

SQL Training

SYED SIRAJ UL HAQUE (Siraj)

T-Mobile – WROPS

Document version: 1.4 (1/24/2018)

Contents

Introduction to the SSMS and Basic “Select statement”	6
SQL (Type of statements).....	9
DML (Data Manipulation Language)	9
MERGE – UPSERT operation (insert or update)	9
DDL (Data Definition Language).....	9
TRUNCATE – remove all records from a table (does not delete table)	9
DCL (Data Control Language)	9
TCL (Transaction Control Language)	9
Comments in SQL.....	10
Query Regions	10
Standard Operators.....	10
WHERE clause with Operators (LIKE / AND / OR / BETWEEN / NOT / IN)	11
A. Finding a row by using a simple equality	11
B. Finding rows that contain a value as a part of a string.....	11
C. Finding rows by using a comparison operator.....	11
D. Finding rows that meet any of three conditions	11
E. Finding rows that must meet several conditions.....	11
F. Finding rows that are in a list of values.....	11
G. Finding rows that have a value between two values	11
H. Not operator	11
Using [] with column name or table name.....	11
Group by and Having	12
“Group By” Clause.....	12
Having Clause	13
HAVING Vs Where.....	14
Difference between WHERE and HAVING clause:	14
Filtering Groups:.....	14
DISTINCT vs GROUP BY.....	14
CASE statement.....	15
SQL Server IIF vs CASE	16
Common String Functions.....	17
Sample Select queries	17
Joins.....	23
• Inner joins	23

• Outer joins.....	23
• Cross joins	24
• NATURAL JOIN (Not Supported in SQL Server)	25
Sample.....	27
Common Table Expressions (CTE) in SQL SERVER.....	29
When to use CTE	29
When to Use Common Table Expressions	30
Recursive loop with CTE.....	31
Temporary Tables	32
1. Local Temp Table	32
2. Global Temp Table	32
Table Variable	33
Working with NULL Values in SQL Server	34
What is NULL?	34
Considering NULLs When Designing Tables.....	34
Inserting NULLs	34
Querying NULLs.....	34
Updating NULLS	35
Deleting NULLs	35
Null Values and Joins.....	36
NOT IN clause with NULL values	37
NULL and Aggregation function (Like Count, MAX etc)	39
SQL Insert Statements.....	40
SQL Update Statements	40
SQL CREATE TABLE Statement	40
Composite PK	40
SQL Delete Statement.....	41
SQL TRUNCATE Statement.....	41
SQL DROP Statement:	41
Changing Data by Using a Cursor	42
Normalization.....	43
First Normal Form	43
Second Normal Form	43
Third Normal Form.....	44
Boyce-Codd Normal Form.....	45

Types of SQL Keys	46
Defining Keys in SQL Server	47
Stored Procedure and Functions	48
Stored Procedure	48
User Defined function	48
Difference between Stored Procedure and Function in SQL Server	48
Different Types of SQL Server Stored Procedures	49
Types of Stored Procedure.....	49
Different Types of SQL Server Functions	50
Types of Function	50
1. System Defined Function	50
2. User Defined Function	51
Note (SP vs Function)	53
Views	53
SQL Data Type	54
What is the difference between char, nchar, varchar, and nvarchar in SQL Server?	55
ASCII Vs Unicode	55
What is an index?	56
General Structure.....	56
Clustered and nonclustered	57
Considerations for creating indexes	57
Dealing with float point / Casting / Round off	58
Cast() Function	58
CONVERT() Function (SQL Server only).....	58
TRY_CAST	59
TRY_CONVERT.....	59
TRY_PARSE	59
Round ().....	60
Floor ().....	60
CEILING ()	60
Avoiding division by zero with NULLIF and COALESCE.....	61
NULLIF() Function	61
COALESCE() Function	61
Option 1 (using case)	61
Option 2 (using nullif)	61

Option 3 (using nullif and COALESCE)	61
DateTime (Casting / Formatting)	62
Convert string to date using style (format) numbers	62
Convert datetime to text style (format) list - sql time format	62
SQL date & time eliminating dividing characters.....	65
SQL DATEDIFF with string date	65
SQL Server date string search guidelines - comparing dates	65
SQL Server convert from string to smalldatetime	66
Translate/convert string/text hours and minutes to seconds	66

Introduction to the SSMS and Basic “Select statement”

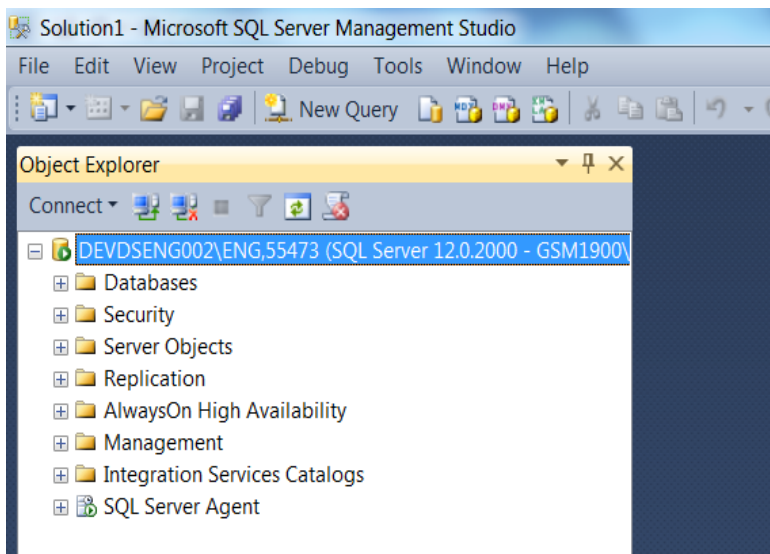


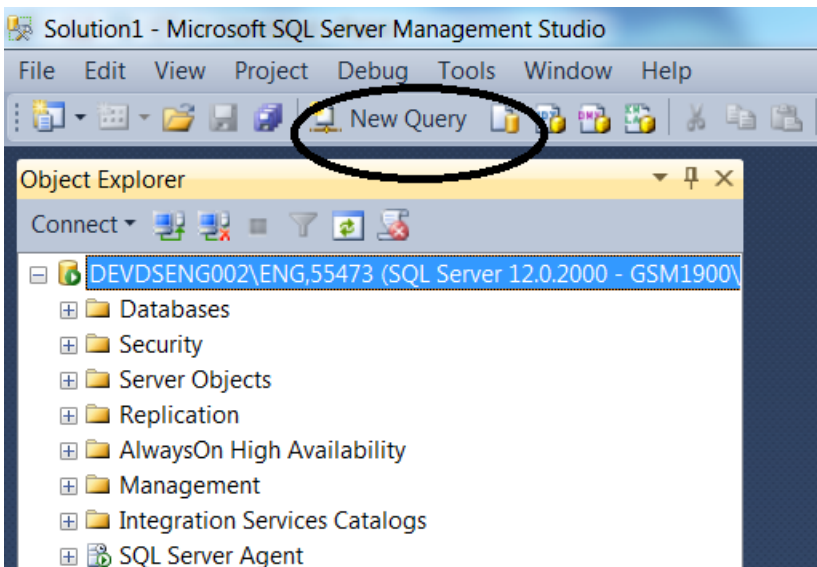
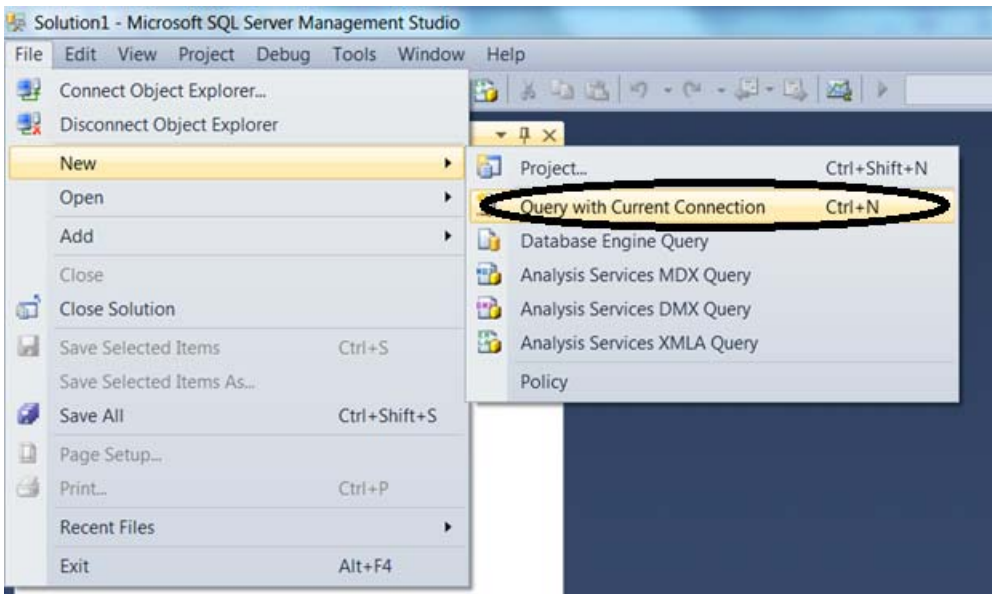
SSMS => SQL server management studio

Server Name: DEVDSENG002\ENG,55473 => “DEVDSENG002” is Machine name, “ENG” is instance name and “55473” is port number. These are set by database administrator, end users are supposed use it as it is, they don’t have option to change it. If you don’t see port in “server name” string, that means you are using default sql server port (The default instance of SQL Server listens on Port 1433) which does not need to be defined.

They are two **types of authentication** “Windows Authentication” and “SQL server authentication”. They are also set by DB administrator.

See SQL server instance in “**Object Explorer**” below. You can connect to multiple instance at same time. But you cannot write queries which involves multiple instances. In that scenario, you might want to use “linked server”.





Sql Is case in sensitive !!!!!

<https://solutions.eng.t-mobile.com/jira/servicedesk/customer/portal/13>

The screenshot shows the Microsoft SQL Server Enterprise Manager interface. The Object Explorer on the left shows the server hierarchy. The query window in the center contains the following query:

```
select top 100 * FROM [DM_Insite].[dbo].[INS_AllInfo]
```

The Results pane at the bottom displays a table with 14 columns: region, market_name, market_co..., operating..., mkt_regi..., ring, search_area_st..., latitude_search_a..., longitude_search_a..., site_name, siteid, sector, and cellid. The table contains 14 rows of data.

Annotations in the image point to the following components:

- Servers:** Points to the server list in the Object Explorer.
- SQL Query:** Points to the query text in the query window.
- Connection state (before execution) or Status of query (after execution):** Points to the status bar at the bottom.
- Instance name and version of the connected instance:** Points to the instance name in the Object Explorer.
- Login name & SPID:** Points to the login name in the Object Explorer.
- Current database:** Points to the current database in the Object Explorer.
- Time taken to execute:** Points to the time taken to execute in the status bar.
- No of rows:** Points to the number of rows in the status bar.

By default Current database is used (in this case [DW_GSMNetwork]), but in query you can define different database by just specifying it.

In given scenario (above Snapshot) this will work.

```
select top 100 * FROM [DM_Insite].[dbo].[INS_AllInfo]
SELECT TOP 100 * FROM DM_Insite.dbo.INS_AllInfo
```

This will work.

```
use [DM_Insite]
select top 100 * FROM [INS_AllInfo]
```

These wont work (because SSMS is assuming INS_AllInfo is in [DW_GSMNetwork])

```
select top 100 * FROM [dbo].[INS_AllInfo]
select top 100 * FROM [INS_AllInfo]
```

This wont work because schema is not defined in between table and database name

```
SELECT TOP 100 * FROM DM_Insite.INS_AllInfo
```

Each query editor window has its own connection (server connection) for example in above example

The screenshot shows two query editor windows. The first window, titled 'SQLQuery4.sql - 10...900\shaque2 (144)*', contains the query:

```
select top 100 * FROM [INS_AllInfo]
```

The second window, titled 'SQLQuery3.sql - DE...1900\shaque2 (59)', contains the query:

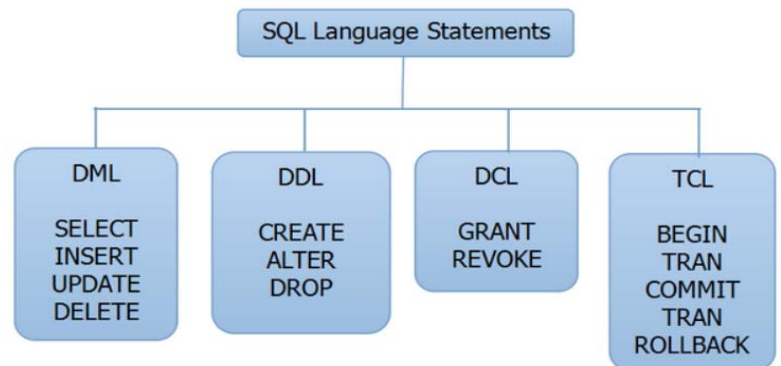
```
SQLQuery3.sql - DEVDSNG002\ENG,55473.master (GSM1900\shaque2 (59))
```


SQL (Type of statements)

SQL language is divided into four types of primary language statements: DML, DDL, DCL and TCL. Using these statements, we can define the structure of a database by creating and altering database objects and we can manipulate data in a table through updates or deletions. We also can control which user can read/write data or manage transactions to create a single unit of work

The 4 main categories of SQL statements are as follows:

1. DML (Data Manipulation Language)
2. DDL (Data Definition Language)
3. DCL (Data Control Language)
4. TCL (Transaction Control Language)



DML (Data Manipulation Language)

DML statements affect records in a table. These are basic operations we perform on data such as selecting a few records from a table, inserting new records, deleting unnecessary records, and updating/modifying existing records. DML statements include the following:

SELECT – select records from a table / Select is DRL (data retrieval Language) sub category of DML

INSERT – insert new records

UPDATE – update/Modify existing records

DELETE – delete existing records

MERGE – UPSERT operation (insert or update)

DDL (Data Definition Language)

DDL statements are used to alter/modify a database or table structure and schema. These statements handle the design and storage of database objects.

CREATE – create a new Table, database, schema

ALTER – alter existing table, column description

DROP – delete existing objects from database

TRUNCATE – remove all records from a table (does not delete table)

DCL (Data Control Language)

DCL statements control the level of access that users have on database objects.

GRANT – allows users to read/write on certain database objects

REVOKE – keeps users from read/write permission on database objects

TCL (Transaction Control Language)

TCL statements allow you to control and manage transactions to maintain the integrity of data within SQL statements.

BEGIN Transaction – opens a transaction

COMMIT Transaction – commits a transaction

ROLLBACK Transaction – ROLLBACK a transaction in case of any error

Comments in SQL

```
-- Single Line Comment
SELECT TOP 100 * FROM
INS_AllInfo
/* Multiple
Line Comments */
```

Query Regions

Prior to the Management Studio for SQL Server 2008 coming out, all code within the Query editor window was a giant block, which caused readability issues. Let's say for example that you are working on a lengthy (approximately more than 500 lines) stored procedure/script consisting of several logical sections. After the development of one particular section is complete, you may not need to look at it again, and might prefer to selectively hide it such that it does not interfere with your working area on the query editor. **The SSMS for SQL Server 2008 introduced a new usability feature – Query Regions.** A vertical line on the left edge of the editor window uses a square with a minus sign (-) to identify the start of each collapsible code region. When you click a minus sign, the text of the code region is replaced with a box that contains three periods (...), and the minus sign changes to a plus sign (+). Clicking on the (+) sign expands the code section.

When working on a database engine (T-SQL) query, the query editor generates outline regions in the following hierarchy:

1. Batches: From the start of the file to the first GO command (or till the end of the file in case no GO commands are present)
2. Blocks of code grouped by BEGIN...END keywords. This includes the following:
 - o BEGIN...END
 - o BEGIN TRY...END TRY
 - o BEGIN CATCH...END CATCH
3. Multi-line statements

```
begin --Antenna
SELECT [Antenna],[AIR_Antenna] FROM [TMO].[dbo].[Antenna]
end
```

```
+begin --Antenna...
```

```
begin /* Multiple
Line Comments */
SELECT TOP 100 * FROM ...
end
```

Ensure you have Outline Statement enabled under Text Editor > Transact-SQL>Intellisense>Enable Intellisense

Standard Operators

Symbol	Words
=	equals
!= or <>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
OR	" " is not operator in SQL
AND	"&&" is not operator in SQL

WHERE clause with Operators (LIKE / AND / OR / BETWEEN / NOT / IN)

The following examples show how to use some common search conditions in the WHERE clause.

