SQL Queries

1) Create

- Create [Database / Table / View] [Name]
- Create Table (name)

```
    Create table tblPerson
 -> (
 -> ID int not null Identity([identity Increment value, Identity seed value]) primary key,
 -> name nvarchar(50) not null,
 -> email nvarchar(50) not null,
 -> Gender int null
 -> );
```

2) Alter

- **Modifying a name**

------ Alter [database / Table] [Existing Name] Modify name = [new name]

---- **using Inbuilt query**

    Sp_renameDB '[existing database name]' , '[new db name]'

- **Setting a database to single user mode and then dropping it**

    Alter Database [name] Set Single_User with Rollback immediate

- **Adding a new Column to a table with a default constraint to an existing table**

    Alter table [table name] add [column name] [data type] [null / not null]
    constraint [constraint name] default [default value]

- **Adding a Foreign key Constraint**

    Alter table [Foreign Key table name] add constraint [Fk Constarint name]
    Foreign Key ([Foreign key column name]) References [Primary key table name]
    ([primary key column name])

Note: Foreign key table name is the table that needs to e a foreign key.

- **Adding a Default Constraint for a column**

    Alter table [table name] add constraint [DK constraint name]
    Default [default value] for [existiing table column name]

- **Dropping a constraint**

    Alter table [table name]
    Drop Constraint [constraint name]

- **Adding a Check constraint for a column**
  Alter table [table name] add constraint [constraint name]
  Check ([Boolean Expression])

Example: Alter table tbl_person add constarint CK_tblperson_Age
Check (Age >0 AND Age<100)
Here age is a column in a table.

- **Adding a unique key constraint**
  Alter table [table name] add constraint
  Unique [column name]

3) Drop

- **Dropping a db / table**
  Drop [DB /table] [name]

4) Use

- Use [DBname] Go (command)

5) Insert

- Insert into [table name] ([column1, column2,….]) values ([value1, value2,….])

6) Update

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

7) Delete

```
DELETE FROM table_name WHERE condition;
```

8) Select

- Select * from [table name]
- Select [Column name1, column name 2] from [table name]

Note: * represents all columns in a table.

- Select Distinct [column name 1, column name 2] from[table name]
- Select * from [table name] where [column name 1] = [value 1]

Note: Operators that can be used with where Clause

Operator	**Purpose**
=	Equal to
!= or <>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
IN	Specify a list of values

Between	Specify a range
Like	Specify a pattern
Not	Not a list, range etc…
%	Specifies zero or more character
-	Specifies exactly one character
[]	Any characters within the bracket
[^]	Not any characters within the bracket

- Select * from [table name] order by [column name1] [DESC | ASC], [column name 2] [DESC | ASC]
- Select top [row value] [column1] [column 2] from [table name]

  OR
- Select top [row value] Percent from [table name]
- Select [column1, column2..] SUM([column4]) AS [alias name] from [table name] Group By[column1, colum2…]

Note:
1) Group By clause must be applied with aggregate functions such as SUM, MIN, MAX, COUNT…
2) Group By clause must contain **having** keyword

9) Delete
- Delete from [table name] where [column1 = value1,….]

10) Join
Note: Joins must have **ON** clause

- **Inner Join -> returns matching rows from 2 tables**

  Select [table1 column1 , table1 common column,… table2 common column…]
  from [table1 name]
  Inner Join [table2 name] ON [table1 name]. [table1 common column] =
  [table2].[table2 common column]

- **Left Join (or) Left Outer Join = return all matching rows in 2 tables + non matching rows from left table**

  Use **Left Join** keyword in upper query

- **Right Join (or) Right Outer Join = return all matching rows in 2 tables + non matching rows from right table**

  Use **Right Join** keyword in upper query

- **Full Join (or) Full Outer Join = return all matching rows in 2 tables + non matching rows from left table**

  Use **Full Join** keyword in upper query

- **Cross Join = table1 x table2**

NOTE: Cross join shouldn't have **ON** clause
Use **Cross Join** keyword in above query

- **Advanced joins or intelligent Joins (we use where clause with ON clause)**

Type 1) Getting only Non Matching rows from left table

Select [table1 column1 , table1 common column,… table2 common column…]
from [table1 name]
Left Join [table2 name] **ON** [table1 name]. [table1 common column] = [table2].[table2 common column]
Where [table1 name | table2 name] . [common column] **IS Null**

Type 2) Getting only Non Matching rows from right table

Select [table1 column1 , table1 common column,… table2 common column…]
from [table1 name]
Right Join [table2 name] **ON** [table1 name]. [table1 common column] = [table2].[table2 common column]
Where [table1 name | table2 name] . [common column] **IS Null**

Type 3) Getting only Non Matching rows from both left and right table

Select [table1 column1 , table1 common column,… table2 common column…]
from [table1 name]
Full Join [table2 name] **ON** [table1 name]. [table1 common column] = [table2].[table2 common column]
Where [table1 name | table2 name] . [common column] **IS Null**

Note: In where clause to compare a **null** value in a condition we cannot use **=** operator instead we need to use **IS** operator

- **Self Join = joining a table by itself (if same table has both primary key and foreign key) I.e: Foreign key is linked to its own primary key**

Select[Alias1.column1] **AS** [specific name1] , [Alias2.column1] **AS** [specific name2]
FROM [table1 name] [Alias1]
[**Inner JOIN | Left Join | Right Join | Full Join | Cross Join**] [table1 name] [Alias2]
ON [Alias1].[common column1] = [alias2].[common column2]

NOTE: Cross Join will not have On clause.

Example: Select E.Name as Employee, M.Name as Manager
From Employee as E
Left Join Employee as M

```
ON E.ManagerID = M.EmployeeID
```

11) Union and Union All -> used to combine two select statements

```
Select [column1 , column 2…] from [table1 name]
[UNION | UNION ALL]
Select [column1, column2,….] from [table2 name]
```

Difference : Union removes duplicate records where as Union all doesn't.

12) Stored Procedure -> methods in sql

- **Creating a stored prcedure**

```
Create Procedure [stored procedure name]
AS
BEGIN
        Select * from [table1]
END

(OR)

Create Proc [stored procedure name]
AS
BEGIN
        Select * from [table1]
END
```

- **Calling a stored procedure (or) Executing a stored procedure**

  I) [Stored procedure name]
  II) EXEC [Stored procedure name]
  III) EXECute [Stored procedure name]

- **Creating a stored procedure with input Parameters**

  Note: For parameters decleration we use @ before its name

```
Create Proc spGetEmployeeByGenderAndDepartmentID
@Gender nvarChar(20)
@DepartmentID Int
AS
Begin
        Select name, gender, departmentID from tblEmployee
        where Gender = @gender AND DepartmentID = @DepartmentID
END
```

- **Calling a stored procedure with input parameters**

```
I) spGetEmployeeByGenderAndDepartmentID 'male', '1'
                         (OR)
   spGetEmployeeByGenderAndDepartmentID  @Gender ='male',
   @DepartmentID = '1'
```

II) Exec spGetEmployeeByGenderAndDepartmentID ‘male’, ‘1’
III) Execute spGetEmployeeByGenderAndDepartmentID ‘male’, ‘1’

- **Viewing a defined stored procedure**
  sp_helptext [stored procedure name]

- **Changing / updating a definition of already created Stored procedure**
  Alter Proc [stored procedure name]
  AS
  BEGIN
  Select * from [table1] Order By [column1]
  END

- **Dropping a Stored Procedure**
  Drop Proc [stored procedure Name]

- **Encrypting an already Created Stored Procedure -> locking a procedure so that no one can view its text**
  Alter Proc [stored procedure name]
  WITH Encryption
  AS
  BEGIN
  Select * from [table1] Order By [column1]
  END

- **Creating a stored procedure with Output Parameters**
  Note: For output parmeter decleration we use Out or Output Keyword.
  Create Proc spGetEmployeeByGenderAndDepartmentID
  @Gender nvarChar(20)
  @DepartmentID Int **Out**
  AS
  Begin
  Select @DepartmentID = Count(*)
  from tblEmployee
  where Gender = @gender
  END

