data Type.

*User Defined Function doesn't support Exception handling.

*User Defined Function can call only Extended Stored Procedure.

*User Defined Function doesn't support set options like set ROWCOUNT etc.

Examples:

A Statement that creates a scalar - valued function:

Here we create a user-defined (Scalar-valued function) which calculates the amount of a company.

It is a simple function and doesn't need a parameter; see:

```
CREATE FUNCTION fnbal_invoice()
RETURNS MONEY
BEGIN
 RETURN(SELECT SUM(invoicetotal-paymenttotal-creadittotal)
 FROM dbo.mcninvoices
        WHEREinvoicetotal-paymenttotal-creadittotal> 0 )
END
```

Aggregate Functions:

```
SELECT AVG(invoicetotal) FROM [dbo].[mcninvoices];
SELECT COUNT (invoiceid) FROM mcninvoices;
 SELECT avg (DISTINCT paymenttotal) FROM mcninvoices;
 select MAX(creadittotal) from mcninvoices
 select MIN(creadittotal) from mcninvoices
select sum(creadittotal) from mcninvoices
```

Scalar User Defined Functions in sql server -

we have learnt how to use many of the built-in system functions that are available in SQL Server.

In this session, we will turn our attention, to creating user defined functions. In short UDF.

We will cover

1. User Defined Functions in sql server

2. Types of User Defined Functions

3. Creating a Scalar User Defined Function

4. Calling a Scalar User Defined Function

5. Places where we can use Scalar User Defined Function

6. Altering and Dropping a User Defined Function

In SQL Server there are 3 types of User Defined functions

1. Scalar functions

2. Inline table-valued functions

3. Multistatement table-valued functions

Scalar functions may or may not have parameters, but always return a single (scalar) value. The returned value can be of any data type, except text, ntext, image, cursor, and timestamp.

To create a function, we use the following syntax:

```
CREATE FUNCTION Function_Name(@Parameter1 DataType, @Parameter2
DataType,..@Parametern Datatype)
RETURNS Return_Datatype
AS
BEGIN
Function Body
Return Return_Datatype
END
```

Let us now create a function which calculates and returns the age of a person. To compute the age we require, date of birth. So, let's pass date of birth as a parameter. So, AGE() function returns an integer and accepts date parameter.

```
CREATE FUNCTION Age(@DOB Date)
RETURNS 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
```

When calling a scalar user-defined function, you must supply a two-part name, OwnerName.FunctionName. dbo stands for database owner.

```
Select dbo.Age( dbo.Age('10/08/1982')
```

You can also invoke it using the complete 3 part name, DatabaseName.OwnerName.FunctionName.

```
Select SampleDB.dbo.Age('10/08/1982')
```

Stored Procedures

The classification of stored procedures is depends on the Where it is Stored.

Based on this you can divide it in 4 sections.

1.System stored procedures

2.Local stored procedures

3.Temporary stored procedures

4.Extended stored procedures

1.System Stored Procedures:

System stored procedures are stored in the Master database and are typically named with a sp_ prefix. They can be used to perform variety of tasks to support SQL Server functions that support external application calls for data in the system tables, general system procedures for database administration, and security management functions.

For example, you can view the contents of the stored procedure by calling

```
sp_helptext [StoredProcedure_Name].
```

There are hundreds of system stored procedures included with SQL Server. For a complete list of system stored procedures, refer to "System Stored Procedures" in SQL Server Books Online.

2.Local stored procedures

Local stored procedures are usually stored in a user database and are typically designed to complete tasks in the database in which they reside. While coding these procedures don't use sp_ prefix to you stored procedure it will create a performance bottleneck. The reason is when you can any procedure that is prefixed with sp_ it will first look at in the mater database then comes to the user local database.

3.Temporary stored procedures

A temporary stored procedure is all most equivalent to a local stored procedure, but it exists only as long as SQL Server is running or until the connection that created it is not closed. The stored procedure is deleted at connection termination or at server shutdown. This is because temporary stored procedures are stored in the TempDB database. TempDB is re-created when the server is restarted.