A. Finding a row by using a simple equality

```
SELECT ProductID, Name
FROM Product
WHERE Name = 'Blade';
```

B. Finding rows that contain a value as a part of a string

```
SELECT ProductID, Name, Color
FROM Product
WHERE Name LIKE ('%Frame%');
```

C. Finding rows by using a comparison operator

```
SELECT ProductID, Name
FROM Product
WHERE ProductID <= 12;
```

D. Finding rows that meet any of three conditions

```
SELECT ProductID, Name
FROM Product
WHERE ProductID = 2    OR ProductID = 4    OR Name = 'Spokes';
```

E. Finding rows that must meet several conditions

```
SELECT ProductID, Name, Color
FROM Product
WHERE Name LIKE ('%Frame%') AND Name LIKE ('HL%') AND Color = 'Red';
```

F. Finding rows that are in a list of values

```
SELECT ProductID, Name, Color
FROM Product
WHERE Name IN ('Blade', 'Crown Race', 'Spokes');
```

G. Finding rows that have a value between two values

```
SELECT ProductID, Name, Color
FROM Product
WHERE ProductID BETWEEN 725 AND 734;
```

H. Not operator

```
SELECT ProductID, Name, Color
FROM Product
WHERE NOT ProductID BETWEEN 725 AND 734;
```

```
SELECT * FROM INS_AllInfo where !(market_name = 'Pittsburgh PA')  --this will not work
-- Don't use && instead of AND or || instead of OR
```

Using [] with column name or table name

If your column name does not have space in it, using [] does not matter. If it does you have to use it .

Example

Select myColumn from myTable is same as Select [myColumn] from myTable (both are correct)

If your column name was "my Column" instead of "myColumn" as above you have to use [] otherwise you get error.

Select [my Column] from myTable --- this will work

Select my Column from myTable --- this won't work

Group by and Having

“Group By” Clause

Group By clause is used for grouping the records of the database table(s). This clause creates a single row for each group and this process is called aggregation. To use group by clause we have to use at least one aggregate function in Select statement. We can use group by clause without where clause.

Syntax for Group By Clause

```
SELECT Col1, Col2, Aggregate_function FROM Table_Name  
WHERE Condition GROUP BY Col1, Col2
```

Let's see how the Group By clause works. Suppose we have a table StudentMarks that contains marks in each subject of the student.

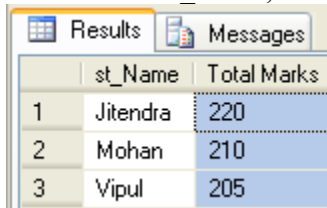
```
Create table StudentMarks  
(  
  st_RollNo int ,  
  st_Name varchar(50),  
  st_Subject varchar(50),  
  st_Marks int  
)  
--Insert data in StudentMarks table  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)  
values(1,'Mohan','Physics',75);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)  
values(1,'Mohan','Chemistry',65);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)  
values(1,'Mohan','Math',70);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks) values(2,'Vipul','Physics',70);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)  
values(2,'Vipul','Chemistry',75);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks) values(2,'Vipul','Math',60);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)  
values(3,'Jitendra','Physics',85);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)  
values(3,'Jitendra','Chemistry',75);  
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)  
values(3,'Jitendra','Math',60);
```

```
select * from StudentMarks
```

	st_RollNo	st_Name	st_Subject	st_Marks
1	1	Mohan	Math	70
2	1	Mohan	Physics	75
3	1	Mohan	Chemistry	65
4	2	Vipul	Physics	70
5	2	Vipul	Chemistry	75
6	2	Vipul	Math	60
7	3	Jitendra	Physics	85
8	3	Jitendra	Chemistry	75
9	3	Jitendra	Math	60

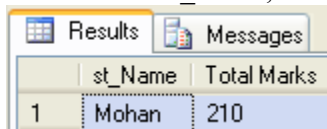
```
-- Group By clause without where condition  
SELECT st_Name, SUM(st_Marks) AS 'Total Marks'
```

```
FROM StudentMarks
GROUP BY st_Name;
```



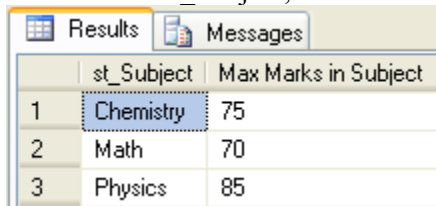
	st_Name	Total Marks
1	Jitendra	220
2	Mohan	210
3	Vipul	205

```
-- Group By clause with where condition
SELECT st_Name, SUM(st_Marks) AS 'Total Marks'
FROM StudentMarks
where st_Name='Mohan'
GROUP BY st_Name;
```



	st_Name	Total Marks
1	Mohan	210

```
-- Group By clause to find max marks in subject
SELECT st_Subject,max(st_Marks) AS 'Max Marks in Subject'
FROM StudentMarks
GROUP BY st_Subject;
```

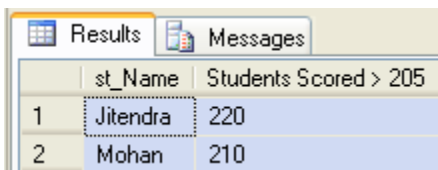


	st_Subject	Max Marks in Subject
1	Chemistry	75
2	Math	70
3	Physics	85

Having Clause

This clause operates only on group rows of table(s) and act as a filter like as where clause. We use having clause to filter data that we get from group by clause. To use having clause we need to use group by clause first.

```
-- Having clause without where condition
SELECT st_Name, SUM(st_Marks) AS 'Students Scored > 205'
FROM StudentMarks
GROUP BY st_Name
HAVING SUM(st_Marks) > 205
```



	st_Name	Students Scored > 205
1	Jitendra	220
2	Mohan	210

```
-- Having clause with where condition
SELECT st_Name, SUM(st_Marks) AS 'Students Scored > 205'
FROM StudentMarks
where st_RollNo between 1 and 3
GROUP BY st_Name
HAVING SUM(st_Marks) > 205
```

	st_Name	Students Scored > 205
1	Jitendra	220
2	Mohan	210

Note

1. To use Group By Clause, we need to use at least one aggregate function
2. All columns that are not used by aggregate function(s) must be in the Group By list
3. We can use Group By Clause with or without Where Clause.
4. To use Having Clause, we have to use Group By Clause since it filters data that we get from Group By Clause

HAVING Vs Where

```
SELECT column_name, column_name_tally
FROM (
    SELECT column_name, COUNT(column_name) AS column_name_tally
    FROM table_name
    WHERE column_name < 3
    GROUP BY column_name
) pointless_range_variable_required_here
WHERE column_name_tally >= 3;
```

Same as

```
SELECT column_name, COUNT( column_name ) AS column_name_tally
FROM table_name
WHERE column_name < 3
GROUP BY column_name
HAVING COUNT( column_name ) >= 3;
```

Difference between WHERE and HAVING clause:

1. **WHERE clause** can be used with - Select, Insert, and Update statements, where as **HAVING clause** can only be used with the Select statement.
2. **WHERE** filters rows before aggregation (GROUPING), whereas, **HAVING** filters groups, after the aggregations are performed.
3. Aggregate functions cannot be used in the **WHERE clause**, unless it is in a sub query contained in a **HAVING clause**, whereas, aggregate functions can be used in Having clause.

Filtering Groups:

WHERE clause is used to filter rows before aggregation, where as HAVING clause is used to filter groups after aggregations

```
Select City, SUM(Salary) as TotalSalary
from tblEmployee
Where Gender = 'Male'
group by City
Having City = 'London'
```

In SQL Server we have got lot of aggregate **functions**. **Examples**
Count() Sum() Avg() Min() Max()

DISTINCT vs GROUP BY

GROUP BY lets you use aggregate functions, like AVG, MAX, MIN, SUM, and COUNT. Other hand DISTINCT just removes duplicates.

CASE statement

The CASE expression has two formats:

- The **simple CASE expression** compares an expression to a set of simple expressions to determine the result. A simple CASE expression *allows for only an equality check*
- The **searched CASE expression** evaluates a set of Boolean expressions to determine the result. A searched CASE expression allows for values to be replaced in the result set *based on comparison values*

Both formats support an optional ELSE argument

Searched CASE expression

```
SELECT distinct [region],[market_name],
area = CASE
    when market_name in ('Irvine','LA North','Los Angeles','Southern california')
    then 'LA TRI MKT AREA'
    when market_name in ('Sacramento','Concord','San Francisco')
    then 'BAY AREA'
    else market_name
end
FROM [DM_Insite].[dbo].[INS_AllInfo] where region = 'west'
```

Same as

```
SELECT distinct [region],[market_name],
CASE
    when market_name in ('Irvine','LA North','Los Angeles','Southern california')
    then 'LA TRI MKT AREA'
    when market_name in ('Sacramento','Concord','San Francisco')
    then 'BAY AREA'
    else market_name
end as area
FROM [DM_Insite].[dbo].[INS_AllInfo] where region = 'west'
```

Simple CASE expression

```
SELECT distinct [region],[market_name],
CASE market_name
    WHEN 'Irvine' THEN 'LA TRI MKT AREA'
    WHEN 'LA North' THEN 'LA TRI MKT AREA'
    WHEN 'Los Angeles' THEN 'LA TRI MKT AREA'
    WHEN 'Southern california' THEN 'LA TRI MKT AREA'

    WHEN 'Sacramento' THEN 'BAY AREA'
    WHEN 'Concord' THEN 'BAY AREA'
    WHEN 'San Francisco' THEN 'BAY AREA'

    ELSE market_name
END as area
FROM [DM_Insite].[dbo].[INS_AllInfo] where region = 'west'
```

SQL Server IIF vs CASE

SQL Server 2012 has introduced the new logical function IIF(). The behavior of function IIF() is quite similar to CASE and IF statements in SQL Server. Using IIF(), you can use fewer lines of code, and your code will be more readable.

IIF is a non-standard T-SQL function. It was added to SQL SERVER 2012, so that Access could migrate to SQL Server without refactoring the IIF's to CASE before hand. Once the Access db is fully migrated into SQL Server, you can refactor.

Example

```
SELECT IIF(1 > 0,'True','False') AS 'Output'
SELECT IIF('10/15/2012' > '01/01/2012','Yes','No') AS 'Output'
```

```
DECLARE @num1 AS INT = 150
DECLARE @num2 AS INT = 100
SELECT IIF( @num1 < @num2, 'True', 'False') AS 'Output'
```

Let us contrast the various methods of comparing two strings by using IF, CASE, and IIF() Function.

///// IF

```
DECLARE @str as varchar(20) = 'tech-recipes'
if(@str = 'tech-recipes')
select 'Yes' AS 'OUTPUT'
ELSE
select 'No' AS 'OUTPUT'
GO
```

///// Case

```
DECLARE @str AS varchar(20) = 'tech-recipes'
select CASE when @str='tech-recipes'
THEN 'Yes'
ELSE 'No'
END AS 'Output'
GO
```

///// IIF

```
DECLARE @str as varchar(20) = 'tech-recipes'
select IIF(@str = 'tech-recipes', 'yes', 'no') as OUTPUT
GO
```

///// problem with IIF support binary output only => two outputs only like Yes or No

```
SELECT distinct [region],[market_name],
iif(
    market_name in ('Irvine','LA North','Los Angeles','Southern california'),
    'LA TRI MKT AREA',
    iif(market_name in ('Sacramento','Concord','San Francisco') , 'BAY AREA', market_name
)
) as area
FROM [DM_Insite].[dbo].[INS_AllInfo] where region = 'west'
```


Common String Functions

```
SELECT LEFT('abcdefg',2); --ab
SELECT LEN('abcdefg'); --7
SELECT LOWER('ABCD'); --abcd
SELECT UPPER('ABCD'); --ABCD
SELECT LTRIM(' abcdefg '); --abcdefg
SELECT RTRIM(' abcdefg '); -- abcdefg
SELECT CHARINDEX('bicycle', 'I have new bicycle'); --12
SELECT RIGHT('LEFT RIGHT',5) AS 'myRight' --RIGHT
SELECT LEFT('LEFT RIGHT',4) AS 'myLEFT' --LEFT
SELECT REPLACE('abcdefghicde','cde','xyz'); --abxyzfghixyz
```

Sample Select queries

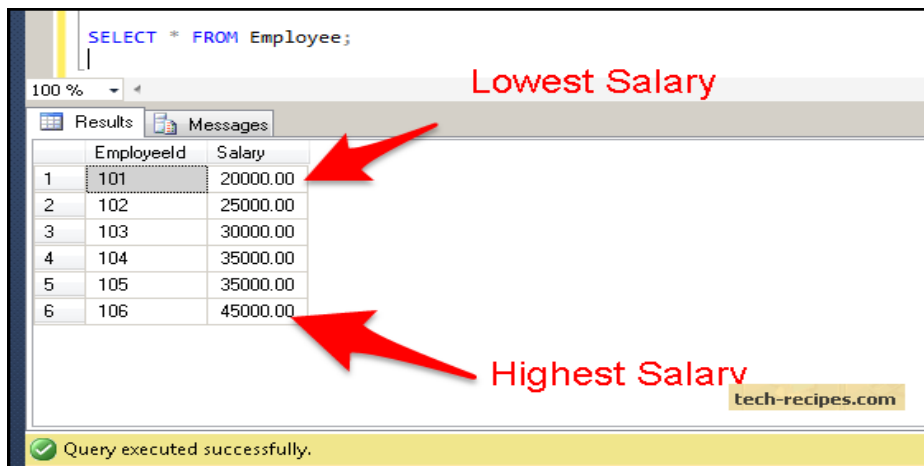
Let's create an employee table and populate it with some test data.

```
IF OBJECT_ID(N'Employee' , N'U' ) IS NOT NULL
DROP TABLE Employee;
```

```
CREATE TABLE Employee
(
EmployeeId INT PRIMARY KEY ,
Salary Numeric( 18,2 )
);
```

```
Insert into Employee Values ( 101,20000.00 );
Insert into Employee Values ( 102,25000.00 );
Insert into Employee Values ( 103,30000.00 );
Insert into Employee Values ( 104,35000.00 );
Insert into Employee Values ( 105,35000.00 );
Insert into Employee Values ( 106,45000.00 );
```

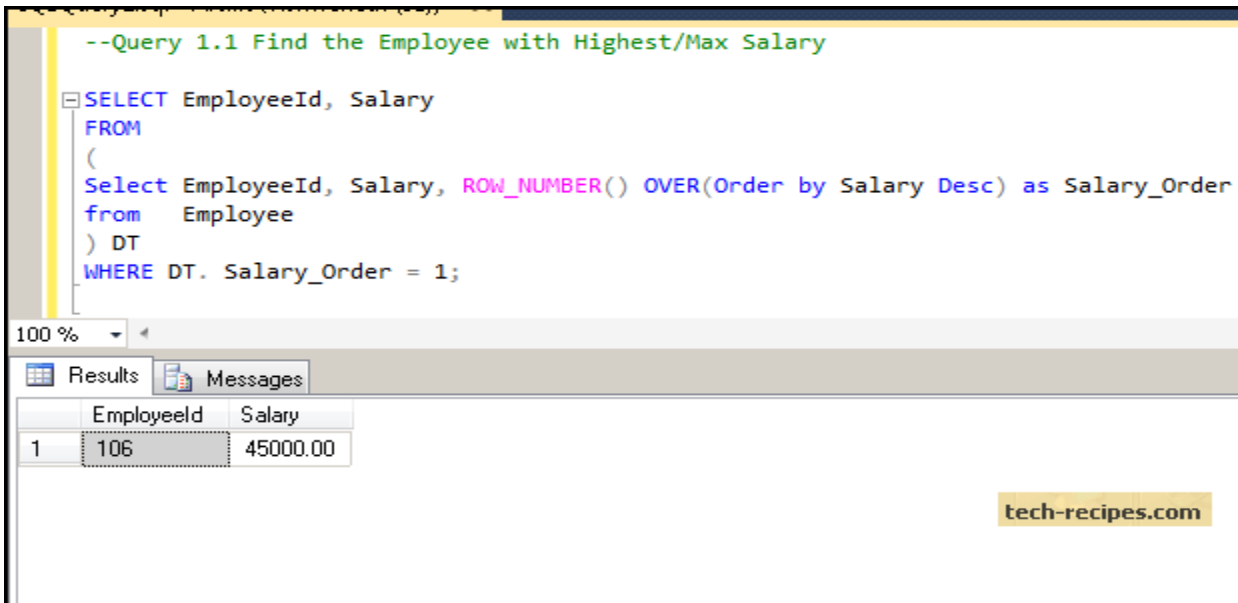
```
SELECT * FROM Employee
```



	EmployeeId	Salary
1	101	20000.00
2	102	25000.00
3	103	30000.00
4	104	35000.00
5	105	35000.00
6	106	45000.00

Query 1.1 Find the Employee with the Highest Salary

```
SELECT EmployeeId, Salary
FROM
(
Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary Desc) as Salary_Order
from Employee
) DT
WHERE DT. Salary_Order = 1 ;
```



The screenshot shows a SQL query editor with the following text:

```
--Query 1.1 Find the Employee with Highest/Max Salary

SELECT EmployeeId, Salary
FROM
(
Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary Desc) as Salary_Order
from Employee
) DT
WHERE DT. Salary_Order = 1;
```

Below the query editor, the 'Results' tab is active, displaying a table with the following data:

	EmployeeId	Salary
1	106	45000.00

A watermark 'tech-recipes.com' is visible in the bottom right corner of the screenshot.