- **Calling a stored procedure with input and output parameters**
  Declare @DepartmentID Int
  Exec spGetEmployeeByGenderAndDepartmentID ‘Male’, @DepartmentID out
  Print @DepartmentID

  (OR)

  Declare @DepartmentID Int

```
Exec spGetEmployeeByGenderAndDepartmentID 'Male',
@DepartmentCount = @DepartmentID out
Print @DepartmentID
```

13) **Cast and Convert functions**

- **Cast**
  Cast ([expression] as [Datatype])
- **Convert**
  Convert ([data type], [expression], [style])

14) Scalar User Defined functions -> return a single value of any datatype

- **Create**

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

- **Invoking**
  Select dbo.[function name]

Note: dbo -> database owner

- **Altering a function**
  Alter Function [function name]

- **Dropping a function**
  Drop function [function name]

15) Inline table valued functions -> returns a table

- **Create**

```
CREATE FUNCTION fn_EmployeesByGender(@Gender nvarchar(10))
RETURNS TABLE
AS
RETURN (Select Id, Name, DateOfBirth, Gender, DepartmentId
from tblEmployees
where Gender = @Gender)
```

NOTE: It doesn't have Begin and end part as a scalar function

- **Calling**
  Select * from dbo.fn_EmployeesByGender('Male')

16) Multi statement table valued functions -> returns a table

- **Create**

```
Create Function fn_MSTVF_GetEmployees()
@Table Table (Id int, Name nvarchar(20), DOB Date)
```

```
as
Begin
Insert into @Table
Select Id, Name, Cast(DateOfBirth as Date)
From tblEmployees

Return
End
```

- **Calling**
  ```
  Select * from dbo.fn_MSTVF_GetEmployees()
  ```

17) Temporary Tables
NOTE: Temporary tables are prefixed with # before the table name.

- **I) Local Temporary tables**

  ```
  Create Table #PersonDetails(Id int, Name nvarchar(20))

  Select * from #PersonDetails

  Insert into #PersonDetails Values(1, 'Mike')

  Drop table #PersonDetails
  ```

- **II) Gloabal Temporary tables** are prefixed with two ## before the table name

18) Indexes -> used to find data from a column ON a table or view

- **Creating a Index**

  ```
  Create Index [index name] on [table name] ([column [ASC | DESC])
  ```

- **Viewing the content of a Index on a table**

  ```
  sp_Helpindex [table name]
  ```

- **Dropping an Index on a table**
  ```
  Drop Index [table name] . [index name]
  ```

- **Types of Indexes**

  1) Clustered Index
  - It sorts data created on a column automatically.
  - Created automatically on a primary key column on a table.
  - One table can have only one clustered index.

  - creating a clustered index on two or more columns -> **Composite Clustered Index**

  ```
  Create Clustered Index IX_tblEmployee_Gender_Salary
  ON tblEmployee(Gender DESC, Salary ASC)
  ```

  2) Non-clustered Index

- Creating a non-clusterd index
```
Create NonClustered Index IX_tblEmployee_Name
ON tblEmployee(Name)
```
- It creates an index on a separate table than the table that contains data.
- A table can have as many non clustered indexes.

3) Unique Index

- used to enforce uniqueness of key values in the index.
- Unique Index is created by default on a primary key column of a table.
- We use **Unique** keyword to create a unique index

19) Views -> Saved query / virtual table

- **Creating a View**
```
Create View vWEmployeesByDepartment
as
Select Id, Name, Salary, Gender, DeptName
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
```
- **Selecting a data from View**
```
SELECT * from vWEmployeesByDepartment
```
- **To view the content of a View**
```
Sp_helptext [view name]
```
- **Alter**
```
Alter View [view name]
```
- **Drop**
```
Drop View [view name]
```
- **Update**
```
Update [view name] Set [ColumnName] = [value1] where condition1 = value1
```
- **Delete**
```
Delete From [view name] where condition1 = value1
```
- **Insert**
```
Insert into [view name] values (value1,value2,…)
```

20) Triggers -> Special type of Stored procedure that runs autonatically when an event occurs

- Triggers are of 3 types

1) DML Triggers

2) DDL Triggers
3) LogOn Triggers

- **DML Triggers -> Fired automatically when DML event (Insert, Update, Delete) occurs**
  DML triggers are of 2 types

  1) **After Triggers or For Triggers** -> Gets fired after performing Trigger actions.
  2) Instead Of Triggers -> Gets fired instead of triggering action.

- **After Triggers**

  I) Create a Trigger OR After Insert Trigger

```
CREATE TRIGGER tr_tblEMployee_ForInsert
ON tblEmployee
FOR INSERT
AS
BEGIN
        Declare @Id int
        Select @Id = Id from inserted
        insert into tblEmployeeAudit
        values('New employee with Id  = ' + Cast(@Id as nvarchar(5))
        + ' is added at ' + cast(Getdate() as nvarchar(20)))
END
```

NOTE: Inserted and Deleted table is a special type of table that contains the lastly entered/ deleted value in an existing table and it is only available only in a context of a trigger.

Ii) **ALTER**

Alter trigger [trigger name]

Iii) **After Delete Trigger**

```
CREATE TRIGGER tr_tblEMployee_ForDelete
ON tblEmployee
FOR DELETE
AS
BEGIN
        Declare @Id int
        Select @Id = Id from deleted

        insert into tblEmployeeAudit
        values('An existing employee with Id  = ' + Cast(@Id as
        nvarchar(5)) + ' is deleted at ' + Cast(Getdate() as
        nvarchar(20)))
END
```

iv) **After Update Trigger**

Note: After Update Trigger makes use of both **Inserted and Deleted** tables. Inserted table contains Updated date and Deleted table contains the old data.

```
Create trigger tr_tblEmployee_ForUpdate
on tblEmployee
for Update
as
Begin

-- Declare variables to hold old and updated data
Declare @Id int
Declare @OldName nvarchar(20), @NewName nvarchar(20)
Declare @OldSalary int, @NewSalary int
Declare @OldGender nvarchar(20), @NewGender nvarchar(20)
Declare @OldDeptId int, @NewDeptId int

 -- Variable to build the audit string
Declare @AuditString nvarchar(1000)

-- Load the updated records into temporary table
Select *
into #TempTable
from inserted

 -- Loop thru the records in temp table
While(Exists(Select Id from #TempTable))
Begin

--Initialize the audit string to empty string
Set @AuditString = ''

 -- Select first row data from temp table
Select Top 1 @Id = Id, @NewName = Name,
@NewGender = Gender, @NewSalary = Salary,
@NewDeptId = DepartmentId
from #TempTable

 -- Select the corresponding row from deleted table
Select @OldName = Name, @OldGender = Gender,
@OldSalary = Salary, @OldDeptId = DepartmentId
from deleted where Id = @Id

-- Build the audit string dynamically
Set @AuditString = 'Employee with Id = ' + Cast(@Id as nvarchar(4))
+ ' changed'
if(@OldName <> @NewName)
Set @AuditString = @AuditString + ' NAME from ' + @OldName + ' to
' + @NewName

if(@OldGender <> @NewGender)
Set @AuditString = @AuditString + ' GENDER from ' + @OldGender
+ ' to ' + @NewGender

if(@OldSalary <> @NewSalary)
Set @AuditString = @AuditString + ' SALARY from
' + Cast(@OldSalary as nvarchar(10))+ ' to ' + Cast(@NewSalary as
nvarchar(10))
```

```
            if(@OldDeptId <> @NewDeptId)
            Set @AuditString = @AuditString + ' DepartmentId from
            ' + Cast(@OldDeptId as nvarchar(10))+ ' to ' + Cast(@NewDeptId as
            nvarchar(10))

            insert into tblEmployeeAudit values(@AuditString)

            -- Delete the row from temp table, so we can move to the next row
             Delete from #TempTable where Id = @Id
            End
        End
```

- **Instead of Trigger -> mainly used when inserting a record into a View**

    I) Instead of Insert trigger

    ```
    Create Trigger [trigger name]
    on [table name / view name]
    Instead of insert
    AS
    Begin
            --- body
    End
    ```

    Ii) Instead of Update trigger

    ```
    Create Trigger [trigger name]
    on [table name / view name]
    Instead of Update
    AS
    Begin
            --- body
    End
    ```

    Iii) Instead of Delete Trigger

    ```
    Create Trigger [trigger name]
    On [table name / view name]
    Instead of delete
    AS
    Begin
            --- body
    End
    ```

Trigger	**INSERTED or DELETED?**
Instead of Insert	DELETED table is always empty and the INSERTED table contains the newly inserted data.
Instead of Delete	INSERTED table is always empty and the DELETED table contains the rows deleted
Instead of Update	DELETED table contains OLD data (before update), and inserted table contains NEW data(Updated dat

21) Table Variable

- Just like TempTables, a table variable is also created in TempDB.
- The scope of a table variable is the batch, stored procedure, or statement block in which it is declared.
- They can be passed as parameters between procedures.