4.Extended Stored Procedures

An extended stored procedure uses an external program, compiled as a 32-bit dynamic link library (DLL), to expand the capabilities of a stored procedure. A number of system stored procedures are also classified as extended stored procedures. For example, the xp_sendmail program, which sends a message and a query result set attachment to the specified e-mail recipients, is both a system stored procedure and an extended stored procedure. Most extended stored procedures use the xp_ prefix as a naming convention. However, there are some extended stored procedures that use the sp_ prefix, and there are some system stored procedures that are not extended and use the xp_ prefix. Therefore, you cannot depend on naming conventions to identify system stored procedures and extended stored procedures.

A stored procedure is group of T-SQL (Transact SQL) statements. 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

call it just by it's name.

There are several advantages of using stored procedures, which we will discuss in a later session.

In this session, we will learn how to create, execute, change and delete stored procedures.

Creating a simple stored procedure without any parameters:

This stored procedure, retrieves Name and Gender of all the employees.

To create a stored procedure we use, CREATE PROCEDURE or CREATE PROC statement.

```
Create Procedure spGetEmployees
as
Begin
  Select Name, Gender from tblEmployee
End
```

To execute the stored procedure, you can just type the procedure name and press F5,

or use EXEC or EXECUTE keywords followed by the procedure name as shown below.

```
1. spGetEmployees
2. EXEC spGetEmployees
3. Execute spGetEmployees
```

Creating a stored procedure with input parameters:

This SP, accepts GENDER and DEPARTMENTID parameters. Parameters and variables have an @ prefix in their name.

```
Create Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
  @DepartmentId int
  as
  Begin
    Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId =
@DepartmentId
End
```

To invoke this procedure, we need to pass the value for @Gender and @DepartmentId parameters.

If you don't specify the name of the parameters, you have to first pass value for @Gender parameter

and then for @DepartmentId.

```
EXECUTE spGetEmployeesByGenderAndDepartment 'Male', 1
```

On the other hand, if you change the order, you will get an error stating "Error converting data type varchar

to int." This is because, the value of "Male" is passed into @DepartmentId parameter. Since @DepartmentId is

an integer, we get the type conversion error.

spGetEmployeesByGenderAndDepartment 1, 'Male'

When you specify the names of the parameters when executing the stored procedure the order doesn't matter.

EXECUTE spGetEmployeesByGenderAndDepartment @DepartmentId=1, @Gender = 'Male'

To view the text, of the stored procedure

1. Use system stored procedure sp_helptext 'SPName'

OR

2. Right Click the SP in Object explorer -> Scrip Procedure as -> Create To -> New Query Editor Window

To change the stored procedure, use ALTER PROCEDURE statement:

```
Alter Procedure spGetEmployeesByGenderAndDepartment
@Gender nvarchar(50),
@DepartmentId int
as
Begin
   Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId =
@DepartmentId order by Name
End
```

To encrypt the text of the SP, use WITH ENCRYPTION option. Once, encrypted, you cannot view the text of the procedure, using sp_helptext system stored procedure. There are ways to obtain the original text, which we will talk about in a later session.

```
Alter Procedure spGetEmployeesByGenderAndDepartment
 @Gender nvarchar(50),
 @DepartmentId int
 WITH ENCRYPTION
```

```
        as
        Begin
        Select Name, Gender from tblEmployee Where Gender = @Gender and DepartmentId =
@DepartmentId
      End
```

To delete the SP, use DROP PROC 'SPName' or DROP PROCEDURE 'SPName'

Stored procedures with output parameters:-

To create an SP with output parameter, we use the keywords OUT or OUTPUT.

@EmployeeCount is an OUTPUT parameter. Notice, it is specified with OUTPUT keyword.

```
Create Procedure spGetEmployeeCountByGender
  @Gender nvarchar(20),
  @EmployeeCount int Output
  as
  Begin
  Select @EmployeeCount = COUNT(Id)
  from tblEmployee
  where Gender = @Gender
End
```

To execute this stored procedure with OUTPUT parameter

1. First initialise a variable of the same datatype as that of the output parameter.

We have declared @EmployeeTotal integer variable.