The query above uses the derived table concept, where the subquery with row_number ranking function assigns a unique sequential number (1,2,3.. to N) to a salary which is ordered in descending order (highest to lowest). Thus, 1 will be assigned to the highest salary, 2 to the second highest salary, and so on, using the derived table to fetch the row with row_number assigned as 1.

Query 1.2 Find the Employee with the Highest Salary When There Is a Tie (Two employees both have the highest salary and the number is the same)

I am inserting one more employee whose salary matches the highest salary fetched using Query 1.1 to demonstrate this example.

```
Insert into Employee Values ( 107,45000.00 );
```

```
SELECT EmployeeId, Salary
FROM
( Select EmployeeId, Salary, DENSE_RANK() OVER(Order by Salary Desc) as Salary_Order
from Employee ) DT
WHERE DT. Salary_Order = 1 ;
```

Here we are deleting employee id 107 which we had inserted for the Query 1.2 demonstration.

```
Delete from employee where EmployeeId = 107;
```


```
--Find the Employee with Highest Salary When There is Tie
--(Same Highest salary for two employees)

SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, DENSE_RANK() OVER(Order by Salary Desc) as Salary_Order
  from Employee
) DT
WHERE DT.Salary_Order = 1;
```

100 %

Results Messages

	EmployeeId	Salary
1	106	45000.00
2	107	45000.00

 **Highest Salary with a Tie**

tech-recipes.com

In the query above, the Dense_rank function assigns the same consecutive rank number when there is a tie. Therefore, it assigns number 1 to both of the highest salaries (45,000), and both are returned using the derived table query with the salary_order = 1 filter.

Query 1.3 Finding the Employee with the Second Highest Salary

```
SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary Desc) as Salary_Order
  from Employee
) DT
WHERE DT. Salary_Order = 2 ;
```

SQLQuery1.sql - AN...0\Wishwanath (52))* X

```
--Query 1.3 --Finding Employee with 2nd Highest Salary

SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary Desc) as Salary_Order
  from Employee
) DT
WHERE DT.Salary_Order = 2;
```

100 %

Results Messages

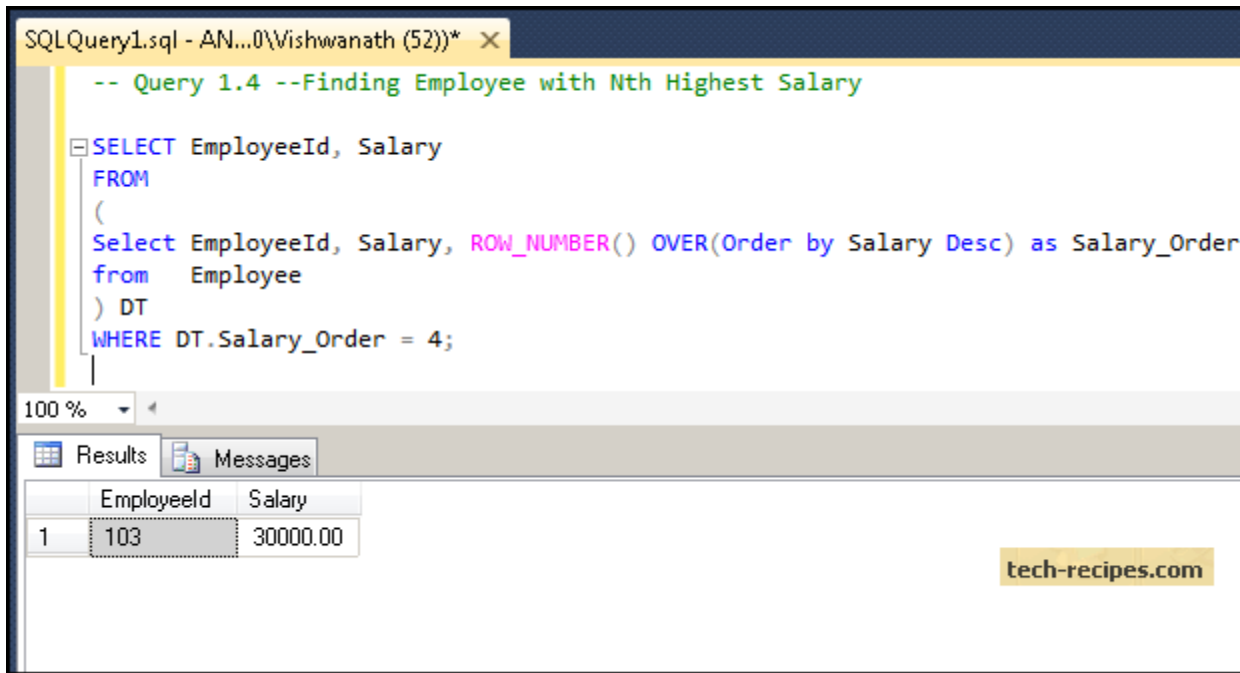
	EmployeeId	Salary
1	104	35000.00

tech-recipes.com

Here we are using the same logic used in [Query 1.1](#) with the ROW_NUMBER() function, but we are using Salary_order = 2 to fetch second Highest salary.

Query 1.4 Finding the Employee with the Nth Highest Salary

```
SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary Desc) as Salary_Order
  from Employee
) DT
WHERE DT. Salary_Order = 4 ;
```



The screenshot shows a SQL Server query window titled "SQLQuery1.sql - AN...0\Wishwanath (52))* X". The query text is as follows:

```
-- Query 1.4 --Finding Employee with Nth Highest Salary

SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary Desc) as Salary_Order
  from Employee
) DT
WHERE DT. Salary_Order = 4;
```

The Results pane shows the following data:

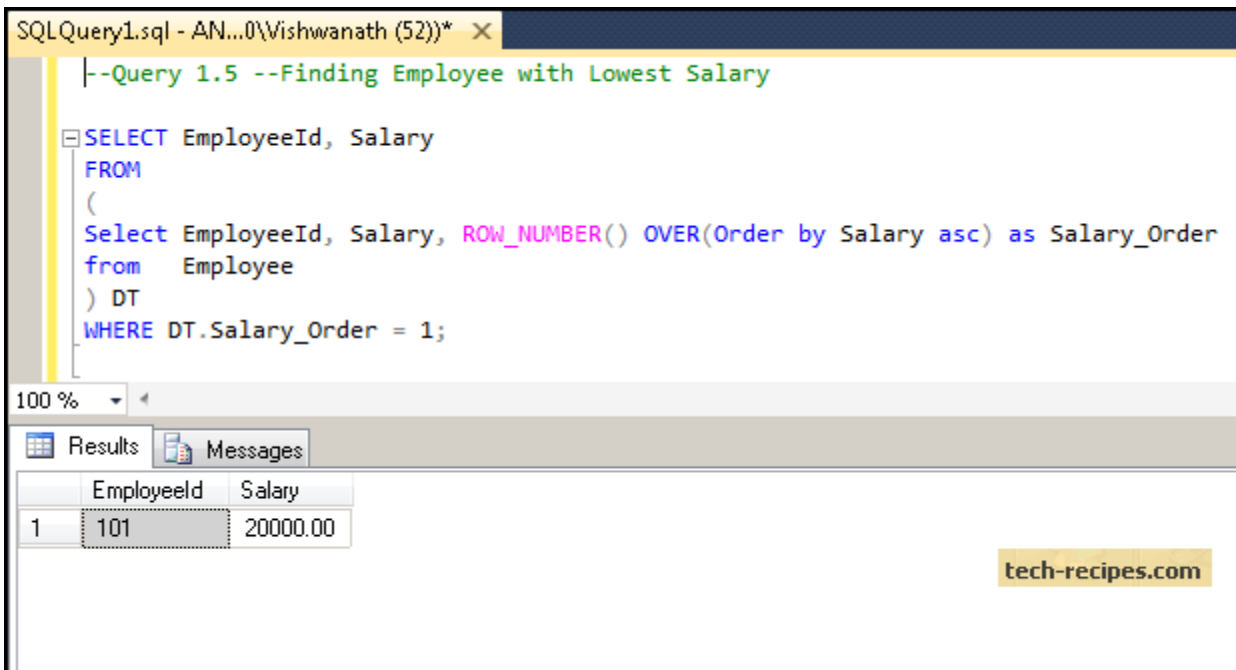
	EmployeeId	Salary
1	103	30000.00

A watermark "tech-recipes.com" is visible in the bottom right corner of the screenshot.

Here we are using the same logic used in [Query 1.1](#) and [Query 1.3](#). Nth means you can specify any number, and the query will retrieve the salary at Nth number.

Query 1.5 Finding the Employee with the Lowest Salary

```
SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary asc) as Salary_Order
  from Employee
) DT
WHERE DT. Salary_Order = 1 ;
```



The screenshot shows a SQL Server Enterprise Manager window titled "SQLQuery1.sql - AN...0\Vishwanath (52))* X". The query editor contains the following SQL code:

```
--Query 1.5 --Finding Employee with Lowest Salary

SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary asc) as Salary_Order
  from Employee
) DT
WHERE DT.Salary_Order = 1;
```

Below the query editor, the "Results" tab is active, displaying a table with two columns: "EmployeeId" and "Salary". The table contains one row with the values 101 and 20000.00.

	EmployeeId	Salary
1	101	20000.00

A watermark "tech-recipes.com" is visible in the bottom right corner of the screenshot.

To find the lowest salary, we are using Order by salary in ascending order, so the result is sorted in ascending order for salary (lowest to highest). Hence, the lowest salary will get row_number = 1 and so on. We are using the filter Salary_Order = 1 to retrieve the first lowest salary in the employee table.

Query 1.6 Finding the Employee with the Lowest Salary When There Is a Tie (Two employees both have the lowest salary and it is the same)

I am inserting one more employee whose salary matches the lowest salary fetched using the query above to demonstrate this example.

```
Insert into Employee Values (109,20000.00);
SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, DENSE_RANK() OVER(Order by Salary asc) as Salary_Order
  from Employee
) DT
WHERE DT. Salary_Order = 1 ;
```

Here we are deleting the employee with 108 id which was inserted to demonstrate the query above.

```
delete from employee where employeeid = 109;
```

```
SQLQuery1.sql - AN...0\Vishwanath (52))* X
--Query 1.6 --Finding Employee with Lowest Salary when there is a Tie
--(Same Lowest salary for two employees)

Insert into Employee Values (109,20000.00);

SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, DENSE_RANK() OVER(Order by Salary asc) as Salary_Order
  from Employee
) DT
WHERE DT.Salary_Order = 1;

delete from employee where employeeid = 109;
```

100 %

Results Messages

	EmployeeId	Salary
1	101	20000.00

tech-recipes.com

To find the lowest salary with ties, we are using the dense_rank function which is the same as [Query 1.2](#). The dense_rank function will assign consecutive numbers where there is a duplicate salary, so for the lowest salary (20000.00), it will assign the number 1 to both the salaries. Using the Salary_Order = 1 filter, we are able to retrieve both the lowest salary when there is tie using the dense_rank function.

Query 1.7 Finding the Employee with the Second Lowest Salary

```
SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary asc) as Salary_Order
  from Employee
) DT
WHERE DT. Salary_Order = 2
```

```
SQLQuery1.sql - AN...0\Vishwanath (52))* X
--Query 1.7 --Finding Employee with 2nd Lowest Salary

SELECT EmployeeId, Salary
FROM
(
  Select EmployeeId, Salary, ROW_NUMBER() OVER(Order by Salary asc) as Salary_Order
  from Employee
) DT
WHERE DT.Salary_Order = 2;
```

100 %

Results Messages

	EmployeeId	Salary
1	102	25000.00

tech-recipes.com

Here we are using the same logic used in [Query 1.3](#) with the ROW_NUMBER() function, but we are using Salary_order = 2 to fetch second lowest salary.

Joins

Join conditions can be specified in either the FROM or WHERE clauses; specifying them in the FROM clause is recommended. WHERE and HAVING clauses can also contain search conditions to further filter the rows selected by the join conditions. Joins can be categorized as:

- **Inner joins**

The typical join operation, which uses some comparison operator like = or <>. These include equi-joins and natural joins. Inner joins use a comparison operator to match rows from two tables based on the values in common columns from each table. For example, retrieving all rows where the student identification number is the same in both the students and courses tables.

Example

```
SELECT *
FROM Employee AS e
     INNER JOIN Person AS p
       ON e.BusinessEntityID = p.BusinessEntityID
ORDER BY p.LastName
```

```
SELECT DISTINCT p.ProductID, p.Name, p.ListPrice, sd.UnitPrice AS 'Selling Price'
FROM SalesOrderDetail AS sd
     JOIN Product AS p
       ON sd.ProductID = p.ProductID AND sd.UnitPrice < p.ListPrice
WHERE p.ProductID = 718;
```

```
SELECT DISTINCT p1.ProductSubcategoryID, p1.ListPrice
FROM Product p1
     INNER JOIN Product p2
       ON p1.ProductSubcategoryID = p2.ProductSubcategoryID
       AND p1.ListPrice <> p2.ListPrice
WHERE p1.ListPrice < $15 AND p2.ListPrice < $15
ORDER BY ProductSubcategoryID;
```

"INNER JOIN" or just "JOIN" both represent inner join.