- **Defining a table variable**

```
Declare @tblEmployeeCount table
(DeptName nvarchar(20),DepartmentId int, TotalEmployees int)

Insert @tblEmployeeCount
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId

Select DeptName, TotalEmployees
From @tblEmployeeCount
where  TotalEmployees >= 2
```

22) Derived tables
Using a table definition to create another table is called derived table.

```
Select DeptName, TotalEmployees
from
(
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId
)
as EmployeeCount
where TotalEmployees >= 2
```

- Here EmployeeCount is a derived table.

NOTE: Derived tables are available only in the context of the current query.

23) CTE (Common Table Expression)

- **Common table expression (CTE)** is a temporary result set, that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement, that immediately follows the **CTE.**

- **Creating a CTE**

Syntax:
A)Single CTE

```
        with [CTE name] (column1 , column2,…)
          As
                (
            --- CTE Query
        )
```

B) Multiple CTE

```
        with [CTE name1] (column1 , column2,…)
         As
        (
            --- CTE Query1
        ) ,
        [CTE name2] (column1 , column2,…)
        As
        (
            --- CTE Query2
        )
```

Example:

```
With EmployeeCount(DeptName, DepartmentId, TotalEmployees)
as
(
Select DeptName, DepartmentId, COUNT(*) as TotalEmployees
from tblEmployee
join tblDepartment
on tblEmployee.DepartmentId = tblDepartment.DeptId
group by DeptName, DepartmentId
)

Select DeptName, TotalEmployees
from EmployeeCount
where TotalEmployees >= 2
```

Note: A CTE is similar to a derived table in that it is not stored as an object and lasts only for the duration of the query.

- **Update-able CTE**

```
With Employees_Name_Gender
as
(
Select Id, Name, Gender from tblEmployee
)
Update Employees_Name_Gender Set Gender = 'Female' where Id = 1
```

NOTE:
1. A CTE is based on a single base table, then the UPDATE succeeds and works as expected.

2. A CTE is based on more than one base table, and if the UPDATE affects multiple base tables, the update is not allowed and the statement terminates with an error.

3. A CTE is based on more than one base table, and if the UPDATE affects only one base table, the UPDATE succeeds(but not as expected always)

24) Database Normalization -> breaking a table into many to normalize the data.

- **Database normalization** is the process of organizing data to minimize data redundancy (data duplication), which in turn ensures data consistency.

- **1- Normal Form:**

A table is said to be in 1NF, if
1. The data in each column should be **atomic**. No multiple values, separated by comma.
2. The table does not contain any **repeating column groups**
3. Identify each record **uniquely using primary key**.

- **2- Normal Form**

A table is said to be in 2NF, if
1. The table meets all the **conditions of 1NF**
2. Move **redundant** data to a separate table
3. Create **relationship** between these tables using foreign keys.

- **3 - Normal Form**

A table is said to be in 3NF, if the table
1. Meets all the conditions of **1NF and 2NF**
2. Does not contain columns (attributes) that are not fully **dependent upon the primary key**

25) Pivot Operator

- **Pivot operator** can be used to turn **unique values from one column**, into **multiple columns** in the output, there by effectively **rotating a table**.

Synatx:
```
SELECT <non-pivoted column>,
   [first pivoted column] AS <column name>,
   [second pivoted column] AS <column name>,
   ...
   [last pivoted column] AS <column name>
FROM
   (<SELECT query that produces the data>)
   AS <alias for the source query>
PIVOT
(
   <aggregation function>(<column being aggregated>)
FOR
   [<column that contains the values that will become column headers>]
   IN ( [first pivoted column], [second pivoted column], ... [last pivoted column])
)
AS <alias for the pivot table>
<optional ORDER BY clause>;
```

Example:

```
Select SalesAgent, India, US, UK
from
```

```
(
  Select SalesAgent, SalesCountry, SalesAmount from tblProductsSale
) as SourceTable
Pivot
(
 Sum(SalesAmount) for SalesCountry in (India, US, UK)
) as PivotTable
```

26) Error Handling in SQL

- Error handling in sql server 2000 -> **@@Error**
- Error handling in sql server 2005 & later -> **try and catch**

NOTE: Sometimes, system functions that begin with two at signs **(@@)**, are called as **global variables**. They are not variables and do not have the same behaviours as variables, instead they are very similar to functions.

- To return an error use **RaiseError()**
  RAISERROR('Error Message', ErrorSeverity, ErrorState)

NOTE:
a) **Severity and State** are integers. In most cases, when you are returning custom errors, the severity level is 16, which indicates general errors that can be corrected by the user.

b) **ErrorState** is also an integer between 1 and 255. RAISERROR only generates errors with state from 1 through 127.

NOTE:
A) @@ERROR is cleared and reset on each statement execution. Check it immediately following the statement being verified, or save it to a local variable that can be checked later.

B) @@Error return non zero value if there is a error else returns **0.**

- **Error Handling in SQL server 2005 and beyond**

Syntax:

```
BEGIN TRY
   { Any set of SQL statements }
       Commit Transaction
END TRY
BEGIN CATCH
   [ Optional: Any set of SQL statements ]
       Print Error OR
- Rollback Transaction
END CATCH
[Optional: Any other SQL Statements]
```

System Inbuilt functions	Description
@@Error	@@ERROR return the error number for last executed T-SQL statements. It returns 0 if the previous Transact-SQL

	statement encountered no errors else return an error number.
Error_Number()	ERROR_NUMBER() returns the error number that caused the error. It returns zero if called outside the catch block.
Error_State()	ERROR_STATE returns the state number of the error. The return type of ERROR_STATE is **INT**.
Error_Line()	ERROR_LINE returns the line number at which an error occurred. The return type of ERROR_LINE is **INT**.
Error_Message()	ERROR_MESSAGE returns the message text of the error that caused the error. The return type of ERROR_MESSAGE is **nvarchar(4000)**.
Error_Procedure()	ERROR_PROCEDURE returns the name of the Stored Procedure or trigger of where an error occurred. The return type of ERROR_PROCEDURE is **nvarchar(128)**.
Error_Severity()	ERROR_SEVERITY returns the severity of the error. The return type of ERROR_SEVERITY is **INT**.
RaiseError()	RAISEERROR is used to generate an error message and initiates error processing for the session.
GoTo()	GOTO causes a jump to a specific step or statement. It alters the flow of execution to a label. We declare some labels in batch and alter we can move at a specific label. GOTO can exist within a conditional control-of-flow statements, statement blocks, or procedures, but it cannot go to a label outside the batch. GOTO cannot be used to jump into a TRY or CATCH scope.
@@TranCount	**@@TRANCOUNT** returns the count of open transactions in the current session. It increments the count value whenever we open a transaction and decrements the count whenever we commit the transaction.

27) Transactions in SQL

- **Transaction is a group of commands** that change the data stored in a database.
- A transaction, is treated as a single unit.