2. Then pass the @EmployeeTotal variable to the SP. You have to specify the OUTPUT keyword.

If you don't specify the OUTPUT keyword, the variable will be NULL.

3. Execute

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender 'Female', @EmployeeTotal output
Print @EmployeeTotal
```

If you don't specify the OUTPUT keyword, when executing the stored procedure,

the @EmployeeTotal variable will be NULL. Here, we have not specified OUTPUT keyword. When you execute,

you will see '@EmployeeTotal is null' printed.

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender 'Female', @EmployeeTotal
if(@EmployeeTotal is null)
 Print '@EmployeeTotal is null'
else
  Print '@EmployeeTotal is not null'
```

You can pass parameters in any order, when you use the parameter names.

Here, we are first passing the OUTPUT parameter and then the input @Gender parameter.

```
Declare @EmployeeTotal int
Execute spGetEmployeeCountByGender @EmployeeCount = @EmployeeTotal OUT, @Gender
= 'Male'
Print @EmployeeTotal
```

The following system stored procedures, are extremely useful when working procedures.

sp_help SP_Name : View the information about the stored procedure, like parameter names,

their datatypes etc. sp_help can be used with any database object, like tables, views, SP's, triggers etc.

Alternatively, you can also press ALT+F1, when the name of the object is highlighted.

sp_helptext SP_Name : View the Text of the stored procedure

sp_depends SP_Name : View the dependencies of the stored procedure.

This system SP is very useful, especially if you want to check, if there are any stored procedures that

are referencing a table that you are abput to drop. sp_depends can also be used with other database objects like table etc.

Stored procedure output parameters or return values:-

1. Understand what are stored procedure return values

2. Difference between stored procedure return values and output parameters

3. When to use output parameters over return values

What are stored procedure status variables?

Whenever, you execute a stored procedure, it returns an integer status variable. Usually, zero indicates success,

and non-zero indicates failure. To see this yourself, execute any stored procedure from the object explorer, in sql server management studio.

1. Right Click and select 'Execute Stored Procedure

2. If the procedure, expects parameters, provide the values and click OK.

3. Along with the result that you expect, the stored procedure, also returns a Return Value = 0

So, from this we understood that, when a stored procedure is executed, it returns an integer status variable.

With this in mind, let's understand the difference between output parameters and RETURN values.

We will use the Employees table below for this purpose.

The following procedure returns total number of employees in the Employees table,

using output parameter - @TotalCount.

```
Create Procedure spGetTotalCountOfEmployees1
@TotalCount int output
as
Begin
  Select @TotalCount = COUNT(ID) from tblEmployee
End
```

```
Executing spGetTotalCountOfEmployees1 returns 3.
```

```
Declare @TotalEmployees int
Execute spGetTotalCountOfEmployees @TotalEmployees Output
Select @TotalEmployees
```

Re-written stored procedure using return variables:

```
Create Procedure spGetTotalCountOfEmployees2
as
Begin
  return (Select COUNT(ID) from Employees)
End
```

Executing spGetTotalCountOfEmployees2 returns 3.

```
Declare @TotalEmployees int
Execute @TotalEmployees = spGetTotalCountOfEmployees2
Select @TotalEmployees
```

So, we are able to achieve what we want, using output parameters as well as return values.

Now, let's look at example, where return status variables cannot be used, but Output parameters can be used.

In this SP, we are retrieving the Name of the employee, based on their Id, using the output parameter @Name.

```
Create Procedure spGetNameById1
@Id int,
@Name nvarchar(20) Output
as
Begin
 Select @Name = Name from tblEmployee Where Id = @Id
End
```

Executing spGetNameById1, prints the name of the employee

```
Declare @EmployeeName nvarchar(20)
Execute spGetNameById1 3, @EmployeeName out
Print 'Name of the Employee = ' + @EmployeeName
```

Now let's try to achieve the same thing, using return status

```
Create Procedure spGetNameById2
@Id int
as
Begin
 Return (Select Name from tblEmployee Where Id = @Id)
End

Declare @EmployeeName nvarchar(20)
Execute @EmployeeName = spGetNameById2 1
Print 'Name of the Employee = ' + @EmployeeName
```

So, using return values, we can only return integers, and that too, only one integer.