- **Outer joins**

Outer joins can be a left, a right, or full outer join. Outer joins are specified with one of the following sets of keywords when they are specified in the FROM clause:

- **LEFT JOIN or LEFT OUTER JOIN**

The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause, not just the ones in which the joined columns match. When a row in the left table has no matching rows in the right table, the associated result set row contains null values for all select list columns coming from the right table.

```
SELECT p.Name, pr.ProductReviewID
FROM Product p
     LEFT OUTER JOIN ProductReview pr
       ON p.ProductID = pr.ProductID
```

- **RIGHT JOIN or RIGHT OUTER JOIN**

A right outer join is the reverse of a left outer join. All rows from the right table are returned. Null values are returned for the left table any time a right table row has no matching row in the left table.

```
SELECT st.Name AS Territory, sp.BusinessEntityID
FROM Sales.SalesTerritory st
RIGHT OUTER JOIN Sales.SalesPerson sp
ON st.TerritoryID = sp.TerritoryID
WHERE st.SalesYTD < $2000000;
```

- **FULL JOIN or FULL OUTER JOIN**

A full outer join returns all rows in both the left and right tables. Any time a row has no match in the other table, the select list columns from the other table contain null values. When there is a match between the tables, the entire result set row contains data values from the base tables.

```
-- The OUTER keyword following the FULL keyword is optional.
SELECT p.Name, sod.SalesOrderID
FROM Product p
FULL OUTER JOIN SalesOrderDetail sod
ON p.ProductID = sod.ProductID
WHERE p.ProductID IS NULL
OR sod.ProductID IS NULL
ORDER BY p.Name ;
```

- **Cross joins**

Cross joins return all rows from the left table. Each row from the left table is combined with all rows from the right table. Cross joins are also called Cartesian products.

```
SELECT p.BusinessEntityID, t.Name AS Territory
FROM SalesPerson p
CROSS JOIN SalesTerritory t
ORDER BY p.BusinessEntityID;
```

However, if a WHERE clause is added, the cross join behaves as an inner join. For example, the following Transact-SQL queries produce the same result set.

```
SELECT p.BusinessEntityID, t.Name AS Territory
FROM SalesPerson p
CROSS JOIN SalesTerritory t
WHERE p.TerritoryID = t.TerritoryID
ORDER BY p.BusinessEntityID;
```

-- this gives same result

```
SELECT p.BusinessEntityID, t.Name AS Territory
FROM SalesPerson p
INNER JOIN SalesTerritory t
ON p.TerritoryID = t.TerritoryID
ORDER BY p.BusinessEntityID;
```


- **NATURAL JOIN (Not Supported in SQL Server)**

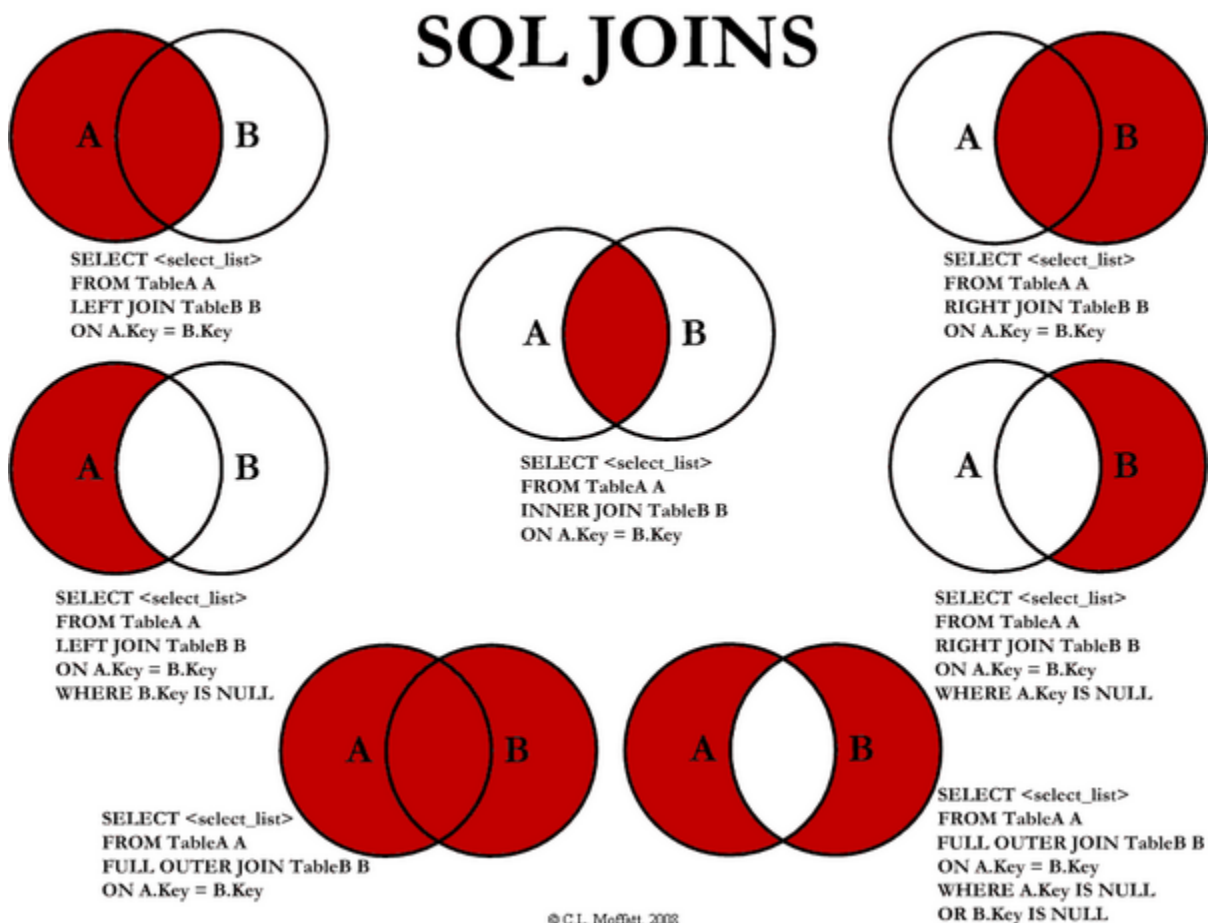
The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with same name of associate tables will appear once only.

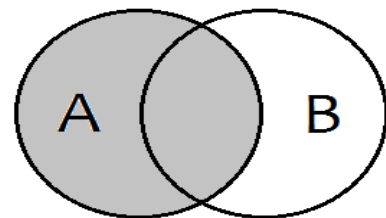
The associated tables have one or more pairs of identically named columns.

The columns must be the same data type.

Don't use ON clause in a natural join.

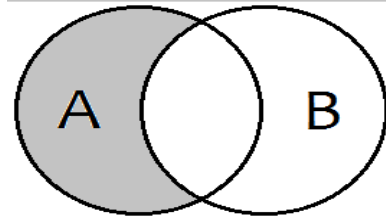
```
SELECT *
FROM table_A
NATURAL JOIN table_B;
```



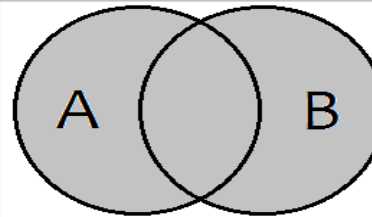


SELECT *
FROM TableA a
LEFT JOIN TableB b
ON a.Key = b.Key

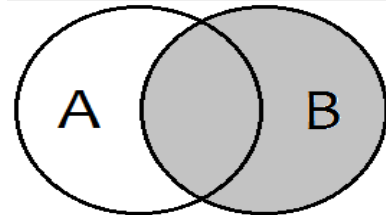
SQL JOINS



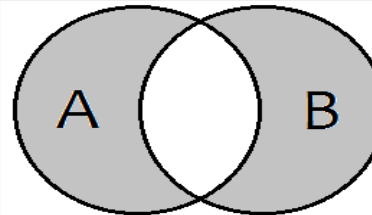
SELECT *
FROM TableA a
LEFT JOIN TableB b
ON a.Key = b.Key
WHERE b.Key IS NULL



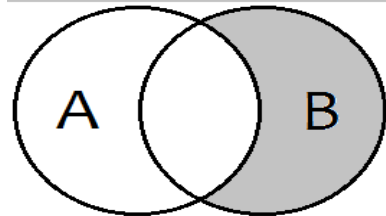
SELECT *
FROM TableA a
FULL OUTER JOIN TableB b
ON a.Key = b.Key



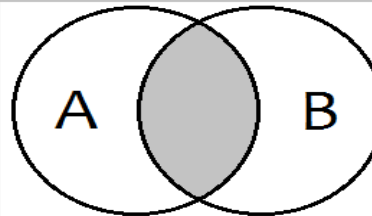
SELECT *
FROM TableA a
RIGHT JOIN TableB b
ON a.Key = b.Key



SELECT *
FROM TableA a
FULL OUTER JOIN TableB b
ON a.Key = b.Key
WHERE a.Key IS NULL
OR b.Key IS NULL

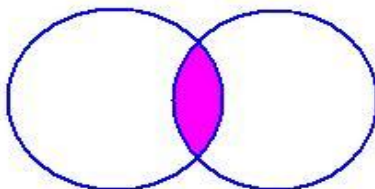


SELECT *
FROM TableA a
RIGHT JOIN TableB b
ON a.Key = b.Key
WHERE a.Key IS NULL

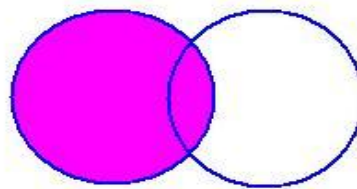


SELECT *
FROM TableA a
INNER JOIN TableB b
ON a.Key = b.Key

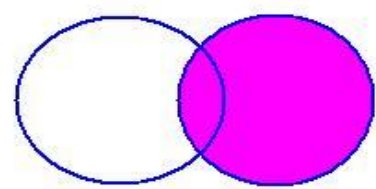
JOINS AND SET OPERATIONS IN RELATIONAL DATABASES



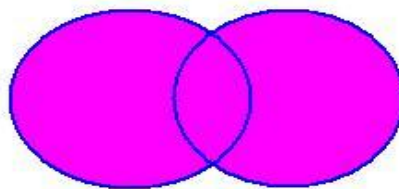
Inner join (result similar to Intersect)



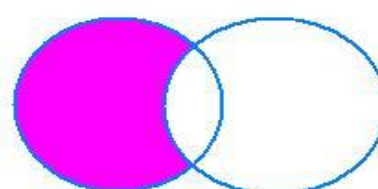
Left outer join



Right outer join



Full outer join



Minus

Sample

```
SELECT [TB_KEY],[VAL] FROM [SQL_Training].[dbo].[Table_A]
```

TB_KEY	VAL
1	10
2	100
3	1000
4	10000
0	0

```
SELECT [TB_KEY],[DESC] FROM [SQL_Training].[dbo].[Table_B]
```

TB_KEY	DESC
1	Ten
2	Hundred
4	Ten thousand
5	Hundred thousand

Inner Join

```
SELECT [Table_A].TB_KEY, [VAL], [Table_B].TB_KEY, [DESC]  
FROM [Table_A] inner join [Table_B] On [Table_A].TB_KEY = [Table_B].TB_KEY
```

TB_KEY	VAL	TB_K...	DESC
1	10	1	Ten
2	100	2	Hundred
4	10000	4	Ten thousand

Left Join

```
SELECT [Table_A].TB_KEY, [VAL], [Table_B].TB_KEY, [DESC]  
FROM [Table_A] Left join [Table_B] On [Table_A].TB_KEY = [Table_B].TB_KEY
```

TB_KEY	VAL	TB_K...	DESC
1	10	1	Ten
2	100	2	Hundred
3	1000	NULL	NULL
4	10000	4	Ten thousand
0	0	NULL	NULL

Right Join

```
SELECT [Table_A].TB_KEY, [VAL], [Table_B].TB_KEY, [DESC]  
FROM [Table_A] Right join [Table_B] On [Table_A].TB_KEY = [Table_B].TB_KEY
```

TB_KEY	VAL	TB_K...	DESC
1	10	1	Ten
2	100	2	Hundred
4	10000	4	Ten thousand
NULL	NULL	5	Hundred thousand

Minus Join

```
SELECT [Table_A].TB_KEY, [VAL], [Table_B].TB_KEY, [DESC]
FROM [Table_A]
left join [Table_B]
On [Table_A].TB_KEY = [Table_B].TB_KEY
where [Table_B].TB_KEY is null
```

TB_KEY	VAL	TB_K...	DESC
3	1000	NULL	NULL
0	0	NULL	NULL

```
SELECT [Table_A].TB_KEY, [VAL], [Table_B].TB_KEY, [DESC]
FROM [Table_A]
right join [Table_B]
On [Table_A].TB_KEY = [Table_B].TB_KEY
where [Table_A].TB_KEY is null
```

TB_KEY	VAL	TB_K...	DESC
NULL	NULL	5	Hundred thousand

Full Outer Join

```
SELECT [Table_A].TB_KEY, [VAL], [Table_B].TB_KEY, [DESC]
FROM [Table_A]
full join [Table_B]
On [Table_A].TB_KEY = [Table_B].TB_KEY
```

TB_KEY	VAL	TB_K...	DESC
1	10	1	Ten
2	100	2	Hundred
3	1000	NULL	NULL
4	10000	4	Ten thousand
0	0	NULL	NULL
NULL	NULL	5	Hundred thousand

TB_KEY	VAL	TB_K...	DESC
1	10	1	Ten
2	100	1	Ten
3	1000	1	Ten
4	10000	1	Ten
0	0	1	Ten
1	10	2	Hundred
2	100	2	Hundred
3	1000	2	Hundred
4	10000	2	Hundred
0	0	2	Hundred
1	10	4	Ten thousand
2	100	4	Ten thousand
3	1000	4	Ten thousand
4	10000	4	Ten thousand
0	0	4	Ten thousand
1	10	5	Hundred thousand
2	100	5	Hundred thousand
3	1000	5	Hundred thousand
4	10000	5	Hundred thousand
0	0	5	Hundred thousand

Cross Join

```
SELECT [Table_A].TB_KEY, [VAL], [Table_B].TB_KEY, [DESC]
FROM [Table_A] cross join [Table_B]
```

Common Table Expressions (CTE) in SQL SERVER

CTE stands for Common Table expressions. It is a temporary result set and typically it may be a result of complex sub-query. Unlike temporary table its life is limited to the current query.

When to use CTE

1. This is used to store result of a complex sub query for further use.
2. This is also used to create a recursive query.

It is defined by using WITH statement (You can specify the column names in braces, but it is not mandatory). CTE improves readability and ease in maintenance of complex queries and sub-queries. Always begin CTE with semicolon to show termination of pervious statement if any (not required).

A sub query without CTE is given below :

```
SELECT * FROM
(
    SELECT Addr.Address, Emp.Name, Emp.Age From Address Addr
    Inner join Employee Emp on Emp.EID = Addr.EID
) Temp
WHERE Temp.Age > 50
ORDER BY Temp.NAME
```

By using CTE above query can be re-written as follows :

```
With CTE1(Address, Name, Age)    --Column names for CTE, which are optional
AS
(
    SELECT Addr.Address, Emp.Name, Emp.Age from Address Addr
    INNER JOIN EMP Emp ON Emp.EID = Addr.EID
)

SELECT * FROM CTE1 --Using CTE
WHERE CTE1.Age > 50
ORDER BY CTE1.NAME
```

It's a headache for developers to write or read a complex SQL query using a number of Joins. Complex SQL statements can be made easier to understand and maintainable in the form of CTE or Common Table expressions. CTE allows you to define the subquery at once, name it using an alias and later call the same data using the alias just like what you do with a normal table. CTE is standard ANSI SQL standard.

For instance, you have a query like this:

```
SELECT * FROM (
    SELECT A.Address, E.Name, E.Age From Address A
    Inner join Employee E on E.EID = A.EID
) T
WHERE T.Age > 50
ORDER BY T.NAME
```

The query looks really a mess. Even if I need to write something that wraps around the entire query, it would gradually become unreadable. CTE allows you to generate Tables beforehand and use it later when we actually bind the data into the output.

Rewriting the query using CTE expressions would look like:

```
;With T(Address, Name, Age) --Column names for Temporary table
AS
(
SELECT A.Address, E.Name, E.Age from Address A
INNER JOIN EMP E ON E.EID = A.EID
)

SELECT * FROM T --SELECT or USE CTE temporary Table
WHERE T.Age > 50
ORDER BY T.NAME
```

Yes as you can see, the second query is much more readable using CTE. You can specify as many query expressions as you want and the final query which will output the data to the external environment will eventually get reference to all of them.

When to Use Common Table Expressions

Common Table Expressions offer the same functionality as a view, but are ideal for one-off usages where you don't necessarily need a view defined for the system. Even when a CTE is not necessarily needed, it can improve readability. In Using Common Table Expressions, Microsoft offers the following four advantages of CTEs:

- Create a recursive query.
- Substitute for a view when the general use of a view is not required; that is, you do not have to store the definition in metadata.
- Enable grouping by a column that is derived from a scalar subselect, or a function that is either not deterministic or has external access.
- Reference the resulting table multiple times in the same statement.

Using a CTE offers the advantages of improved readability and ease in maintenance of complex queries. The query can be divided into separate, simple, logical building blocks. These simple blocks can then be used to build more complex, interim CTEs until the final result set is generated.

Using scalar subqueries (such as the (`SELECT COUNT(1) FROM ...`)) cannot be grouped or filtered directly in the containing query. Similarly, when using SQL Server 2005's ranking functions - `ROW_NUMBER()`, `RANK()`, `DENSE_RANK()`, and so on - the containing query cannot include a filter or grouping expression to return only a subset of the ranked results. For both of these instances, CTEs are quite handy.

CTEs can also be used to recursively enumerate hierarchical data.