- A transaction ensures that, either all of the commands succeed, or none of them. If one of the commands in the transaction fails, all of the commands fail, and any data that was modified in the database is rolled back. In this way, transactions maintain the integrity of data in a database.
- **Transaction processing follows these steps:**

```
1. Begin a transaction.
2. Process database commands.
3. Check for errors.
  If errors occurred,
     rollback the transaction,
  else,
     commit the transaction
```

- **Example:**

```
Create Procedure spUpdateAddress
as
Begin
Begin Try
Begin Transaction
Update tblMailingAddress set City = 'LONDON'
where AddressId = 1 and EmployeeNumber = 101

Update tblPhysicalAddress set City = 'LONDON'
where AddressId = 1 and EmployeeNumber = 101
Commit Transaction
End Try
Begin Catch
Rollback Transaction
End Catch
End
```

- Common transaction statements used in exception handling process,

A) **Begin transaction** ->

If you were to add BEGIN TRANSACTION (or BEGIN TRAN) before the statement it automatically makes the transaction explicit and holds a lock on the table until the transaction is either committed or rolled back.

BEGIN TRANSACTION marks the starting point of an explicit, local transaction.

*B) **Rollback transaction** ->*

ROLLBACK TRANSACTION rolls back an explicit or implicit transaction to the beginning of the transaction, or to a save point inside the transaction. It also frees resources held by the transaction.

*C) **Commit Transaction** ->*

COMMIT TRANSACTION marks the end of a successful implicit or explicit transaction. If @@TRANCOUNT is 1, COMMIT TRANSACTION makes all data modifications performed since the start of the transaction a permanent *part of the database, frees the resources held by the transaction, and decrements @@TRANCOUNT to 0. If @@TRANCOUNT is greater than 1, COMMIT TRANSACTION decrements @@TRANCOUNT only by 1 and the transaction stays active.*

*D) **End Transaction** -> mentions the end of the transaction.*

28) Transaction ACID Test

- A successful transaction must pass the "ACID" test, that is, it must be
  A - Atomic
  C - Consistent
  I - Isolated
  D - Durable.
- **Atomic** - All statements in the transaction either completed successfully or they were all rolled back. The task that the set of operations represents is either accomplished or not, but in any case not left half-done.
- **Consistent** - All data touched by the transaction is left in a **logically consistent state**.
- **Isolated** - The transaction must affect data without interfering with other concurrent transactions, or being interfered with by them. This prevents transactions from making changes to data based on uncommitted information, for example changes to a record that are subsequently rolled back. **Most databases use locking to maintain transaction isolation**.
- **Durable** - Once a change is made, it is permanent. If a system error or power failure occurs before a set of commands is complete, those commands are undone and the data is restored to its original state once the system begins running again.

29) Cursors

- **A Cursor** is a pointer for a row in a table.
- **However, if there is ever a need to process the rows, on a row-by-row basis**, then cursors are your choice.
- Cursors are very bad for performance, and should be avoided always. Most of the time, cursors can be very easily replaced using joins.
- Cursor types

1. Forward-Only
2. Static
3. Keyset
4. Dynamic

Example:

```
Declare @ProductId int

-- Declare the cursor using the declare keyword
Declare ProductIdCursor CURSOR FOR
Select ProductId from tblProductSales

-- Open statement, executes the SELECT statment
-- and populates the result set
```

```
Open ProductIdCursor

-- Fetch the row from the result set into the variable
Fetch Next from ProductIdCursor into @ProductId

-- If the result set still has rows, @@FETCH_STATUS will be ZERO
While(@@FETCH_STATUS = 0)
Begin
Declare @ProductName nvarchar(50)
Select @ProductName = Name from tblProducts where Id = @ProductId

if(@ProductName = 'Product - 55')
Begin
Update tblProductSales set UnitPrice = 55 where ProductId = @ProductId
End
else if(@ProductName = 'Product - 65')
Begin
Update tblProductSales set UnitPrice = 65 where ProductId = @ProductId
End
else if(@ProductName like 'Product - 100%')
Begin
Update tblProductSales set UnitPrice = 1000 where ProductId = @ProductId
End

Fetch Next from ProductIdCursor into @ProductId
End

-- Release the row set
CLOSE ProductIdCursor
-- Deallocate, the resources associated with the cursor
DEALLOCATE ProductIdCursor
```

30) Merge

- Merge statement allows us to perform Inserts, Updates and Deletes in one statement.
- **With merge statement we require 2 tables**
  1. Source Table - Contains the changes that needs to be applied to the target table
  2. Target Table - The table that require changes (Inserts, Updates and Deletes)
- **Merge statement syntax:**

```
MERGE [TARGET table name] AS T
USING [SOURCE table name] AS S
  ON [JOIN_CONDITIONS]
 WHEN MATCHED THEN
     [UPDATE STATEMENT]
 WHEN NOT MATCHED BY TARGET THEN
     [INSERT STATEMENT]
 WHEN NOT MATCHED BY SOURCE THEN
     [DELETE STATEMENT]
```

Example:

```
MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, NAME) VALUES(S.ID, S.NAME)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

31) Concurrent Transaction

- When two transactions runs on a same table then those transactions are called concurrent transactions.
- Concurrent transaction Problem,

1) Dirty read

A dirty read happens when one transaction is permitted to read data that has been modified by another transaction that has not yet been committed. In most cases this would not cause a problem. However, if the first transaction is rolled back after the second reads the data, the second transaction has dirty data that does not exist anymore

We can read dirty data by setting transaction 2 isolation level as Read uncomitted or using No lock