It is not possible, to return more than one value using return values, where as output parameters,

can return any datatype and an sp can have more than one output parameters. I always prefer, using output parameters, over RETURN values.

In general, RETURN values are used to indicate success or failure of stored procedure,

especially when we are dealing with nested stored procedures.Return a value of 0, indicates success, and any nonzero value indicates failure.

Difference between return values and output parameters:

Return status values

1)Only integer datatype

2)Only one value

3)Use to convey success or failure

output parameters:

1)Any datatype

2)More than one value

3)Use to return value like name count. Ect

VIEWS

***What is a View?**

A view is nothing more than a saved SQL query. A view can also be considered as a virtual table.

Let's understand views with an example. We will base all our examples on tblEmployee and tblDepartment tables.

SQL Script to create tblEmployee table:

```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Salary int,
  Gender nvarchar(10),
  DepartmentId int
)
```

SQL Script to create tblDepartment table:

```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)
```

```
Insert data into tblDepartment table
```

```
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')

Insert data into tblEmployee table
Insert into tblEmployee values (1,'murali', 5000, 'Male', 3)
Insert into tblEmployee values (2,'guru', 3400, 'Male', 2)
Insert into tblEmployee values (3,'swetha', 6000, 'Female', 1)
Insert into tblEmployee values (4,'sathya', 4800, 'Male', 4)
Insert into tblEmployee values (5,'sri', 3200, 'Female', 1)
Insert into tblEmployee values (6,'benny', 4800, 'Male', 3)
```

To get the expected output, we need to join tblEmployees table with tblDepartments table. If you are new to

joins, please click here to view the video on Joins in SQL Server.

```
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

Now let's create a view, using the JOINS query, we have just written.

```
Create View vWEmployeesByDepartment
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

To select data from the view, SELECT statement can be used the way, we use it with a table.

```
SELECT * from vWEmployeesByDepartment
```

When this query is executed, the database engine actually retrieves the data from the underlying base tables,

tblEmployees and tblDepartments. The View itself, doesnot store any data by default. However, we can change

this default behaviour, which we will talk about in a later session. So, this is the reason, a view is considered, as just, a stored query or a virtual table.

Advantages of using views:

1. Views can be used to reduce the complexity of the database schema, for non IT users. The sample view, vWEmployeesByDepartment, hides the complexity of joins. Non-IT users, finds it easy to query the view, rather than writing complex joins.

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

Row Level Security:

For example, I want an end user, to have access only to IT Department employees.

If I grant him access to the underlying tblEmployees and tblDepartments tables, he will be able to see, every department employees. To achieve this, I can create a view, which returns only IT Department employees, and grant the user access to the view and not to the underlying table.

View that returns only IT department employees:

```
Create View vWITDepartment_Employees
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
where tblDepartment.DeptName = 'IT'
```

Column Level Security:

Salary is confidential information and I want to prevent access to that column. To achieve this, we can create a view, which excludes the Salary column, and then grant the end user access to this views, rather than the base tables.

View that returns all columns except Salary column:

```
Create View vWEmployeesNonConfidentialData
as
Select Id, Name, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

3. Views can be used to present only aggregated data and hide detailed data.

View that returns summarized data, Total number of employees by Department.

```
Create View vWEmployeesCountByDepartment
as
Select DeptName, COUNT(Id) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
Group By DeptName
```

To look at view definition - sp_helptext vWName

To modify a view - ALTER VIEW statement

To Drop a view - DROP VIEW vWName

Updateable Views:-In this session we will learn about Updateable Views.

Let's create tblEmployees table and populate it with some sample data.

```
Create Table tblEmployee Script:
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Salary int,
  Gender nvarchar(10),
  DepartmentId int
)

Script to insert data:
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)
Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
Insert into tblEmployee values (4,'Todd', 4800, 'Male', 4)
Insert into tblEmployee values (5,'Sara', 3200, 'Female', 1)
Insert into tblEmployee values (6,'Ben', 4800, 'Male', 3)
```

Let's create a view, which returns all the columns from the tblEmployees table, except Salary column.