Recursive loop with CTE

See the awful series with CTE:

```
WITH ShowMessage(STATEMENT, LENGTH)
AS
(
    SELECT STATEMENT = CAST('I Like ' AS VARCHAR(300)), LEN('I Like ')
    UNION ALL
    SELECT
        CAST(STATEMENT + 'CodeProject! ' AS VARCHAR(300)),
        LEN(STATEMENT) FROM ShowMessage
    WHERE LENGTH < 300
)

SELECT STATEMENT, LENGTH FROM ShowMessage
```

So this will produce like this:

	STATEMENT
1	I Like
2	I Like CodeProject!
3	I Like CodeProject! CodeProject!
4	I Like CodeProject! CodeProject! CodeProject!
5	I Like CodeProject! CodeProject! CodeProject! CodeProject!
6	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
7	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
8	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
9	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
10	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
11	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
12	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
13	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
14	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
15	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
16	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!
17	I Like CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject! CodeProject!

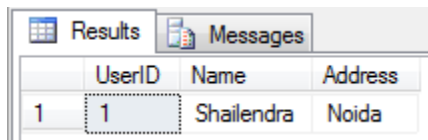
Temporary Tables

In SQL Server, temporary tables are created at run-time and you can do all the operations which you can do on a normal table. These tables are created inside Tempdb database. Based on the scope and behavior temporary tables are of two types as given below-

1. Local Temp Table

Local temp tables are only available to the SQL Server session or connection (means single user) that created the tables. These are automatically deleted when the session that created the tables has been closed. Local temporary table name is started with single hash ("#") sign.

```
CREATE TABLE #LocalTemp
(
    UserID int,
    Name varchar(50),
    Address varchar(150)
)
GO
insert into #LocalTemp values ( 1, 'Shailendra','Noida');
GO
Select * from #LocalTemp
```



The screenshot shows a SQL Server Results window with a single row of data. The columns are UserID, Name, and Address. The values are 1, Shailendra, and Noida respectively.


	UserID	Name	Address
1	1	Shailendra	Noida

The scope of Local temp table exist to the current session of current user means to the current query window. If you will close the current query window or open a new query window and will try to find above created temp table, it will give you the error.

2. Global Temp Table

Global temp tables are available to all SQL Server sessions or connections (means all the user). These can be created by any SQL Server connection user and these are automatically deleted when all the SQL Server connections have been closed. Global temporary table name is started with double hash ("##") sign.

```
CREATE TABLE ##GlobalTemp
(
    UserID int, Name varchar(50), Address varchar(150)
)
GO
insert into ##GlobalTemp values ( 1, 'Shailendra','Noida');
GO
Select * from ##GlobalTemp
```



The screenshot shows a SQL Server Results window with a single row of data. The columns are UserID, Name, and Address. The values are 1, Shailendra, and Noida respectively.

	UserID	Name	Address
1	1	Shailendra	Noida

Global temporary tables are visible to all SQL Server connections while Local temporary tables are visible to only current SQL Server connection.

Table Variable

This acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory. This also allows you to create primary key, identity at the time of Table variable declaration but not non-clustered index.

```
DECLARE @TProduct TABLE
(
    SNo INT IDENTITY(1,1),
    ProductID INT,
    Qty INT
)

--Insert data to Table variable @Product
INSERT INTO @TProduct(ProductID,Qty)
SELECT DISTINCT ProductID, Qty FROM ProductsSales ORDER BY ProductID ASC

--Select data
Select * from @TProduct

--Next batch
GO
Select * from @TProduct --gives error in next batch
```

Note

1. Temp Tables are physically created in the Tempdb database. These tables act as the normal table and also can have constraints, index like normal tables.
2. CTE is a named temporary result set which is used to manipulate the complex sub-queries data. This exists for the scope of statement. This is created in memory rather than Tempdb database. You cannot create any index on CTE.
3. Table Variable acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory.

Working with NULL Values in SQL Server

What is NULL?

NULL is a keyword that signifies that no value exists.

Considering NULLs When Designing Tables

We need to determine which fields are going to allow NULL values. When designing tables, we need to consider which fields are required or mandatory to be filled in by the user and which fields are not mandatory. Based on this, we can decide whether to allow NULL values.

```
CREATE TABLE Customer
(
Customer_id INT PRIMARY KEY,
FirstName VARCHAR(50) NOT NULL,
LastName VARCHAR(50) NOT NULL,
MobileNo VARCHAR(15) NULL
);
```

We have designed a table, Customer, in which every field is required (NOT NULL), except the MobileNo field which allows NULL values.

Note: Primary key constraints are always NOT NULL, so there is no need to specify it explicitly.

NOT NULL indicates a value is mandatory (to be supplied).

NULL indicates the value is not mandatory.

Inserting NULLs

We are using NULL for the mobile number in Case 2 because we do not have a Mobile Number for CustomerId = 2 right now. Make sure you do not put single quotes around NULL as it will be considered as a string where NULL is a special keyword.

Case 1 : We have a valid MobileNo.

```
Insert Into dbo.Customer (Customer_id, FirstName, LastName, MobileNo)
Values (1, 'Hen', 'Kaz', 9833844);
```

Case 2 : We do not have a MobileNo and, hence, we are inserting it as NULL.

```
Insert Into dbo.Customer (Customer_id, FirstName, LastName, MobileNo)
values (2, 'Rec', 'John', NULL);
```

Querying NULLs

Querying NULLs in SQL is different from querying regular data because we cannot query for the string literal 'NULL' or search for a value equal to NULL.

Case 1 : Incorrect Query

```
SELECT * FROM CUSTOMER
WHERE MobileNo = NULL
```

Case 2 : Incorrect Query

```
SELECT * FROM CUSTOMER
WHERE MobileNo like 'NULL'
```

Case 3 : Valid Query to Find MobileNos Having a NULL Value

```
SELECT * FROM CUSTOMER
WHERE MobileNo IS NULL
```

Case 4 : Valid Query to Find MobileNos Not Having a NULL Value

```
SELECT * FROM CUSTOMER
WHERE MobileNo IS NOT NULL
```

Updating NULLS

Now, we have received the mobile number for customer id = 2, which earlier was NULL.

Let us update the mobile number for customer id = 2, where the mobile number is now NULL.

```
--For customer id 2 whose MobileNo is now NULL
```

```
SELECT * FROM CUSTOMER
WHERE MobileNo IS NULL And Customer_id = 2
```

```
--Update MobileNo
```

```
UPDATE CUSTOMER
SET     MobileNo = 91244
WHERE MobileNo IS NULL And Customer_id = 2
```

```
--See the changes
```

```
SELECT * FROM CUSTOMER
WHERE Customer_id = 2
```

```
--Setting up MobileNo to NULL again to include it in delete NULLs
```

```
UPDATE CUSTOMER
SET     MobileNo = NULL
WHERE  Customer_id = 2
```

```
SELECT * FROM CUSTOMER
WHERE MobileNo IS NULL And Customer_id = 2
```

Deleting NULLs

When updating NULLs, it is recommended that you update the MobileNo for Customer Id = 2 to NULL again.

Use the following to delete rows having NULL values:

```
Delete from Customer
where MobileNo IS NULL
```

Null Values and Joins

When there are null values in the columns of the tables being joined, the null values do not match each other. The presence of null values in a column from one of the tables being joined can be returned only by using an outer join (unless the WHERE clause excludes null values).

Here are two tables that each have NULL in the column that will participate in the join:

table1		table2	
a	b	c	d
1	one	NULL	two
NULL	three	4	four
4	join4		

A join that compares the values in column a against column c does not get a match on the columns that have values of NULL:

```
SELECT *
FROM table1 t1 JOIN table2 t2
    ON t1.a = t2.c
ORDER BY t1.a
```

Only one row with 4 in column a and c is returned:

a	b	c	d
4	join4	4	four

(1 row(s) affected)

Null values returned from a base table are also difficult to distinguish from the null values returned from an outer join. For example, the following SELECT statement does a left outer join on these two tables:

```
SELECT *
FROM table1 t1 LEFT OUTER JOIN table2 t2
    ON t1.a = t2.c
ORDER BY t1.a
```

Here is the result set.

a	b	c	d
NULL	three	NULL	NULL
1	one	NULL	NULL
4	join4	4	four

(3 row(s) affected)

The results do not make it easy to distinguish a NULL in the data from a NULL that represents a failure to join. When null values are present in data being joined, it is usually preferable to omit them from the results by using a regular join.

NOT IN clause with NULL values

The **NOT IN** clause returns rows from the outer table which do not exist in the inner table used in the subquery. In this tutorial, we will examine using the NOT IN clause with null values.

Example

```
SELECT * FROM OuterTable
WHERE PK_Id NOT IN (SELECT FK_id from InnerTable);
```

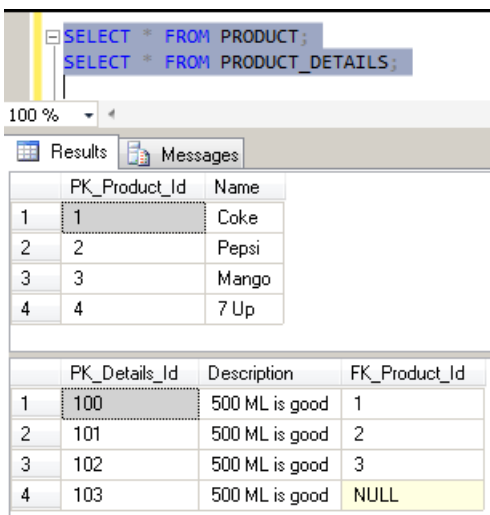
Example of the NOT IN Clause with NULL Values in InnerTable.

```
CREATE TABLE PRODUCT ( PK_Product_Id INT PRIMARY KEY , Name VARCHAR( 255) );

INSERT INTO PRODUCT VALUES ( 1, 'Coke' ), (2, 'Pepsi' ), (3 , 'Mango' ), (4 , '7 Up' );

CREATE TABLE PRODUCT_DETAILS
(
PK_Details_Id INT PRIMARY KEY ,
[Description] VARCHAR( 500),
FK_Product_Id INT FOREIGN KEY REFERENCES PRODUCT (PK_Product_Id )
);

INSERT INTO PRODUCT_DETAILS VALUES ( 100, '500 ML is good' ,1);
INSERT INTO PRODUCT_DETAILS VALUES ( 101, '500 ML is good' ,2);
INSERT INTO PRODUCT_DETAILS VALUES ( 102, '500 ML is good' ,3);
INSERT INTO PRODUCT_DETAILS VALUES ( 103, '500 ML is good' ,NULL);
```



	PK_Product_Id	Name
1	1	Coke
2	2	Pepsi
3	3	Mango
4	4	7 Up

	PK_Details_Id	Description	FK_Product_Id
1	100	500 ML is good	1
2	101	500 ML is good	2
3	102	500 ML is good	3
4	103	500 ML is good	NULL

As shown in the image above, the PRODUCT_DETAILS table contains one **NULL value**.

Now, our aim is **to find out the names of the products from the Product table whose details are not available in the Product_Details table**. Ideally, it should return Product 4 from the Product table since the details of Product 4 do not exist in the Product_Details table.

We might think this can be easily achieved using the NOT IN predicate by writing the following query.

```
SELECT * FROM PRODUCT
```

```
WHERE PK_Product_Id NOT IN (SELECT Fk_Product_Id FROM PRODUCT_DETAILS);
```

Result of query is empty

The query above does not return anything although syntactically it is correct. Because of the existence of a NULL value in the Product_Details table, it fails to return the expected results.

SQL Server uses the **Three-Valued logic** concept here.

As we are aware, a NULL value does exist in the Product_Details table in the fk_product_id column. A **NULL value is an unknown or missing value**.

SQL Server converts the NOT IN clause using three-value logic and evaluates it in the following manner.

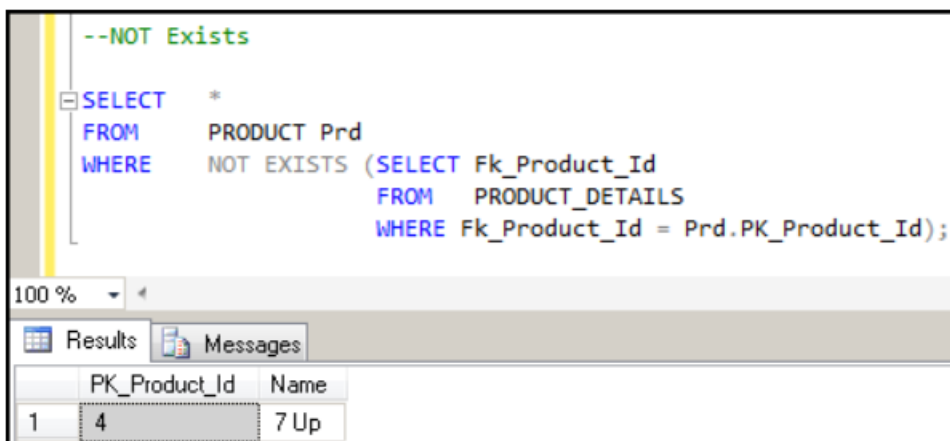
```
NOT IN (SELECT 1 OR 2 OR 3 OR NULL)
NOT IN (Fk_product_id = 1 OR Fk_product_id = 2 OR Fk_product_id = 3 OR Fk_product_id = NULL)
NOT IN (Fk_product_id = 1 OR Fk_product_id = 2 OR Fk_product_id = 3 OR UNKNOWN )
NOT IN (TRUE OR TRUE OR TRUE OR UNKNOWN )
NOT IN (TRUE OR TRUE OR UNKNOWN)
NOT IN (TRUE OR UNKNOWN)
NOT IN (UNKNOWN )
```

As the **final result is evaluated as UNKNOWN**, the NOT IN query does not return any result because of the existence of a NULL value.

The solution is to make the NOT IN queries work with the existence of NULL values and use two-valued logic, only TRUE or FALSE.

```
--NOT IN WITH IS NOT NULL Filter
SELECT * FROM PRODUCT
WHERE PK_Product_Id
NOT IN (SELECT Fk_Product_Id FROM PRODUCT_DETAILS WHERE Fk_Product_Id IS NOT NULL);

--NOT Exists
SELECT * FROM PRODUCT Prd
WHERE NOT EXISTS
(SELECT Fk_Product_Id FROM PRODUCT_DETAILS WHERE Fk_Product_Id = Prd.PK_Product_Id);
```



```
--NOT IN WITH IS NOT NULL Filter
SELECT *
FROM PRODUCT
WHERE PK_Product_Id
      NOT IN (SELECT Fk_Product_Id
              FROM PRODUCT_DETAILS WHERE Fk_Product_Id IS NOT NULL);
```

100 %

Results Messages

	PK_Product_Id	Name
1	4	7 Up

NOT EXISTS also gives us the right results because it uses two-valued Boolean logic, only TRUE or False, to filter out the rows.

--Using Left Join

```
SELECT *
FROM PRODUCT Prd LEFT JOIN PRODUCT_DETAILS PrdDetails
ON Prd. PK_Product_Id = PrdDetails .FK_Product_Id
WHERE PrdDetails. FK_Product_Id IS NULL
```

Using Left join, we can also retrieve records which exist in the Product table but do not exist in the Product_Details page when we do a join based on the primary key (pk_product_id) and the foreign key (fk_product_id).

NULL and Aggregation function (Like Count, MAX etc)

Be extra careful when you count null column, use count(1) or count(*) instead. SUM(), MIN(), MAX ignores null Values.

```
SELECT [Name] ,[MKT] ,[Cost]
FROM [SQL_Training].[dbo].[Table_D]
```

100 %

Results Messages

	Name	MKT	Cost
1	Siraj	SF	1
2	Dave	SF	2
3	John	SF	NULL
4	Ray	VG	0
5	Phil	LA	9
6	Awais	La	3
7	Mike	NULL	100

```
SELECT [MKT],count(1) as col0,count(*) as col1, count(mkt) as col2, count(cost) as col3, sum(cost) as col4,
min(cost) as col5, max(cost) as col6
FROM [SQL_Training].[dbo].[Table_D]
group by mkt
```

100 %

Results Messages

	MKT	col0	col1	col2	col3	col4	col5	col6
1	NULL	1	1	0	1	100	100	100
2	LA	2	2	2	2	12	3	9
3	SF	3	3	3	2	3	1	2
4	VG	1	1	1	1	0	0	0

SQL Insert Statements

1. Insert using a VALUES clause to specify the data values for one row. For example:

```
INSERT INTO MyTable (PriKey, Description) VALUES (123, 'A description of part 123.');
```
2. Insert using a SELECT subquery to specify the data values for one or more rows, such as:

```
INSERT INTO MyTable (PriKey, Description)
SELECT ForeignKey, Description FROM SomeView;
```
3. Insert using SELECT with INTO

```
SELECT LastName, FirstName, Phone
INTO dbo.PhoneList492 FROM dbo.Customers WHERE Phone LIKE '492%'
```

SQL Update Statements

```
UPDATE Person.Address
SET PostalCode = '98000'
WHERE City = 'Bothell';
```

SQL CREATE TABLE Statement

```
CREATE TABLE employees
( employee_id INT NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  first_name VARCHAR(50),
  salary MONEY
);
```

This SQL Server CREATE TABLE example creates a table called *employees* which has 4 columns.