```
Set Transaction Isolation Level Read Uncommitted


Select * from tblInventory (NOLOCK) where Id=1
```

2) Lost Update

Lost update problem happens when 2 transactions read and update the same data.

Check below table to know which isolation level has problem.

3) Non repeatable reads

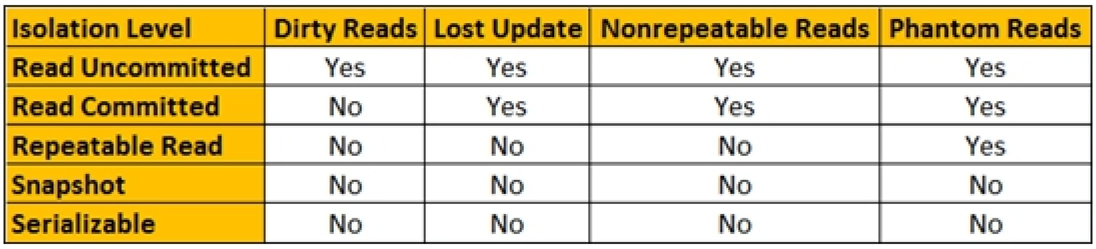
Non repeatable read problem happens when one transaction reads the same data twice and another transaction updates that data in between the first and second read of transaction one.

4) Phantom reads.

Phantom read happens when one transaction executes a query twice and it gets a different number of rows in the result set each time. This happens when a second transaction inserts a new row that matches the WHERE clause of the query executed by the first transaction.

NOTE: concurrent transaction problems can be resolved by setting the isolation levels on a transaction

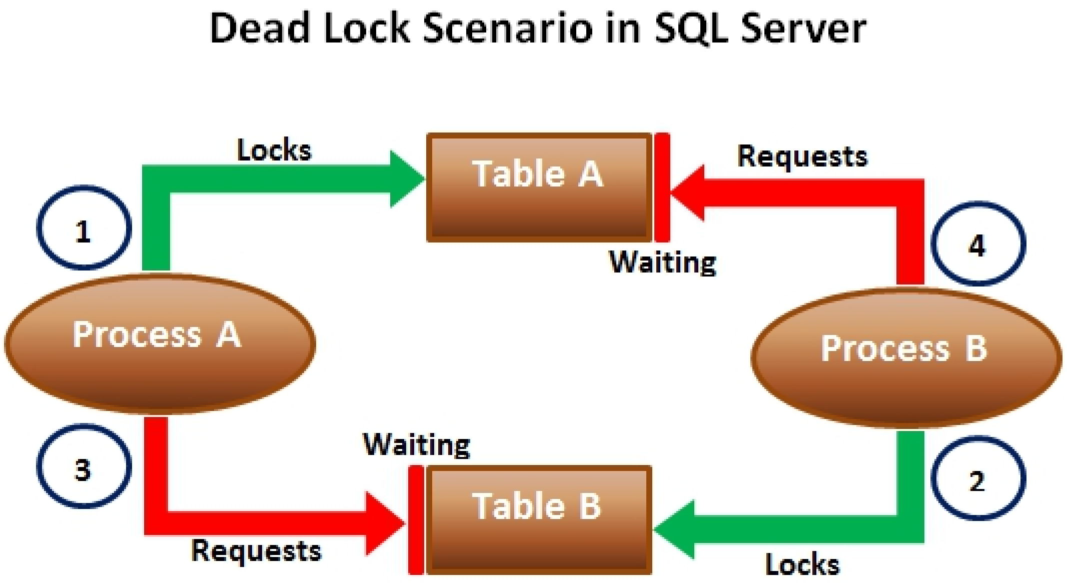
Set Transacton isolation level



Isolation Level	Dirty Reads	Lost Update	Nonrepeatable Reads	Phantom Reads
Read Uncommitted	Yes	Yes	Yes	Yes
Read Committed	No	Yes	Yes	Yes
Repeatable Read	No	No	No	Yes
Snapshot	No	No	No	No
Serializable	No	No	No	No

32) Deadlocks in SQL server

- In a database, a deadlock occurs when two or more processes have a resource locked, and each process requests a lock on the resource that another process has already locked. Neither of the transactions here can move forward, as each one is waiting for the other to release the lock. The following diagram explains this.



- When deadlocks occur, SQL Server will choose one of processes as the deadlock victim and rollback that process, so the other process can move forward.
- **How SQL Server detects deadlocks**
  Lock monitor thread in SQL Server, runs every 5 seconds by default to detect if there are any deadlocks. If the lock monitor thread finds deadlocks, the deadlock detection interval will drop from 5 seconds to as low as 100 milliseconds depending on the frequency of deadlocks. If the lock monitor thread stops finding deadlocks, the Database Engine increases the intervals between searches to 5 seconds.

- **What happens when a deadlock is detected**
  When a deadlock is detected, the Database Engine ends the deadlock by choosing one of the threads as the deadlock victim. The deadlock victim's transaction is then rolled back and returns a 1205 error to the application. Rolling back the transaction of the deadlock victim releases all locks held by that transaction. This allows the other transactions to become unblocked and move forward.
- **What is DEADLOCK_PRIORITY**
  By default, SQL Server chooses a transaction as the deadlock victim that is least expensive to roll back. However, a user can specify the priority of sessions in a deadlock situation using the SET DEADLOCK_PRIORITY statement. The session with the lowest deadlock priority is chosen as the deadlock victim.

  Example : SET DEADLOCK_PRIORITY NORMAL

  DEADLOCK_PRIORITY
  1. The default is Normal
  2. Can be set to LOW, NORMAL, or HIGH
  3. Can also be set to a integer value in the range of -10 to 10.
  LOW : -5
  NORMAL : 0
  HIGH : 5
- **What is the deadlock victim selection criteria**
  1. If the DEADLOCK_PRIORITY is different, the session with the lowest priority is selected as the victim
  2. If both the sessions have the same priority, the transaction that is least expensive to rollback is selected as the victim
  3. If both the sessions have the same deadlock priority and the same cost, a victim is chosen randomly
- **Enable Trace flag :** To enable trace flags use DBCC command. -1 parameter indicates that the trace flag must be set at the global level. If you omit -1 parameter the trace flag will be set only at the session level.

  ```
  DBCC Traceon(1222, -1)
  ```

  To check the status of the trace flag
  ```
  DBCC TraceStatus(1222, -1)
  ```

  To turn off the trace flag
  ```
  DBCC Traceoff(1222, -1)
  ```
- **To read the error log**
  ```
  execute sp_readerrorlog
  ```

The deadlock information in the error log has three sections

Section	Description
Deadlock Victim	Contains the ID of the process that was selected as the deadlock victim and killed by SQL Server.
Process List	Contains the list of the processes that participated in the deadlock.
Resource List	Contains the list of the resources (database objects) owned by the processes involved in the deadlock

- **Process List :** The process list has lot of items. Here are some of them that are particularly useful in understanding what caused the deadlock.

Node	Description

loginname	The loginname associated with the process
isolationlevel	What isolation level is used
procname	The stored procedure name
Inputbuf	The code the process is executing when the deadlock occured

- **Resource List :** Some of the items in the resource list that are particularly useful in understanding what caused the deadlock.

Node	Description
objectname	Fully qualified name of the resource involved in the deadlock
owner-list	Contains (owner id) the id of the owning process and the lock mode it has acquired on the resource. lock mode determines how the resource can be accessed by concurrent transactions. S for Shared lock, U for Update lock, X for Exclusive lock etc
waiter-list	Contains (waiter id) the id of the process that wants to acquire a lock on the resource and the lock mode it is requesting

- To prevent the deadlock that we have in our case, we need to ensure that database objects (Table A & Table B) are accessed in the same order every time.

33) Blocking Transaction / queries

- Blocking occurs if there are open transactions that hasn't committed.
- We can identify only last open transaction from below query

```
DBCC OpenTran
```

- Killing the process using SQL command :

```
KILL [Process_ID]
```

- What happens when you kill a session?

All the work that the transaction has done will be rolled back. The database must be put back in the state it was in, before the transaction started.

34) Except Operator

- **EXCEPT operator** returns unique rows from the left query that aren't in the right query's results.

Note:

- The number and the order of the columns must be the same in all queries
- The data types must be same or compatible.

Example:

```
Select Id, Name, Gender
From TableB
Except
Select Id, Name, Gender
From TableA
```

35) Difference between Except and NOT IN

1. Except filters duplicates and returns only DISTINCT rows from the left query that aren't in the right query's results, where as NOT IN does not filter the duplicates.
2. EXCEPT operator expects the same number of columns in both the queries, where as NOT IN, compares a single column from the outer query with a single column from the sub query.

36) Intersect operator:

- Intersect operator retrieves the common records from both the left and the right query of the Intersect operator.

NOTE:

- The number and the order of the columns must be same in both the queries.
- The data types must be same or at least compatible.

37) Difference between intersect and inner join

1. INTERSECT filters duplicates and returns only DISTINCT rows that are common between the LEFT and Right Query, where as INNER JOIN does not filter the duplicates.
2. INNER JOIN treats two NULLS as two different values**.** INTERSECT treats two Nulls as a same value and it returns all matching rows.

38) Cross Apply and Outer Apply

- The APPLY operator is used to join a table to a table-valued function.
- The Table Valued Function on the right hand side of the APPLY operator gets called for each row from the left (also called outer table) table.
- Cross Apply returns only matching rows (semantically equivalent to Inner Join)
- Outer Apply returns matching + non-matching rows (semantically equivalent to Left Outer Join). The unmatched columns of the table valued function will be set to NULL.
- Example:

```
Select D.DepartmentName, E.Name, E.Gender, E.Salary
from Department D
[Outer Apply | Cross Apply] fn_GetEmployeesByDepartmentId(D.Id) E
```

39) DDL Triggers

- **DDL triggers fire in response to DDL events** - CREATE, ALTER, and DROP (Table, Function, Index, Stored Procedure etc...).
- **What is the use of DDL triggers**
  - -If you want to execute some code in response to a specific DDL event
  - -To prevent certain changes to your database schema
  - -Audit the changes that the users are making to the database structure

Syntax for creating DDL trigger

```
CREATE TRIGGER [Trigger_Name]
ON [Scope (Server|Database)]
```

```
FOR [EventType1, EventType2, EventType3, ...],
AS
BEGIN
  -- Trigger Body
END
```

- **DDL triggers scope :** DDL triggers can be created in a specific database or at the server level.
- **Example:**