```
Create view vWEmployeesDataExceptSalary
as
Select Id, Name, Gender, DepartmentId
from tblEmployee
```

Select data from the view: A view does not store any data. So, when this query is executed, the database engine actually retrieves data, from the underlying tblEmployee base table.

```
Select * from vWEmployeesDataExceptSalary
```

Is it possible to Insert, Update and delete rows, from the underlying tblEmployees table, using view vWEmployeesDataExceptSalary?

Yes, SQL server views are updateable.

The following query updates, Name column from Mike to Mikey. Though, we are updating the view, SQL server, correctly updates the base table tblEmployee. To verify, execute, SELECT statement, on tblEmployee table.

```
Update vWEmployeesDataExceptSalary Set Name = 'Mikey' Where Id = 2
```

Along the same lines, it is also possible to insert and delete rows from the base table using views.

```
Delete from vWEmployeesDataExceptSalary where Id = 2
Insert into vWEmployeesDataExceptSalary values (2, 'Mikey', 'Male', 2)
```

Now, let us see, what happens if our view is based on multiple base tables. For this purpose, let's create tblDepartment table and populate with some sample data.

SQL Script to create tblDepartment table

```
CREATE TABLE tblDepartment
(
 DeptId int Primary Key,
 DeptName nvarchar(20)
)

--Insert data into tblDepartment table
Insert into tblDepartment values (1,'IT')
Insert into tblDepartment values (2,'Payroll')
Insert into tblDepartment values (3,'HR')
Insert into tblDepartment values (4,'Admin')
```

Create a view which joins tblEmployee and tblDepartment tables, and return the result as shown below

View that joins tblEmployee and tblDepartment

```
Create view vwEmployeeDetailsByDepartment
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```

Select Data from view vwEmployeeDetailsByDepartment

Select * from vwEmployeeDetailsByDepartment

Now, let's update, John's department, from HR to IT. At the moment, there are 2 employees (Ben, and John) in the HR department.

```
Update vwEmployeeDetailsByDepartment set DeptName='IT' where Name = 'John'
```

The UPDATE statement, updated DeptName from HR to IT in tblDepartment table, instead of upadting DepartmentId column in tblEmployee table. So, the conclusion - If a view is based on multiple tables, and if you update the view, it may not update the underlying base tables correctly. To correctly update a view, that is based on multiple table, INSTEAD OF triggers are used.

Indexed views in sql server:-

What is an Indexed View or What happens when you create an Index on a view?

A standard or Non-indexed view, is just a stored SQL query. When, we try to retrieve data from the view, the data is actually retrieved from the underlying base tables. So, a view is just a virtual table it does not store any data, by default.

However, when you create an index, on a view, the view gets materialized. This means, the view is now, capable of storing data. In SQL server, we call them Indexed views and in Oracle, Materialized views.

Let's now, look at an example of creating an Indexed view. For the purpose of this video, we will be using tblProduct and tblProductSales tables.Script to create table tblProduct

```
Create Table tblProduct
(
 ProductId int primary key,
 Name nvarchar(20),
 UnitPrice int
)

--Script to pouplate tblProduct, with sample data
Insert into tblProduct Values(1, 'Books', 20)
Insert into tblProduct Values(2, 'Pens', 14)
Insert into tblProduct Values(3, 'Pencils', 11)
Insert into tblProduct Values(4, 'Clips', 10)
```

Script to create table tblProductSales

```
Create Table tblProductSales
(
 ProductId int,
 QuantitySold int
)


--     Script to pouplate tblProductSales, with sample data
       Insert into tblProductSales values(1, 10)
       Insert into tblProductSales values(3, 23)
       Insert into tblProductSales values(4, 21)
       Insert into tblProductSales values(2, 12)
       Insert into tblProductSales values(1, 13)
       Insert into tblProductSales values(3, 12)
       Insert into tblProductSales values(4, 13)
       Insert into tblProductSales values(1, 11)
       Insert into tblProductSales values(2, 12)
       Insert into tblProductSales values(1, 14)
```

Script to create view vWTotalSalesByProduct

```
Create view vWTotalSalesByProduct
with SchemaBinding
as
Select Name,
SUM(ISNULL((QuantitySold * UnitPrice), 0)) as TotalSales,
COUNT_BIG(*) as TotalTransactions
from dbo.tblProductSales
join dbo.tblProduct
```

```
on dbo.tblProduct.ProductId = dbo.tblProductSales.ProductId
group by Name
```

If you want to create an Index, on a view, the following rules should be followed by the view.