- The first column is called *employee* which is created as an INT datatype and can not contain NULL values.
- The second column is called *last_name* which is a VARCHAR datatype (50 maximum characters in length) and also can not contain NULL values.
- The third column is called *first_name* which is a VARCHAR datatype but can contain NULL values.
- The fourth column is called *salary* which is a MONEY datatype which can contain NULL values.

Now the only problem with this CREATE TABLE statement is that you have not defined a primary key for the table in SQL Server. We could modify this CREATE TABLE statement and define the *employee_id* as the primary key as follows:

```
CREATE TABLE employees
( employee_id INT PRIMARY KEY,
  last_name VARCHAR(50) NOT NULL,
  first_name VARCHAR(50),
  salary MONEY
);
```

Composite PK

```
CREATE TABLE survey (
  surveyID int primary key,
  student_name nvarchar(20),
  year int, )
```

```
CREATE TABLE fields (
  fieldID int primary key,
  field_name nvarchar(20),
  year int )
```



```
CREATE TABLE data(  
    surveyID int,  
    fieldID int,  
    value float,  
    primary key (surveyID, fieldID),  
    foreign key(studentID),  
    foreign key(fieldID) )
```

SQL Delete Statement

The DELETE Statement is used to delete rows from a table.

To delete an employee with id 100 from the employee table, the sql delete query would be like,

```
DELETE FROM employee WHERE id = 100;
```

To delete all the rows from the employee table, the query would be like,

```
DELETE FROM employee;
```

SQL TRUNCATE Statement

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

To delete all the rows from employee table, the query would be like,

```
TRUNCATE TABLE employee;
```

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.

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

SQL DROP Statement:

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.

To drop the table employee, the query would be like

```
DROP TABLE employee;
```

Difference between DROP and TRUNCATE Statement:

If a table is dropped, all the relationships with other tables will no longer be valid, the integrity constraints will be dropped, grant or access privileges on the table will also be dropped, if you want use the table again it has to be recreated

with the integrity constraints, access privileges and the relationships with other tables should be established again. But, if a table is truncated, the table structure remains the same, therefore any of the above problems will not exist.

Changing Data by Using a Cursor

The ADO, OLE DB, and ODBC APIs support updating the current row on which the application is positioned in a result set. The following steps describe the fundamental process:

1. Bind the result set columns to program variables.
2. Execute the query.
3. Call API functions or methods to position the application on a row within the result set.
4. Fill the bound program variables with the new data values for any columns to be updated.
5. Call one of these functions or methods to insert the row:
 - o In ADO, call the Update method of the Recordset object.
 - o In OLE DB, call the SetData method of the IRowsetChange interface.
 - o In ODBC, call the SQLSetPos function with the SQL_UPDATE option.

When you use a Transact-SQL server cursor, you can update the current row by using an UPDATE statement that includes a WHERE CURRENT OF clause. Changes made with this clause affect only the row on which the cursor is positioned. When a cursor is based on a join, only the table_name specified in the UPDATE statement is modified. Other tables participating in the cursor are not affected.

```
DECLARE complex_cursor CURSOR FOR
  SELECT a.BusinessEntityID
  FROM HumanResources.EmployeePayHistory AS a
  WHERE RateChangeDate <>
        (SELECT MAX(RateChangeDate)
         FROM HumanResources.EmployeePayHistory AS b
         WHERE a.BusinessEntityID = b.BusinessEntityID) ;

OPEN complex_cursor;
FETCH FROM complex_cursor;

UPDATE HumanResources.EmployeePayHistory
SET PayFrequency = 2
WHERE CURRENT OF complex_cursor;

CLOSE complex_cursor;
DEALLOCATE complex_cursor;
```

Normalization

If a database design is not perfect, it may contain anomalies, which are like a bad dream for any database administrator. Managing a database with anomalies is next to impossible.

- **Update anomalies** – If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.
- **Deletion anomalies** – We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.
- **Insert anomalies** – We tried to insert data in a record that does not exist at all.

Normalization is a method to remove all these anomalies and bring the database to a consistent state.

First Normal Form

First Normal Form is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains. The values in an atomic domain are indivisible units.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

We re-arrange the relation (table) as below, to convert it to First Normal Form.

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

Each attribute must contain only a single value from its pre-defined domain.

Second Normal Form

Before we learn about the second normal form, we need to understand the following –

- **Prime attribute** – An attribute, which is a part of the prime-key, is known as a prime attribute.
- **Non-prime attribute** – An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.

If we follow second normal form, then every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if $X \rightarrow A$ holds, then there should not be any proper subset Y of X , for which $Y \rightarrow A$ also holds true.

Student_Project



We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called **partial dependency**, which is not allowed in Second Normal Form.

Student



Project



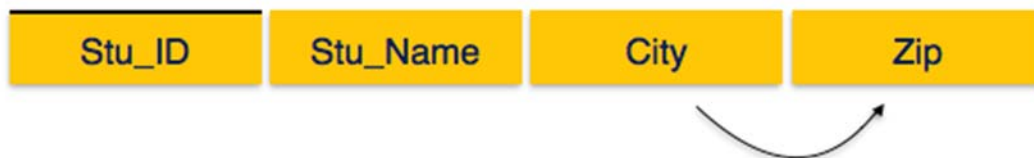
We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

Third Normal Form

For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy –

- No non-prime attribute is transitively dependent on prime key attribute.
- For any non-trivial functional dependency, $X \rightarrow A$, then either –
 - X is a superkey or,
 - A is prime attribute.

Student_Detail



We find that in the above Student_detail relation, Stu_ID is the key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, $Stu_ID \rightarrow Zip \rightarrow City$, so there exists **transitive dependency**.

To bring this relation into third normal form, we break the relation into two relations as follows –

Student_Detail

Stu_ID	Stu_Name	Zip
--------	----------	-----

ZipCodes

Zip	City
-----	------

Boyce-Codd Normal Form

Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms. BCNF states that –

- For any non-trivial functional dependency, $X \rightarrow A$, X must be a super-key.

In the above image, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So,

$\text{Stu_ID} \rightarrow \text{Stu_Name}, \text{Zip}$
and
 $\text{Zip} \rightarrow \text{City}$

Which confirms that both the relations are in BCNF.

Types of SQL Keys

We have following types of keys in SQL which are used to fetch records from tables and to make relationship among tables or views.

1. Super Key

Super key is a set of one or more than one keys that can be used to identify a record uniquely in a table. **Example :** Primary key, Unique key, Alternate key are subset of Super Keys.

2. Candidate Key

A Candidate Key is a set of one or more fields/columns that can identify a record uniquely in a table. There can be multiple Candidate Keys in one table. Each Candidate Key can work as Primary Key.

Example: In below diagram ID, RollNo and EnrollNo are Candidate Keys since all these three fields can be work as Primary Key.

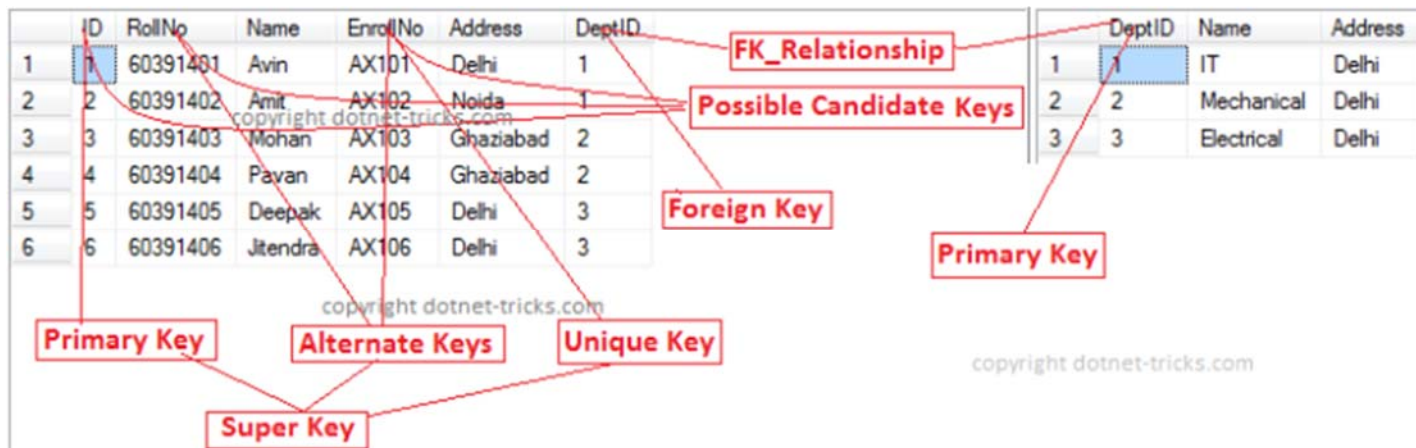
3. Primary Key

Primary key is a set of one or more fields/columns of a table that uniquely identify a record in database table. It cannot accept null, duplicate values. Only one Candidate Key can be Primary Key.

4. Alternate key

A Alternate key is a key that can be work as a primary key. Basically it is a candidate key that currently is not primary key.

Example: In below diagram RollNo and EnrollNo becomes Alternate Keys when we define ID as Primary Key.



5. Composite/Compound Key

Composite Key is a combination of more than one fields/columns of a table. It can be a Candidate key, Primary key.

6. Unique Key

Uniquekey is a set of one or more fields/columns of a table that uniquely identify a record in database table. It is like Primary key but it can accept only one null value and it can not have duplicate values. For more help refer the article [Difference between primary key and unique key](#).

7. Foreign Key

Foreign Key is a field in database table that is Primary key in another table. It can accept multiple null, duplicate values. For more help refer the article [Difference between primary key and foreign key](#).

Example : We can have a DeptID column in the Employee table which is pointing to DeptID column in a department table where it a primary key.

Defining Keys in SQL Server

```
--Department Table
CREATE TABLE Department
(
    DeptID int PRIMARY KEY, --primary key
    Name varchar (50) NOT NULL,
    Address varchar (200) NOT NULL
)

--Student Table
CREATE TABLE Student
(
    ID int PRIMARY KEY, --primary key
    RollNo varchar(10) NOT NULL,
    Name varchar(50) NOT NULL,
    EnrollNo varchar(50) UNIQUE, --unique key
    Address varchar(200) NOT NULL,
    DeptID int FOREIGN KEY REFERENCES Department(DeptID) --foreign key
)
```

Note

Practically in database, we have only three types of keys Primary Key, Unique Key and Foreign Key. Other types of keys are only concepts of RDBMS which you should know.

Stored Procedure and Functions

Stored Procedure

```
create PROCEDURE getVendor @code NVARCHAR(50)
AS
SELECT [VENDOR_NAME], [VENDOR_CODE]
FROM [TMO].[dbo].[VENDOR]
where [VENDOR_CODE] = @code
```

```
EXEC getVendor @code='ee'
```

User Defined function

```
CREATE FUNCTION Vfunction ( @code NVARCHAR(50) )
RETURNS NVARCHAR(50)
AS
BEGIN
    DECLARE @ven NVARCHAR(50)
    SELECT @ven = [VENDOR_NAME] FROM [TMO].[dbo].[VENDOR]
    WHERE [VENDOR_CODE] = @code
    RETURN(@ven)
END
```

```
SELECT [VENDOR_NAME] , [VENDOR_CODE], dbo.Vfunction ([VENDOR_CODE]) FROM [TMO].[dbo].[VENDOR]
```

Difference between Stored Procedure and Function in SQL Server

Stored Procedures are pre-compile objects which are compiled for first time and its compiled format is saved which executes (compiled code) whenever it is called. But Function is compiled and executed every time when it is called.

Basic Difference

1. Function must return a value but in Stored Procedure it is optional (Procedure can return zero or n values).
2. Functions can have only input parameters for it whereas Procedures can have input/output parameters.
3. Functions can be called from Procedure whereas Procedures cannot be called from Function.

Advance Difference

1. Procedure allows SELECT as well as DML(INSERT/UPDATE/DELETE) statement in it whereas Function allows only SELECT statement in it.
2. Procedures cannot be utilized in a SELECT statement whereas Function can be embedded in a SELECT statement.
3. Stored Procedures cannot be used in the SQL statements anywhere in the WHERE/HAVING/SELECT section whereas Function can be.
4. Functions that return tables can be treated as another rowset. This can be used in JOINS with other tables.
5. Inline Function can be thought of as views that take parameters and can be used in JOINS and other Rowset operations.
6. Exception can be handled by try-catch block in a Procedure whereas try-catch block cannot be used in a Function.
7. We can go for Transaction Management in Procedure whereas we can't go in Function.

Different Types of SQL Server Stored Procedures

Stored procedure is a precompiled set of one or more SQL statements that is stored on Sql Server. Benifit of Stored Procedures is that they are executed on the server side and perform a set of actions, before returning the results to the client side. This allows a set of actions to be executed with minimum time and also reduce the network traffic. Hence stored procedure improve performance to execute sql statements. For more about stored procedure refer the article [CRUD Operations using Stored Procedures](#).

Stored procedure can accepts input and output parameters. Stored procedure can returns multiple values using output parameters. Using stored procedure, we can Select,Insert,Update,Delete data in database.

Types of Stored Procedure

1. System Defined Stored Procedure

These stored procedure are already defined in Sql Server. These are physically stored in hidden Sql Server Resource Database and logically appear in the sys schema of each user defined and system defined database. These procedure starts with the sp_ prefix. Hence we don't use this prefix when naming user-defined procedures. Here is a list of some useful system defined procedure.

SP	Description
sp_rename	It is used to rename an database object like stored procedure,views,table etc.
sp_changewowner	It is used to change the owner of an database object.
sp_help	It provides details on any database object.
sp_helpdb	It provide the details of the databases defined in the Sql Server.
sp_helptext	It provides the text of a stored procedure reside in Sql Server
sp_depends	It provide the details of all database objects that depends on the specific database object.

2. Extended Procedure

Extended procedures provide an interface to external programs for various maintenance activities. These extended procedures starts with the xp_ prefix and stored in Master database. Basically these are used to call programs that reside on the server automatically from a stored procedure or a trigger run by the server.

Example Below statements are used to log an event in the NT event log of the server without raising any error on the client application.

```
declare @logmsg varchar(100)
set @logmsg = suser_sname() + ': Tried to access the dotnet system.'
exec xp_logevent 50005, @logmsg
print @logmsg
```

Example The below procedure will display details about the BUILTIN\Administrators Windows group.

```
EXEC xp_logininfo 'BUILTIN\Administrators'
```

3. User Defined Stored Procedure

These procedures are created by user for own actions. These can be created in all system databases except the Resource database or in a user-defined database.

4. CLR Stored Procedure

CLR stored procedure are special type of procedure that are based on the CLR (Common Language Runtime) in .net framework. CLR integration of procedure was introduced with SQL Server 2008 and allow for procedure to be coded in one of .NET languages like C#, Visual Basic and F#. I will discuss CLR stored procedure later.

Different Types of SQL Server 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 Function

1. System Defined Function

These functions are defined by Sql Server for different purpose. We have two types of system defined function in Sql Server

a. Scalar Function

Scalar functions operates 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.

b. Aggregate Function

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

Aggregate Function	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.

2. User Defined Function

These functions are created by user in system database or in user defined database. We have three types of user defined functions.