```
CREATE TRIGGER trMyFirstTrigger
ON Database
FOR CREATE_TABLE
AS
BEGIN
  Print 'New table created'
END
```

- **To disable trigger**
  1. Right click on the trigger in object explorer and select **"Disable"** from the context menu
  2. You can also disable the trigger using the following T-SQL command

```
DISABLE TRIGGER trMyFirstTrigger ON DATABASE
```

- **To enable trigger**
  1. Right click on the trigger in object explorer and select "Enable" from the context menu
  2. You can also enable the trigger using the following T-SQL command
  `ENABLE TRIGGER trMyFirstTrigger ON DATABASE`
- **To drop trigger**
  1. Right click on the trigger in object explorer and select "Delete" from the context menu
  2. You can also drop the trigger using the following T-SQL command
  `DROP TRIGGER trMyFirstTrigger ON DATABASE`

40) Logon Trigger

- **Logon triggers fire in response to a LOGON event**. Logon triggers fire after the authentication phase of logging in finishes, but before the user session is actually established.
- **Logon triggers can be used for**
  1. Tracking login activity
  2. Restricting logins to SQL Server
  3. Limiting the number of sessions for a specific login
- **Example**:

```
CREATE TRIGGER tr_LogonAuditTriggers
ON ALL SERVER
FOR LOGON
AS
BEGIN
   DECLARE @LoginName NVARCHAR(100)

   Set @LoginName = ORIGINAL_LOGIN()

   IF (SELECT COUNT(*) FROM sys.dm_exec_sessions
       WHERE is_user_process = 1
       AND original_login_name = @LoginName) > 3
   BEGIN
```

```
        Print 'Fourth connection of ' + @LoginName + ' blocked'
        ROLLBACK
    END
END
```

41) Select INTO

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

SELECT INTO statement in SQL Server can do the following

1. Copy all rows and columns from an existing table into a new table. This is extremely useful when you want to make a backup copy of the existing table.

```
SELECT * INTO EmployeesBackup FROM Employees
```

2. Copy all rows and columns from an existing table into a new table in an external database.

```
SELECT * INTO HRDB.dbo.EmployeesBackup FROM Employees
```

3. Copy only selected columns into a new table

```
SELECT Id, Name, Gender INTO EmployeesBackup FROM Employees
```

4. Copy only selected rows into a new table

```
SELECT * INTO EmployeesBackup FROM Employees WHERE DeptId = 1
```

5. Copy columns from 2 or more table into a new table

```
SELECT * INTO EmployeesBackup
FROM Employees
INNER JOIN Departments
ON Employees.DeptId = Departments.DepartmentId
```

6. Create a new table whose columns and datatypes match with an existing table.

```
SELECT * INTO EmployeesBackup FROM Employees WHERE 1 <> 1
```

7. Copy all rows and columns from an existing table into a new table on a different SQL Server instance. For this, create a linked server and use the 4 part naming convention

```
SELECT * INTO TargetTable
FROM [SourceServer].[SourceDB].[dbo].[SourceTable]
```

Please note : You cannot use SELECT INTO statement to select data into an existing table. For this you will have to use INSERT INTO statement.

```
INSERT INTO ExistingTable (ColumnList)
SELECT ColumnList FROM SourceTable
```

42) Difference between Where and Having

- WHERE clause cannot be used with aggregates where as HAVING can. This means WHERE clause is used for filtering individual rows where as HAVING clause is used to filter groups.
- WHERE comes before GROUP BY. This means WHERE clause filters rows before aggregate calculations are performed. HAVING comes after GROUP BY. This means HAVING clause filters rows after aggregate calculations are

performed. So from a performance standpoint, HAVING is slower than WHERE and should be avoided when possible.

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

43) Grouping Sets

- IF a query includes 2 or more group by clause which are combined tpgether with union all in that place we can use grouping sets to do same process.
- **Example:**

```
Select Country, Gender, Sum(Salary) TotalSalary
From Employees
Group BY
    GROUPING SETS
    (
        (Country, Gender), -- Sum of Salary by Country and Gender
        (Country),         -- Sum of Salary by Country
        (Gender) ,         -- Sum of Salary by Gender
        ()                 -- Grand Total
    )
```

44) Roll Up

- **ROLLUP in SQL Server** is used to do aggregate operation on multiple levels in hierarchy.
- Roll Up is used with group by to get grand total of a particular column.
- **Example:**

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees

GROUP BY ROLLUP(Country)
```

OR

The above query can also be rewritten as shown below

```
SELECT Country, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Country WITH ROLLUP
```

45) CUBE

- Cube() in SQL Server produces the result set by generating all combinations of columns specified in GROUP BY CUBE().
- **Example:**

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Cube(Country, Gender)

--OR
```

```
SELECT Country, Gender, SUM(Salary) AS TotalSalary
FROM Employees
```

46) Difference between cube and rollup in SQL Server

- **CUBE generates a result set** that shows aggregates for all combinations of values in the selected columns, where as ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

NOTE: You won't see any difference when you use ROLLUP and CUBE on a single column. Both the following queries produces the same output.

47) Grouping function

- Grouping(Column) indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set.
- **Example:**

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
         GROUPING(Continent) AS GP_Continent,
         GROUPING(Country) AS GP_Country,
         GROUPING(City) AS GP_City
FROM Sales
GROUP BY ROLLUP(Continent, Country, City)
```

- **What is the use of Grouping function in real world**
  When a column is aggregated in the result set, the column will have a NULL value. If you want to replace NULL with All then this GROUPING function is very handy.

Note : Grouping function can be used with Rollup, Cube and Grouping Sets

48) Grouping_ID() function

- GROUPING_ID function computes the level of grouping.
- **Difference between GROUPING and GROUPING_ID**

  Syntax : GROUPING function is used on single column, where as the column list for GROUPING_ID function must match with GROUP BY column list.

```
GROUPING(Col1)
GROUPING_ID(Col1, Col2, Col3,...)
```

GROUPING indicates whether the column in a GROUP BY list is aggregated or not. Grouping returns 1 for aggregated or 0 for not aggregated in the result set.

GROUPING_ID() function concatenates all the GOUPING() functions, perform the binary to decimal conversion, and returns the equivalent integer. In short
GROUPING_ID(A, B, C) = GROUPING(A) + GROUPING(B) + GROUPING©

- **Example:**

```
SELECT   Continent, Country, City, SUM(SaleAmount) AS TotalSales,
         GROUPING_ID(Continent, Country, City) AS GPID
FROM Sales
```

```
GROUP BY ROLLUP(Continent, Country, City)
ORDER BY GPID
```

49) OVER Clause

- The **OVER** clause combined with **PARTITION BY** is used to break up data into partitions.
  Syntax : function (...) OVER (PARTITION BY col1, Col2, ...)

  The specified function operates for each partition.
- **For example :**
  COUNT(Gender) OVER (PARTITION BY Gender) will partition the data by **GENDER** i.e there will 2 partitions (Male and Female) and then the COUNT() function is applied over each partition.

  Any of the following functions can be used. Please note this is not the complete list.
  COUNT(), AVG(), SUM(), MIN(), MAX(), ROW_NUMBER(), RANK(), DENSE_RANK() etc.
- **Example:**

```
SELECT Name, Salary, Gender,
       COUNT(Gender) OVER(PARTITION BY Gender) AS GenderTotals,
       AVG(Salary) OVER(PARTITION BY Gender) AS AvgSal,
       MIN(Salary) OVER(PARTITION BY Gender) AS MinSal,
       MAX(Salary) OVER(PARTITION BY Gender) AS MaxSal
FROM Employees
```

50) Row_Number () function

- Returns the sequential number of a row starting at 1
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, row number is reset to 1 when the partition changes

Syntax : ROW_NUMBER() OVER (ORDER BY Col1, Col2)

- **Example:**

```
SELECT Name, Gender, Salary,
       ROW_NUMBER() OVER (ORDER BY Gender) AS RowNumber
FROM Employees
```

Please note : If ORDER BY clause is not specified you will get the following error
The function 'ROW_NUMBER' must have an OVER clause with ORDER BY

- **Row_Number function with PARTITION BY :** In this example, data is partitioned by Gender, so ROW_NUMBER will provide a consecutive numbering only for the rows with in a partion. When the partition changes the row number is reset to 1.

```
SELECT Name, Gender, Salary,
       ROW_NUMBER() OVER (PARTITION BY Gender ORDER BY Gender) AS Ro
wNumber
FROM Employees
```

- **Use case for Row_Number function :** Deleting all duplicate rows except one from a sql server table.