1. The view should be created with SchemaBinding option

2. If an Aggregate function in the SELECT LIST, references an expression, and if there is a possibility for

that expression to become NULL, then, a replacement value should be specified. In this example, we are using,

ISNULL() function, to replace NULL values with ZERO.

3. If GROUP BY is specified, the view select list must contain a COUNT_BIG(*) expression

4. The base tables in the view, should be referenced with 2 part name. In this example, tblProduct and

tblProductSales are referenced using dbo.tblProduct and dbo.tblProductSales respectively.

Now, let's create an Index on the view:

The first index that you create on a view, must be a unique clustered index. After the unique clustered index has been created, you can create additional nonclustered indexes.

Create Unique Clustered Index UIX_vWTotalSalesByProduct_Name

on vWTotalSalesByProduct(Name)

Since, we now have an index on the view, the view gets materialized. The data is stored in the view. So when we execute Select * from vWTotalSalesByProduct, the data is retrurned from the view itself, rather than retrieving data from the underlying base tables. Indexed views, can significantly improve the performance of queries that involves JOINS and Aggeregations. The cost of maintaining an indexed view is much higher than the cost of maintaining a table index.

Indexed views are ideal for scenarios, where the underlying data is not frequently changed. Indexed views are more often used in OLAP systems, because the data is mainly used for reporting and analysis purposes. Indexed views, may not be suitable for OLTP systems, as the data is frequently addedd and changed.

Limitations of views:-

1. You cannot pass parameters to a view. Table Valued functions are an excellent replacement for parameterized views.

We will use tblEmployee table for our examples. SQL Script to create tblEmployee table:

```
CREATE TABLE tblEmployee
(
  Id int Primary Key,
  Name nvarchar(30),
  Salary int,
  Gender nvarchar(10),
  DepartmentId int
)

Insert data into tblEmployee table
Insert into tblEmployee values (1,'John', 5000, 'Male', 3)
Insert into tblEmployee values (2,'Mike', 3400, 'Male', 2)
Insert into tblEmployee values (3,'Pam', 6000, 'Female', 1)
Insert into tblEmployee values (4,'Todd', 4800, 'Male', 4)
Insert into tblEmployee values (5,'Sara', 3200, 'Female', 1)
Insert into tblEmployee values (6,'Ben', 4800, 'Male', 3)
```

```
Create View vWEmployeeDetails
@Gender nvarchar(20)
as
Select Id, Name, Gender, DepartmentId
from  tblEmployee
where Gender = @Gender
```

Table Valued functions can be used as a replacement for parameterized views.

```
Create function fnEmployeeDetails(@Gender nvarchar(20))
Returns Table
as
Return
(Select Id, Name, Gender, DepartmentId
from tblEmployee where Gender = @Gender)
```

Calling the function

Select * from dbo.fnEmployeeDetails('Male')

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.

```
Create View vWEmployeeDetailsSorted
as
Select Id, Name, Gender, DepartmentId
from tblEmployee
order by Id
```

4. Views cannot be based on temporary tables.

```
Create Table ##TestTempTable(Id int, Name nvarchar(20), Gender nvarchar(10))
Insert into ##TestTempTable values(101, 'Martin', 'Male')
Insert into ##TestTempTable values(102, 'Joe', 'Female')
Insert into ##TestTempTable values(103, 'Pam', 'Female')
```

PROVIDED BY VINODH

```
Insert into ##TestTempTable values(104, 'James', 'Male')
```

-- Error: Cannot create a view on Temp Tables

```
Create View vwOnTempTable
as
Select Id, Name, Gender
from ##TestTempTable
```