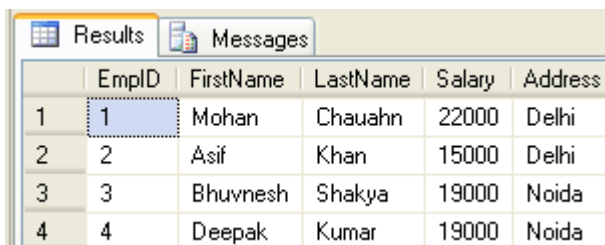
a. Scalar Function

User defined scalar function also returns single value as a result of actions performed by function. We return any datatype value from function.

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

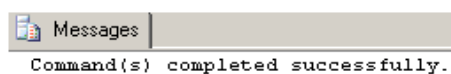
Insert into Employee(EmpID,FirstName,LastName,Salary,Address)
Values(1,'Mohan','Chauhan',22000,'Delhi');
Insert into Employee(EmpID,FirstName,LastName,Salary,Address)
Values(2,'Asif','Khan',15000,'Delhi');
Insert into Employee(EmpID,FirstName,LastName,Salary,Address)
Values(3,'Bhuvnesh','Shakya',19000,'Noida');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

--See created table
Select * from Employee
```



	EmpID	FirstName	LastName	Salary	Address
1	1	Mohan	Chauhan	22000	Delhi
2	2	Asif	Khan	15000	Delhi
3	3	Bhuvnesh	Shakya	19000	Noida
4	4	Deepak	Kumar	19000	Noida

```
--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
```

Results			Messages
	Name	Salary	
1	Mohan Chauahn	22000	
2	Asif Khan	15000	
3	Bhuvnesh Shakya	19000	
4	Deepak Kumar	19000	

b. 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 fnGetEmployee()
returns Table
As      return (Select * from Employee)
```

Messages

Command(s) completed successfully.

```
Select * from fnGetEmployee()
```

Results

Messages

	EmpID	FirstName	LastName	Salary	Address
1	1	Mohan	Chauahn	22000	Delhi
2	2	Asif	Khan	15000	Delhi
3	3	Bhuvnesh	Shakya	19000	Noida
4	4	Deepak	Kumar	19000	Noida

c. 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 statements.

```
--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
```

Messages

Command(s) completed successfully.

```
--Now call the above created function
Select * from fnGetMulEmployee()
```

	EmpID	FirstName	Salary
1	1	Mohan	25000
2	2	Asif	15000
3	3	Bhuvnesh	19000
4	4	Deepak	19000

```
--Now see the original table. This is not affected by above function
update command
Select * from Employee
```

	EmpID	FirstName	LastName	Salary	Address
1	1	Mohan	Chauahn	22000	Delhi
2	2	Asif	Khan	15000	Delhi
3	3	Bhuvnesh	Shakya	19000	Noida
4	4	Deepak	Kumar	19000	Noida

Note (SP vs Function)

1. Unlike Stored Procedure, Function returns only single value.
2. Unlike Stored Procedure, Function accepts only input parameters.
3. Unlike Stored Procedure, Function is not used to Insert, Update, Delete data in database table(s).
4. Like Stored Procedure, Function can be nested up to 32 level.
5. UDF can have upto 1023 input parameters while a SP can have up to 2100 input parameters.
6. User Defined Function can't returns XML Data Type.
7. User Defined Function doesn't support Exception handling.
8. User Defined Function can call only Extended Stored Procedure.
9. User Defined Function doesn't support set options like set ROWCOUNT etc.

Views

Database views allow you to create "virtual tables" that are generated on the fly when they are accessed. A view is stored on the database server as an SQL statement that pulls data from one or more tables and (optionally) performs transformations on that data. Users may then query the view just as they would any real database table. Views are often used to alleviate security concerns by providing users with access to a certain view of a database table without providing access to the underlying table itself.

S.No.	View	Stored Procedure
1	Does not accepts parameters	Accept parameters
2	Can be used as a building block in large query.	Can not be used as a building block in large query.
3	Can contain only one single Select query.	Can contain several statement like if, else, etc.
4	Can not perform modification to any table.	Can perform modification to one or several tables.
5	Can be used (sometimes) as the target for Insert, update, delete queries.	Can not be used as the target for Insert, up delete queries.

SQL Data Type

Data type	Description	Storage
char(n)	Fixed width character string. Maximum 8,000 characters	Defined width
varchar(n)	Variable width character string. Maximum 8,000 characters	2 bytes + number of chars
varchar(max)	Variable width character string. Maximum 1,073,741,824 characters	2 bytes + number of chars
text	Variable width character string. Maximum 2GB of text data	4 bytes + number of chars
nchar	Fixed width Unicode string. Maximum 4,000 characters	Defined width x 2
nvarchar	Variable width Unicode string. Maximum 4,000 characters	
nvarchar(max)	Variable width Unicode string. Maximum 536,870,912 characters	
ntext	Variable width Unicode string. Maximum 2GB of text data	
bit	Allows 0, 1, or NULL	
binary(n)	Fixed width binary string. Maximum 8,000 bytes	
varbinary	Variable width binary string. Maximum 8,000 bytes	
varbinary(max)	Variable width binary string. Maximum 2GB	
image	Variable width binary string. Maximum 2GB	
tinyint	Allows whole numbers from 0 to 255	1 byte
smallint	Allows whole numbers between -32,768 and 32,767	2 bytes
int	Allows whole numbers between -2,147,483,648 and 2,147,483,647	4 bytes
bigint	Allows whole numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807	8 bytes
decimal(p,s)	Fixed precision and scale numbers.	5-17 bytes
	Allows numbers from $-10^{38} + 1$ to $10^{38} - 1$.	
	The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18.	
numeric(p,s)	The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0	5-17 bytes
	Fixed precision and scale numbers.	
	Allows numbers from $-10^{38} + 1$ to $10^{38} - 1$.	
numeric(p,s)	The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18.	5-17 bytes
	The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0	
smallmoney	Monetary data from -214,748.3648 to 214,748.3647	4 bytes
money	Monetary data from -922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
float(n)	Floating precision number data from $-1.79E + 308$ to $1.79E + 308$.	4 or 8 bytes

	The n parameter indicates whether the field should hold 4 or 8 bytes. float(24) holds a 4-byte field and float(53) holds an 8-byte field. Default value of n is 53.	
real	Floating precision number data from -3.40E + 38 to 3.40E + 38	4 bytes
datetime	From January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds	8 bytes
datetime2	From January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds	6-8 bytes
smalldatetime	From January 1, 1900 to June 6, 2079 with an accuracy of 1 minute	4 bytes
date	Store a date only. From January 1, 0001 to December 31, 9999	3 bytes
time	Store a time only to an accuracy of 100 nanoseconds	3-5 bytes
datetimeoffset	The same as datetime2 with the addition of a time zone offset	8-10 bytes
timestamp	Stores a unique number that gets updated every time a row gets created or modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable	
sql_variant	Stores up to 8,000 bytes of data of various data types, except text, ntext, and timestamp	
uniqueidentifier	Stores a globally unique identifier (GUID)	
xml	Stores XML formatted data. Maximum 2GB	
cursor	Stores a reference to a cursor used for database operations	
table	Stores a result-set for later processing	

What is the difference between char, nchar, varchar, and nvarchar in SQL Server?

- **nchar** and **nvarchar** can store **Unicode** characters.
- **char** and **varchar** **cannot** store **Unicode** characters.
- **char** and **nchar** are **fixed-length** which will **reserve storage space** for number of characters you specify even if you don't use up all that space.
- **varchar** and **nvarchar** are **variable-length** which will only use up spaces for the characters you store. It **will not** **reserve storage like char or nchar**.

nchar and nvarchar will take up twice as much storage space, so it may be wise to use them only if you need *Unicode* support

The deprecated types text and ntext correspond to the new types varchar(max) and nvarchar(max) respectively.

VARCHAR(MAX): is a BLOB variable length non-Unicode. Check difference between **in-row** & **BLOB** string.

ASCII Vs Unicode

1. ASCII uses an 8-bit encoding while Unicode uses a variable bit encoding.
2. Unicode is standardized while ASCII isn't.
3. Unicode represents most written languages in the world while ASCII does not.
4. ASCII has its equivalent within Unicode.

What is an index?

An index is a structure within SQL that is used to quickly locate specific rows within a table. It can be useful to imagine an index at the back of a textbook when thinking about SQL indexes. They both serve the same purpose – to find specific information quickly.

General Structure

An index is defined on one or more columns, called key columns. The key columns (also referred to as the index key) can be likened to the terms listed in a book index. They are the values that the index will be used to search for. As with the index found at the back of a text book (see figure 1), the index is sorted by the key columns.

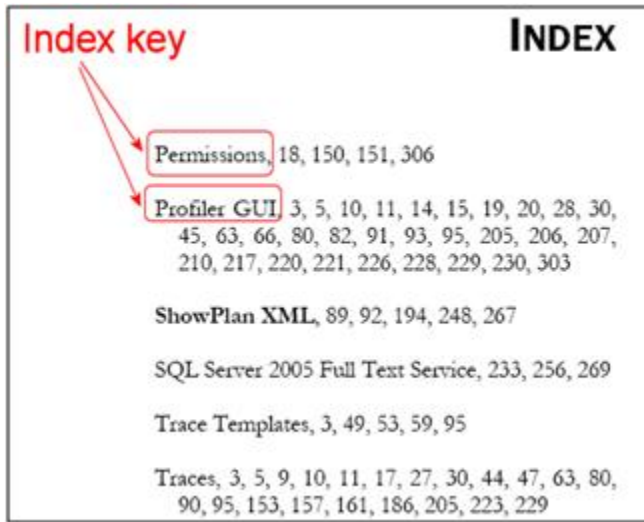


Figure 1: Book index. Image copyright Simple Talk publishing.

If an index is created with more than one key column, it is known as a composite index.

The general structure of an index is that of a balanced tree (b-tree). The index will have a single root page, zero or more intermediate levels and then a leaf level. A page is an 8 kilobyte chunk of the data file, with a header and footer and is identified by a combination of File ID and Page number.

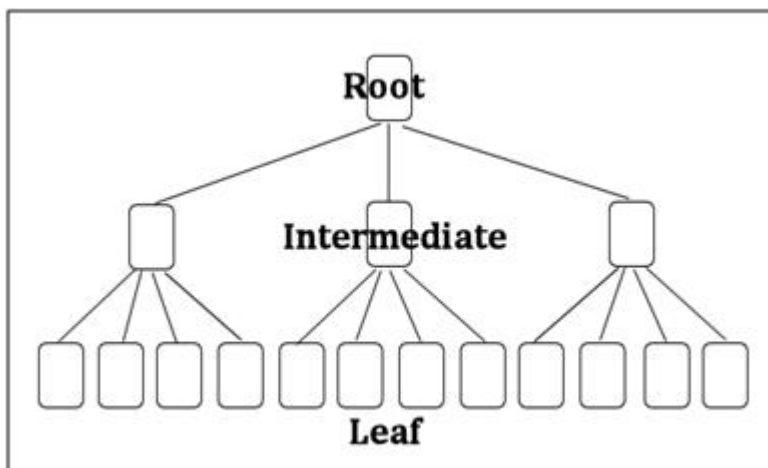


Figure 2: Index Structure

Note: Commonly the root page is shown at the top of the tree diagram and the leaf pages at the bottom. Think of it as an inverted tree.

In the leaf level, there's one entry for each row in the index¹. The entries in the index are ordered logically² in the order of the index key.

The non-leaf levels of the index contain one row per page of the level below, referencing the lowest index key value on each page. If all of those rows fit onto a single page, then that page is considered the root and the index is only two levels deep. If all of those rows will not fit on a single page, then one (or more) intermediate levels are added to the index.

The number of levels in an index is referred to as the depth of the index. This is an important consideration for evaluating the efficiency of the index. The index illustrated in figure 2 has a depth of 3.

- (1) With the exception of SQL 2008's filtered indexes, an index will have the same number of rows at the leaf level as the table.
- (2) I'm using the phrase 'logically ordered' because the index does not necessarily define the physical storage of the rows. The rows are stored in a way that SQL can retrieve them ordered.

Clustered and nonclustered

There are two main types of indexes in SQL Server, the clustered index and the nonclustered index

Clustered indexes define the logical order of the table. The leaf level of the clustered index has the actual data pages of the table. Because of this there can only be one clustered index per table. A table that does not have a clustered index is referred to as a heap.

Nonclustered indexes are separate from the table. The leaf level of a nonclustered index has a pointer as part of each index row. That pointer is either the clustered index key in the cases where the base table has a clustered index or the RID (Row Identifier) in the cases where the table is a heap. The RID is an 8-byte structure comprised of File ID, Page Number and Slot Index and will uniquely identify a row in the underlying heap. Either way, the each row of a nonclustered index has a reference to the complete data row.

Considerations for creating indexes

I'll be going into more detail on considerations for indexes in the next two parts, but in general:

- Clustered index should be narrow, because the clustering key is part of all nonclustered indexes.
- Composite nonclustered indexes are generally more useful than single column indexes, unless all queries against the table filter on one column at a time.
- Indexes should be no wider than they have to be. Too many columns wastes space and increases the amount of places that data must be changed when an insert/update/delete occurs.
- If an index is unique, specify that it is unique. The optimiser can sometimes use that information to generate more optimal execution plans.
- Be careful of creating lots of indexes on frequently modified tables as it can slow down data modifications.

<http://www.sqlservercentral.com/articles/Indexing/68439/>

Dealing with float point / Casting / Round off

Cast() Function

The Cast() function is used to convert a data type variable or data from one data type to another data type. The Cast() function provides a data type to a dynamic parameter (?) or a NULL value.

CAST ([Expression] AS Datatype)

```
DECLARE @A varchar(2)
DECLARE @B varchar(2)
DECLARE @C varchar(2)
set @A=25
set @B=15
set @C=33
Select CAST(@A as int) + CAST(@B as int) + CAST (@C as int) as Result
```

CONVERT() Function (SQL Server only)

When you convert expressions from one type to another, in many cases there will be a need within a stored procedure or other routine to convert data from a datetime type to a varchar type. The Convert function is used for such things. The CONVERT() function can be used to display date/time data in various formats.

CONVERT(data_type(length), expression, style)

```
CONVERT(VARCHAR(19),GETDATE())
CONVERT(VARCHAR(10),GETDATE(),10)
CONVERT(VARCHAR(10),GETDATE(),110)
CONVERT(VARCHAR(11),GETDATE(),6)
CONVERT(VARCHAR(11),GETDATE(),106)
CONVERT(VARCHAR(24),GETDATE(),113)
```

The result would look something like this:

```
Nov 04 2014 11:45 PM
11-04-14
11-04-2014
04 Nov 14
04 Nov 2014
04 Nov 2014 11:45:34:243
```

Anything you can do with CAST you can do with CONVERT.

CAST is part of the ANSI-SQL specification; whereas, CONVERT is not. In fact, CONVERT is SQL implementation specific. CONVERT differences lie in that it accepts an optional style parameter which is used for formatting.