51) Rank () and DenseRank () function

- Returns a rank starting at 1 based on the ordering of rows imposed by the ORDER BY clause
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, rank is reset to 1 when the partition changes
- **Difference between Rank and Dense_Rank functions**
  Rank function skips ranking(s) if there is a tie where as Dense_Rank will not.

  For example : If you have 2 rows at rank 1 and you have 5 rows in total.

```
RANK() returns - 1, 1, 3, 4, 5
DENSE_RANK returns - 1, 1, 2, 3, 4
```

Syntax :

```
RANK() OVER (ORDER BY Col1, Col2, ...)
DENSE_RANK() OVER (ORDER BY Col1, Col2, ...)
```

- Example:

```
SELECT Name, Salary, Gender,
RANK() OVER (ORDER BY Salary DESC) AS [Rank],
DENSE_RANK() OVER (ORDER BY Salary DESC) AS DenseRank
FROM Employees
```

52) Difference between rank dense_rank and row_number in SQL

- **Similarities between RANK, DENSE_RANK and ROW_NUMBER functions**
- Returns an increasing integer value starting at 1 based on the ordering of rows imposed by the ORDER BY clause (if there are no ties)
- ORDER BY clause is required
- PARTITION BY clause is optional
- When the data is partitioned, the integer value is reset to 1 when the partition changes
- **Difference between RANK, DENSE_RANK and ROW_NUMBER functions**
- **ROW_NUMBER :** Returns an increasing unique number for each row starting at 1, even if there are duplicates.
- **RANK :** Returns an increasing unique number for each row starting at 1. When there are duplicates, same rank is assigned to all the duplicate rows, but the next row after the duplicate rows will have the rank it would have been assigned if there had been no duplicates. So RANK function skips rankings if there are duplicates.
- **DENSE_RANK :** Returns an increasing unique number for each row starting at 1. When there are duplicates, same rank is assigned to all the duplicate rows but the DENSE_RANK function will not skip any ranks. This means the next row after the duplicate rows will have the next rank in the sequence.
-

53) NTILE() Function

- ORDER BY Clause is required
- PARTITION BY clause is optional
- Distributes the rows into a specified number of groups
- If the number of rows is not divisible by number of groups, you may have groups of two different sizes.
- Larger groups come before smaller groups

For example

- NTILE(2) of 10 rows divides the rows in 2 Groups (5 in each group)
- NTILE(3) of 10 rows divides the rows in 3 Groups (4 in first group, 3 in 2nd & 3rd group)

Syntax : NTILE (Number_of_Groups) OVER (ORDER BY Col1, Col2, ...)

Example:

```
SELECT Name, Gender, Salary,
NTILE(3) OVER (ORDER BY Salary) AS [Ntile]
FROM Employees
```

54) Lead and Lag functions

- Lead function is used to access subsequent row data along with current row data
- Lag function is used to access previous row data along with current row data
- ORDER BY clause is required
- PARTITION BY clause is optional

Syntax

```
LEAD(Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)
LAG(Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)
```

- **Offset -** Number of rows to lead or lag.
- **Default_Value -** The default value to return if the number of rows to lead or lag goes beyond first row or last row in a table or partition. If default value is not specified NULL is returned.

- Example:

```
SELECT Name, Gender, Salary,
       LEAD(Salary, 2, -1) OVER (ORDER BY Salary) AS Lead_2,
       LAG(Salary, 1, -1) OVER (ORDER BY Salary) AS Lag_1
FROM Employees
```

55) FIRST_VALUE function

- Retrieves the first value from the specified column
- ORDER BY clause is required
- PARTITION BY clause is optional

Syntax : `FIRST_VALUE(Column_Name) OVER (ORDER BY Col1, Col2, ...)`

Example:

```
SELECT Name, Gender, Salary,
FIRST_VALUE(Name) OVER (ORDER BY Salary) AS FirstValue
FROM Employees
```

56) LAST_VALUE function

- Retrieves the last value from the specified column
- ORDER BY clause is required
- PARTITION BY clause is optional
- ROWS or RANGE clause is optional, but for it to work correctly you may have to explicitly specify a value

Syntax : LAST_VALUE(Column_Name) OVER (ORDER BY Col1, Col2, ...)

LAST_VALUE function not working as expected : In the following example, LAST_VALUE function does not return the name of the highest paid employee. This is because we have not specified an explicit value for ROWS or RANGE clause. As a result it is using it's default value RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

- Example:

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (ORDER BY Salary) AS LastValue
FROM Employees
```

- **LAST_VALUE function working as expected :** In the following example, LAST_VALUE function returns the name of the highest paid employee as expected. Notice we have set an explicit value for ROWS or RANGE clause to ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

  This tells the LAST_VALUE function that it's window starts at the first row and ends at the last row in the result set.

```
SELECT Name, Gender, Salary,
   LAST_VALUE(Name) OVER (ORDER BY Salary ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS LastValue
FROM Employees
```

57) UNPIVOT

- PIVOT operator turns ROWS into COLUMNS, where as UNPIVOT turns COLUMNS into ROWS.
- Example:

```
SELECT SalesAgent, Country, SalesAmount
FROM tblProductSales
UNPIVOT
(
    SalesAmount
    FOR Country IN (India, US ,UK)
) AS UnpivotExample
```

58) **Choose function**

- Returns the item at the specified index from the list of available values
- The index position starts at 1 and NOT 0 (ZERO)
- It works as a switch statement in c#

Syntax : CHOOSE(index, val_1, val_2, ...)

Example : Returns the item at index position 2

```
SELECT CHOOSE(2, 'India','US', 'UK') AS Country
```

59) IIF function

- Returns one of two the values, depending on whether the Boolean expression evaluates to true or false
- IIF is a shorthand way for writing a CASE expression

Syntax : IIF (boolean_expression, true_value, false_value)

Example : Returns Male as the boolean expression evaluates to TRUE

```
DECLARE @Gender INT
SET @Gender = 1
SELECT IIF( @Gender = 1, 'Male', 'Femlae') AS Gender
```

60) TRY_PARSE function

- Converts a string to Date/Time or Numeric type
- Returns NULL if the provided string cannot be converted to the specified data type
- Requires .NET Framework Common Language Runtime (CLR)

Syntax : TRY_PARSE (string_value AS data_type)

Example : Convert string to INT. As the string can be converted to INT, the result will be 99 as expected.

```
SELECT TRY_PARSE('99' AS INT) AS Result
```

What is the difference between PARSE and TRY_PARSE
PARSE will result in an error if the conversion fails, where as TRY_PARSE will return NULL instead of an error.

61) TRY_CONVERT function

- Converts a value to the specified data type
- Returns NULL if the provided value cannot be converted to the specified data type
- If you request a conversion that is explicitly not permitted, then TRY_CONVERT fails with an error

Syntax : TRY_CONVERT (data_type, value, [style])

Style parameter is optional. The range of acceptable values is determined by the target data_type. For the list of all possible values for style parameter, please visit the following MSDN article
https://msdn.microsoft.com/en-us/library/ms187928.aspx

Example : Convert string to INT. As the string can be converted to INT, the result will be 99 as expected.