TRY_CAST

takes the value passed to it and tries to convert it to the specified *data_type*. If the cast succeeds, **TRY_CAST** returns the value as the specified *data_type*; if an error occurs, null is returned. However if you request a conversion that is explicitly not permitted, then **TRY_CAST** fails with an error.

```
SELECT
CASE WHEN TRY_CAST('test' AS float) IS NULL
THEN 'Cast failed'
ELSE 'Cast succeeded'
END AS Result;
```

TRY_CONVERT

takes the value passed to it and tries to convert it to the specified *data_type*. If the cast succeeds, **TRY_CONVERT** returns the value as the specified *data_type*; if an error occurs, null is returned. However if you request a conversion that is explicitly not permitted, then **TRY_CONVERT** fails with an error.

```
SELECT
CASE WHEN TRY_CONVERT(float, 'test') IS NULL
THEN 'Cast failed'
ELSE 'Cast succeeded'
END AS Result;
```

TRY_PARSE

Returns the result of an expression, translated to the requested data type, or null if the cast fails in SQL Server. Use TRY_PARSE only for converting from string to date/time and number types.

```
TRY_PARSE ( string_value AS data_type [ USING culture ] )
```

Example

```
SELECT TRY_PARSE('Jabberwokkie' AS datetime2 USING 'en-US') AS Result;
```

Result

```
-----
NULL
```

```
SELECT
CASE WHEN TRY_PARSE('Aragorn' AS decimal USING 'sr-Latn-CS') IS NULL
THEN 'True'
ELSE 'False'
END
AS Result;
```

Result

```
-----
True
```

```
SET LANGUAGE English;
SELECT IIF(TRY_PARSE('01/01/2011' AS datetime2) IS NULL, 'True', 'False') AS Result;
```

Result

```
-----
False
```

Round ()

ROUND(*number*, *decimal_places*, *operation*)

Parameter	Description
<i>number</i>	Required. The number to round
<i>decimal_places</i>	Required. The number of decimal places to round to
<i>operation</i>	Optional. Can be either 0 or any other numeric value. When 0, ROUND() will round the result to the number of <i>decimal_places</i> . When another value than 0, ROUND() will truncate the result to the number of <i>decimal_places</i> . Default value is 0

```
SELECT ROUND(235.415, 0) AS RoundValue  
--Result 235.000
```

```
SELECT ROUND(235.415, 1) AS RoundValue;  
--Result 235.400
```

```
SELECT ROUND(235.415, 2) AS RoundValue;  
--Result 235.420
```

Floor ()

FLOOR(*number*)

The FLOOR() function returns the **largest integer** value that is **equal to** or **less than** the specified number.

```
SELECT FLOOR(25.90) AS FloorValue;  
--Result 25
```

```
SELECT FLOOR(25.20) AS FloorValue;  
--Result 25
```

CEILING ()

CEILING (*number*)

The CEILING() function returns the **smallest integer** value that is **greater than** or **equal to** the specified number.

```
SELECT CEILING(25.90) AS FloorValue;  
--Result 26
```

```
SELECT CEILING(25.20) AS FloorValue;  
--Result 26
```

```
SELECT CEILING(-25.20) AS FloorValue;  
--Result -25
```

Avoiding division by zero with NULLIF and COALESCE

NULLIF() Function

`NULLIF(expr1, expr2)`

The `NULLIF()` function compares two expressions.

If `expr1` and `expr2` are equal, the `NULLIF()` function returns `NULL`. Otherwise, it returns `expr1`.

COALESCE() Function

`COALESCE(expr1, expr2, ..., expr_n)`

The `COALESCE()` function returns the first non-null expression in a list.

Example

```
select 3/0
```

Result

Msg 8134, Level 16, State 1, Line 1

Divide by zero error encountered.

```
select 3/null
```

-- Any number divided by `NULL` gives `NULL`, and no error is generated.

-- result will be null

Option 1 (using case)

```
Select Case when divisor=0 then null
```

```
Else dividend / divisor
```

```
End
```

-- result will be null

Option 2 (using nullif)

```
Select dividend / nullif(divisor, 0)
```

-- result will be null

Option 3 (using nullif and COALESCE)

```
Select dividend / COALESCE( nullif(divisor, 0) , 0)
```

-- result will be zero

DateTime (Casting / Formatting)

Convert string to date using style (format) numbers

```
SELECT convert(datetime, '15/03/18', 3) -- 2018-03-15 00:00:00.000
SELECT convert(datetime, '15.03.18', 4) -- 2018-03-15 00:00:00.000
```

Convert datetime to text style (format) list - sql time format

```
-- SQL Server without century (YY) date styles (there are exceptions!)
-- Generally adding 100 to style number results in century format CCYY / YYYY
SELECT convert(varchar, getdate())      -- Mar 15 2018 10:35AM
SELECT convert(varchar, getdate(), 0)  -- Mar 15 2018 10:35AM
SELECT convert(varchar, getdate(), 1)  -- 03/15/18
SELECT convert(varchar, getdate(), 2)  -- 18.03.15
SELECT convert(varchar, getdate(), 3)  -- 15/03/18
SELECT convert(varchar, getdate(), 4)  -- 15.03.18
SELECT convert(varchar, getdate(), 5)  -- 15-03-18
SELECT convert(varchar, getdate(), 6)  -- 15 Mar 18
SELECT convert(varchar, getdate(), 7)  -- Mar 15, 18
SELECT convert(varchar, getdate(), 8)  -- 10:39:39
SELECT convert(varchar, getdate(), 9)  -- Mar 15 2018 10:39:48:373AM
SELECT convert(varchar, getdate(), 10) -- 03-15-18
SELECT convert(varchar, getdate(), 11) -- 18/03/15
SELECT convert(varchar, getdate(), 15) -- 180315
SELECT convert(varchar, getdate(), 13) -- 15 Mar 2018 10:41:07:590
SELECT convert(varchar, getdate(), 14) -- 10:41:25:903
SELECT convert(varchar, getdate(), 20) -- 2018-03-15 10:43:56
SELECT convert(varchar, getdate(), 21) -- 2018-03-15 10:44:04.950
SELECT convert(varchar, getdate(), 22) -- 03/15/18 10:44:50 AM
SELECT convert(varchar, getdate(), 23) -- 2018-03-15
SELECT convert(varchar, getdate(), 24) -- 10:45:45
SELECT convert(varchar, getdate(), 25) -- 2018-03-15 10:46:11.263

-- T-SQL with century (YYYY or CCYY) datetime styles (formats)
SELECT convert(varchar, getdate(), 100) -- Oct 23 2016 10:22AM (or PM)
SELECT convert(varchar, getdate(), 101) -- 10/23/2016
SELECT convert(varchar, getdate(), 102) -- 2016.10.23
SELECT convert(varchar, getdate(), 103) -- 23/10/2016
SELECT convert(varchar, getdate(), 104) -- 23.10.2016
SELECT convert(varchar, getdate(), 105) -- 23-10-2016
SELECT convert(varchar, getdate(), 106) -- 23 Oct 2016
SELECT convert(varchar, getdate(), 107) -- Oct 23, 2016
SELECT convert(varchar, getdate(), 108) -- 09:10:34
SELECT convert(varchar, getdate(), 109) -- Oct 23 2016 11:10:33:993AM (or PM)
SELECT convert(varchar, getdate(), 110) -- 10-23-2016
SELECT convert(varchar, getdate(), 111) -- 2016/10/23
SELECT convert(varchar, getdate(), 112) -- 20161023
SELECT convert(varchar, getdate(), 113) -- 23 Oct 2016 06:10:55:383
SELECT convert(varchar, getdate(), 114) -- 06:10:55:383(24h)
SELECT convert(varchar, getdate(), 120) -- 2016-10-23 06:10:55(24h)
SELECT convert(varchar, getdate(), 121) -- 2016-10-23 06:10:55.383
SELECT convert(varchar, getdate(), 126) -- 2016-10-23T06:10:55.383
GO
```

```

-- SQL cast string to datetime - time part 0 - sql hh mm
-- SQL Server cast string to DATE (SQL Server 2008 feature) - sql yyyy mm dd
SELECT [Date] = CAST('20120228' AS date) -- 2012-02-28
SELECT [Datetime] = CAST('20120228' AS datetime) -- 2012-02-28 00:00:00.000
SELECT [Datetime] = CAST('20120228' AS smalldatetime) -- 2012-02-28 00:00:00

-- SQL convert string to datetime - time part 0
-- SQL Server convert string to date - sql times format
SELECT [Datetime] = CONVERT(datetime,'2010-02-28')
SELECT [Datetime] = CONVERT(smalldatetime,'2010-02-28')

SELECT [Datetime] = CAST('Mar 15, 2010' AS datetime)
SELECT [Datetime] = CAST('Mar 15, 2010' AS smalldatetime)

SELECT [Datetime] = CONVERT(datetime,'Mar 15, 2010')
SELECT [Datetime] = CONVERT(smalldatetime,'Mar 15, 2010')

SELECT [Datetime] = CAST('Mar 15, 2010 12:07:34.444' AS datetime)
SELECT [Datetime] = CAST('Mar 15, 2010 12:07:34.444' AS smalldatetime)

SELECT [Datetime] = CONVERT(datetime,'Mar 15, 2010 12:07:34.444')
SELECT [Datetime] = CONVERT(smalldatetime,'Mar 15, 2010 12:07:34.444')

SELECT [Datetime] = CAST('2010-02-28 12:07:34.444' AS datetime)
SELECT [Datetime] = CAST('2010-02-28 12:07:34.444' AS smalldatetime)

SELECT [Datetime] = CONVERT(datetime,'2010-02-28 12:07:34.444')
SELECT [Datetime] = CONVERT(smalldatetime,'2010-02-28 12:07:34.444')

-- Double conversion
SELECT [Datetime] = CAST(CAST(getdate() AS VARCHAR) AS datetime)
SELECT [Datetime] = CAST(CAST(getdate() AS VARCHAR) AS smalldatetime)

SELECT [Datetime] = CONVERT(datetime,convert(varchar,getdate()))
SELECT [Datetime] = CONVERT(smalldatetime,convert(varchar,getdate()))
-----

-- MSSQL convert date string to datetime - time is set to 00:00:00.000 or 12:00AM
PRINT CONVERT(datetime,'07-10-2016',110) -- Jul 10 2016 12:00AM
PRINT CONVERT(datetime,'2016/07/10',111) -- Jul 10 2016 12:00AM
PRINT CONVERT(varchar,CONVERT(datetime,'20160710', 112),121)
-- 2016-07-10 00:00:00.000
-----

```

```

-- Selected named date styles
DECLARE @DateTimeValue varchar(32)

-- US-Style
-- Convert string to datetime sql - sql convert string to datetime
SELECT @DateTimeValue = '10/23/2016'
SELECT StringDate=@DateTimeValue,
[US-Style] = CONVERT(datetime, @DatetimeValue)

SELECT @DateTimeValue = '10/23/2016 23:01:05'
SELECT StringDate = @DateTimeValue,
[US-Style] = CONVERT(datetime, @DatetimeValue)

-- UK-Style, British/French
SELECT @DateTimeValue = '23/10/16 23:01:05'
SELECT StringDate = @DateTimeValue,
[UK-Style] = CONVERT(datetime, @DatetimeValue, 3)

SELECT @DateTimeValue = '23/10/2016 04:01 PM'
SELECT StringDate = @DateTimeValue,
[UK-Style] = CONVERT(datetime, @DatetimeValue, 103)

-- German-Style
SELECT @DateTimeValue = '23.10.16 23:01:05'
SELECT StringDate = @DateTimeValue,
[German-Style] = CONVERT(datetime, @DatetimeValue, 4)

SELECT @DateTimeValue = '23.10.2016 04:01 PM'
SELECT StringDate = @DateTimeValue,
[German-Style] = CONVERT(datetime, @DatetimeValue, 104)

-- Double conversion to US-Style 107 with century: Oct 23, 2016
SET @DateTimeValue='10/23/16'
SELECT StringDate=@DateTimeValue,
[US-Style] = CONVERT(varchar, CONVERT(datetime, @DateTimeValue),107)

-- SQL dateformat setting
USE AdventureWorks2008;
SELECT convert(datetime,'14/05/08')
/* Msg 242, Level 16, State 3, Line 1
The conversion of a varchar data type to a datetime data type resulted
in an out-of-range value.
*/
SET DATEFORMAT ymd
SELECT convert(datetime,'14/05/08') -- 2014-05-08 00:00:00.000
-- Setting DATEFORMAT to UK-Style
SET DATEFORMAT dmy
SELECT convert(datetime,'20/05/14') -- 2014-05-20 00:00:00.000
-- Setting DATEFORMAT to US-Style
SET DATEFORMAT mdy
SELECT convert(datetime,'05/20/14') -- 2014-05-20 00:00:00.000
SELECT convert(datetime,'05/20/2014') -- 2014-05-20 00:00:00.000
GO

```


SQL date & time eliminating dividing characters

```
-- MSSQL replace string function
-- T-SQL string concatenate (+)
SELECT replace(convert(VARCHAR(10),getdate(),102),'.','')
-- 20120315
SELECT replace(convert(VARCHAR(10),getdate(),111),'/','')
-- 20120315

-- SQL triple replace
SELECT replace(replace(replace(convert(VARCHAR(25),getdate(),20),'-',''),':',''),' ','')
-- 20120529090427

-- T-SQL concatenating from a date and a time conversion
SELECT replace(convert(VARCHAR(10),getdate(),111),'/','') +
    replace(convert(VARCHAR(8),getdate(),108),':','')
-- 20120315085654
```

SQL DATEDIFF with string date

```
DECLARE @sDate varchar(10)
SET @sDate = '2010/03/15'
-- DATEDIFF (delta) between two dates in months
SELECT GETDATE(), DATEDIFF (MONTH, GETDATE(), @sDate)
SELECT GETDATE(), DATEDIFF (MONTH, GETDATE(), CAST(@sDate as datetime))
SELECT GETDATE(), DATEDIFF (MONTH, GETDATE(), CONVERT(datetime,@sDate))
SELECT GETDATE(), DATEDIFF (MONTH, GETDATE(), CONVERT(datetime,@sDate,111))
-- Same results for above: 2008-12-29 11:04:51.097    15

-- SQL convert to datetime with wrong style (111 correct, 112 incorrect)
SELECT GETDATE(), DATEDIFF (MONTH, GETDATE(), CONVERT(datetime,@sDate,112))
/* ERROR
Msg 241, Level 16, State 1, Line 11
Conversion failed when converting date and/or time from character string.*/
```

SQL Server date string search guidelines - comparing dates

```
-- Date equal search
DECLARE @Date1 datetime, @Date2 datetime, @Date3 datetime
SET @Date1 = '2012-01-01'
SET @Date2 = '2012-01-01 00:00:00.000'
SET @Date3 = '2012-01-01 11:00'

SELECT @Date1, @Date2, @Date3
-- Date-only @Date1 is translated to datetime
-- 2012-01-01 00:00:00.000    2012-01-01 00:00:00.000    2012-01-01 11:00:00.000

-- The following is a datetime comparison, not a date-only comparison
IF (@Date1 = @Date2) PRINT 'EQUAL' ELSE PRINT 'NOT EQUAL'
-- EQUAL
-- Equal test fails because time parts are different
IF (@Date1 = @Date3) PRINT 'EQUAL' ELSE PRINT 'NOT EQUAL'
-- NOT EQUAL
```

```

-- The string date implicitly converted to datetime for the equal test
IF ('2012-01-01' = @Date3) PRINT 'EQUAL' ELSE PRINT 'NOT EQUAL'
-- NOT EQUAL

-- Safe way to search for a specific date
SELECT COUNT(*) FROM AdventureWorks.Sales.SalesOrderHeader
WHERE '2004/02/01' = CONVERT(varchar, OrderDate,111)
-- 244

-- Equivalent to
SELECT COUNT(*) FROM AdventureWorks.Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2004/02/01 00:00:00.000' AND '2004/02/01 23:59:59.997'
-- 244

-- Safe way to search for a specific date range
SELECT COUNT(*) FROM AdventureWorks.Sales.SalesOrderHeader
WHERE CONVERT(varchar, OrderDate,111) BETWEEN '2004/02/01' AND '2004/02/14'
-- 1059

-- Equivalent to
SELECT COUNT(*) FROM AdventureWorks.Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2004/02/01 00:00:00.000' AND '2004/02/14 23:59:59.997'
-- 1059
SELECT COUNT(*) FROM AdventureWorks.Sales.SalesOrderHeader
WHERE OrderDate >= '2004/02/01 00:00:00.000'
      AND OrderDate < '2004/02/15 00:00:00.000'
-- 1059
-----

```

SQL Server convert from string to smalldatetime

```

-- T-SQL convert from format mm/dd/yyyy to smalldatetime
SELECT CONVERT(smalldatetime, '10/23/2016', 101)
-- 2016-10-23 00:00:00
-- MSSQL convert from format dd/mm/yyyy to smalldatetime
SELECT CONVERT(smalldatetime, '23/10/2016', 103)
-- 2016-10-23 00:00:00
-- Month 23 is out of range
SELECT CONVERT(smalldatetime, '23/10/2016', 101)
/* Msg 242, Level 16, State 3, Line 1
The conversion of a varchar data type to a smalldatetime data type resulted
in an out-of-range value.*/

```

Translate/convert string/text hours and minutes to seconds

```

DECLARE @TimeStr varchar(16) = '20:49:30'
SELECT  PARSENAME(REPLACE(@TimeStr, ':', '.'),1)
      + PARSENAME(REPLACE(@TimeStr, ':', '.'),2) * 60
      + PARSENAME(REPLACE(@TimeStr, ':', '.'),3) * 3600
-- 74970
-----

```