```
SELECT TRY_CONVERT(INT, '99') AS Result
```

- **What is the difference between CONVERT and TRY_CONVERT**
  CONVERT will result in an error if the conversion fails, where as TRY_CONVERT will return NULL instead of an error.
- **Difference between TRY_PARSE and TRY_CONVERT functions**
  TRY_PARSE can only be used for converting from string to date/time or number data types where as TRY_CONVERT can be used for any general type conversions.

62) EOMONTH function

- Returns the last day of the month of the specified date

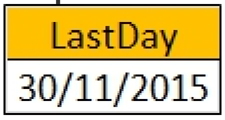
Syntax : EOMONTH (start_date [, month_to_add])

start_date : The date for which to return the last day of the month
month_to_add : Optional. Number of months to add to the start_date. EOMONTH adds the specified number of months to start_date, and then returns the last day of the month for the resulting date.

Example : Returns last day of the month November

```
SELECT EOMONTH('11/20/2015') AS LastDay
```

Output :



LastDay
30/11/2015

The following example adds 2 months to the start_date and returns the last day of the month from the resulting date

```
SELECT EOMONTH('3/20/2016', 2) AS LastDay
```

63) DATEFROMPARTS function

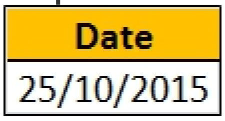
- Returns a date value for the specified year, month, and day
- The data type of all the 3 parameters (year, month, and day) is integer
- If invalid argument values are specified, the function returns an error
- If any of the arguments are NULL, the function returns null

Syntax : DATEFROMPARTS (year, month, day)

Example : All the function arguments have valid values, so DATEFROMPARTS returns the expected date

```
SELECT DATEFROMPARTS ( 2015, 10, 25) AS [Date]
```

Output :



Date
25/10/2015

Other new date and time functions introduced in SQL Server 2012

- EOMONTH (Discussed in Part 125 of SQL Server tutorial)
- **DATETIMEFROMPARTS :** Returns DateTime
- **Syntax :** DATETIMEFROMPARTS (year, month, day, hour, minute, seconds, milliseconds)
- **SMALLDATETIMEFROMPARTS :** Returns SmallDateTime
- **Syntax :** SMALLDATETIMEFROMPARTS (year, month, day, hour, minute)

64) DateTime2FromParts function

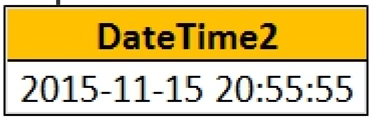
- Returns DateTime2
- The data type of all the parameters is integer
- If invalid argument values are specified, the function returns an error
- If any of the required arguments are NULL, the function returns null
- If the precision argument is null, the function returns an error

Syntax : DATETIME2FROMPARTS (year, month, day, hour, minute, seconds, fractions, precision)

Example : All the function arguments have valid values, so DATETIME2FROMPARTS returns DATETIME2 value as expected.

```
SELECT DATETIME2FROMPARTS ( 2015, 11, 15, 20, 55, 55, 0, 0 ) AS [DateTime2]
```

Output :



DateTime2
2015-11-15 20:55:55

TIMEFROMPARTS : Returns time value

Syntax : TIMEFROMPARTS (hour, minute, seconds, fractions, precision)

65) **OFFSET FETCH Clause**

- Returns a page of results from the result set
- ORDER BY clause is required

OFFSET FETCH Syntax :

```
SELECT * FROM Table_Name
ORDER BY Column_List
OFFSET Rows_To_Skip ROWS
```

```
FETCH NEXT Rows_To_Fetch ROWS ONLY
```

The following SQL query
1. Sorts the table data by Id column
2. Skips the first 10 rows and
3. Fetches the next 10 rows

```
SELECT * FROM tblProducts
ORDER BY Id
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY
```

66) Identifying Object Dependencies

- How to find object dependencies using the following dynamic management functions
- sys.dm_sql_referencing_entities
- sys.dm_sql_referenced_entities

- The following example returns all the objects that depend on Employees table.
  ```
  Select * from sys.dm_sql_referencing_entities('dbo.Employees','Object')
  ```

Difference between referencing entity and referenced entity
A dependency is created between two objects when one object appears by name inside a SQL statement stored in another object. The object which is appearing inside the SQL expression is known as referenced entity and the object which has the SQL expression is known as a referencing entity.

To get the REFERENCING ENTITIES use SYS.DM_SQL_REFERENCING_ENTITIES dynamic management function

To get the REFERENCED ENTITIES use SYS.DM_SQL_REFERENCED_ENTITIES dynamic management function

Now, let us say we have a stored procedure and we want to find the all objects that this stored procedure depends on. This can be very achieved using another dynamic management function, sys.dm_sql_referenced_entities.

The following query returns all the referenced entities of the stored procedure sp_GetEmployeesandDepartments

```
Select * from
sys.dm_sql_referenced_entities('dbo.sp_GetEmployeesandDepartments','Object')
```

Please note : For both these dynamic management functions to work we need to specify the schema name as well. Without the schema name you may not get any results.

Difference between Schema-bound dependency and Non-schema-bound dependency
Schema-bound dependency : Schema-bound dependency prevents referenced objects from being dropped or modified as long as the referencing object exists

Example : A view created with SCHEMABINDING, or a table created with foreign key constraint.

Non-schema-bound dependency : A non-schema-bound dependency doesn't prevent the referenced object from being dropped or modified.

- **sp_depends**
  A system stored procedure that returns object dependencies

For example,

- If you specify a table name as the argument, then the views and procedures that depend on the specified table are displayed
- If you specify a view or a procedure name as the argument, then the tables and views on which the specified view or procedure depends are displayed.

Syntax : Execute sp_depends 'ObjectName
EXample: sp_depends 'Employees'

67) Sequence object

- Generates sequence of numeric values in an ascending or descending order

Syntax :

```
CREATE SEQUENCE [schema_name . ] sequence_name
   [ AS [ built_in_integer_type | user-defined_integer_type ] ]
   [ START WITH <constant> ]
   [ INCREMENT BY <constant> ]
   [ { MINVALUE [ <constant> ] } | { NO MINVALUE } ]
   [ { MAXVALUE [ <constant> ] } | { NO MAXVALUE } ]
   [ CYCLE | { NO CYCLE } ]
   [ { CACHE [ <constant> ] } | { NO CACHE } ]
   [ ; ]
```

Property	Description
DataType	Built-in integer type (tinyint , smallint, int, bigint, decimal etc...) or user-defined integer type. Default bigint.
START WITH	The first value returned by the sequence object
INCREMENT BY	The value to increment or decrement by. The value will be decremented if a negative value is specified.
MINVALUE	Minimum value for the sequence object
MAXVALUE	Maximum value for the sequence object
CYCLE	Specifies whether the sequence object should restart when the max value (for incrementing sequence object) or min value (for decrementing sequence object) is reached. Default is NO CYCLE, which throws an error when minimum or maximum value is exceeded.
CACHE	Cache sequence values for performance. Default value is CACHE.

Creating an Incrementing Sequence : The following code create a Sequence object that starts with 1 and increments by 1

```
CREATE SEQUENCE [dbo].[SequenceObject]
AS INT
START WITH 1
INCREMENT BY 1
```

Generating the Next Sequence Value : Now we have a sequence object created. To generate the sequence value use NEXT VALUE FOR clause

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

Output : 1

Iter the Sequence object to reset the sequence value :

```
ALTER SEQUENCE [SequenceObject] RESTART WITH 1
```

Select the next sequence value to make sure the value starts from 1

```
SELECT NEXT VALUE FOR [dbo].[SequenceObject]
```

Using sequence value in an INSERT query :

```
CREATE TABLE Employees
(
   Id INT PRIMARY KEY,
   Name NVARCHAR(50),
   Gender NVARCHAR(10)
)

-- Generate and insert Sequence values
INSERT INTO Employees VALUES
(NEXT VALUE for [dbo].[SequenceObject], 'Ben', 'Male')
INSERT INTO Employees VALUES
(NEXT VALUE for [dbo].[SequenceObject], 'Sara', 'Female')
```

68) Difference between sequence and identity

- **Sequence object** is similar to the Identity property, in the sense that it generates sequence of numeric values in an ascending order just like the identity property.
- Identity property is a table column property meaning it is tied to the table, where as the sequence is a user-defined database object and is not tied to any specific table meaning it's value can be shared by multiple tables.