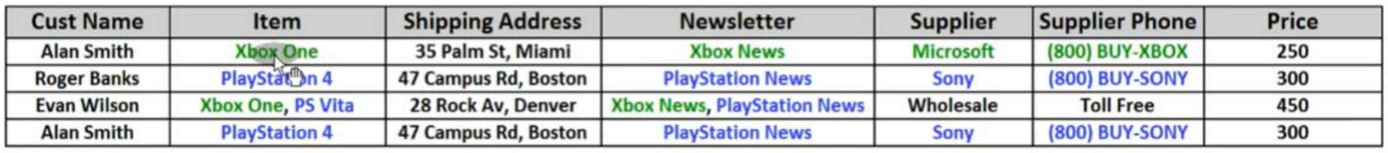
Database Design

Bad database design is superfluous data redundancy, which in itself has two disadvantages: the existence of data anomalies and the use of an unnecessary amount of disk space.

Normalization of data is a process during which the existing tables of a database are tested to find certain dependencies between the columns of a table. If such dependencies exist, the table is restructured into multiple (usually two) tables, which eliminates any column dependencies. If one of these generated tables still contains data dependencies, the process of normalization must be repeated until all dependencies are resolved.

The process of eliminating data redundancy in a table is based upon theory of functional dependencies. This means that by using known value of one column, the corresponding value of another column can always be uniquely determined.

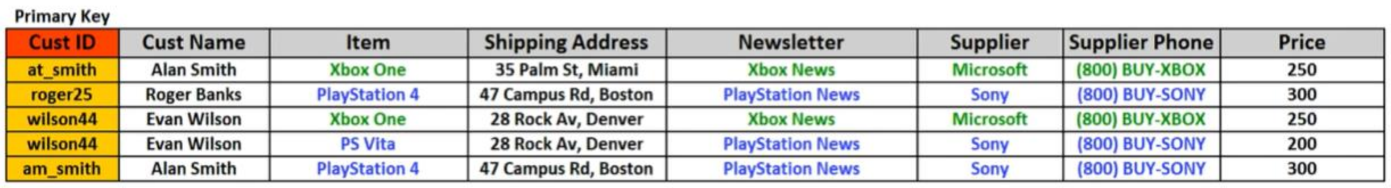


Cust Name	Item	Shipping Address	Newsletter	Supplier	Supplier Phone	Price
Alan Smith	Xbox One	35 Palm St, Miami	Xbox News	Microsoft	(800) BUY-XBOX	250
Roger Banks	PlayStation 4	47 Campus Rd, Boston	PlayStation News	Sony	(800) BUY-SONY	300
Evan Wilson	Xbox One, PS Vita	28 Rock Av, Denver	Xbox News, PlayStation News	Wholesale	Toll Free	450
Alan Smith	PlayStation 4	47 Campus Rd, Boston	PlayStation News	Sony	(800) BUY-SONY	300

Normal Forms:[1]

There are many successive levels to Normalization. These level are called Normal Forms. Each consecutive level is dependent on the previous one. The first three are the most important for practical use.

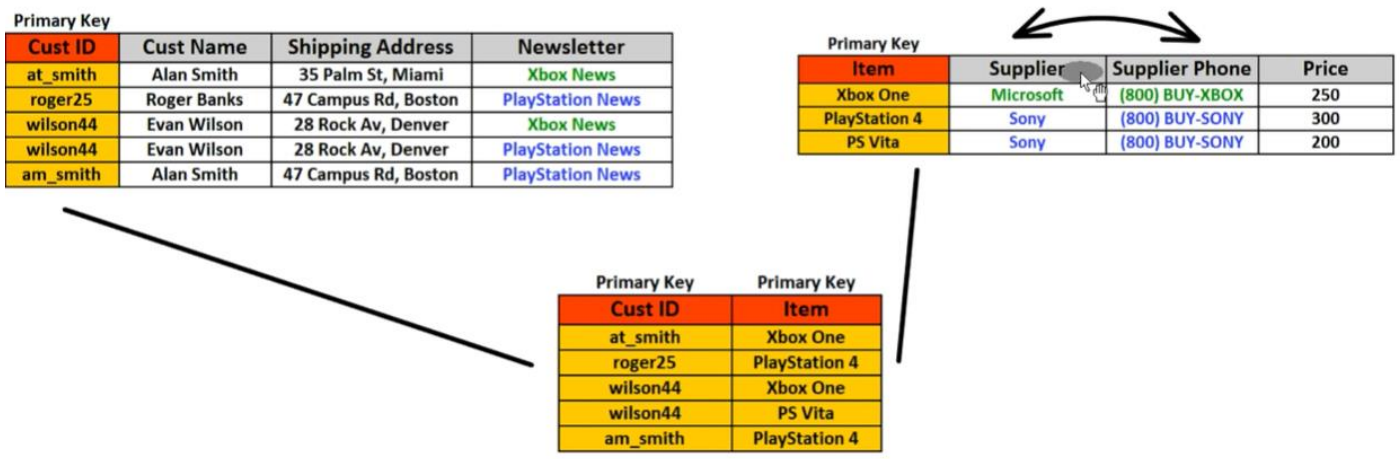
1. **First Normal Form:**
   - Each Cell to be singles valued
   - Entries in a column are same type
   - Rows are uniquely identified – Add unique ID, or Add more column to make unique



Primary Key

Cust ID	Cust Name	Item	Shipping Address	Newsletter	Supplier	Supplier Phone	Price
at_smith	Alan Smith	Xbox One	35 Palm St, Miami	Xbox News	Microsoft	(800) BUY-XBOX	250
roger25	Roger Banks	PlayStation 4	47 Campus Rd, Boston	PlayStation News	Sony	(800) BUY-SONY	300
wilson44	Evan Wilson	Xbox One	28 Rock Av, Denver	Xbox News	Microsoft	(800) BUY-XBOX	250
wilson44	Evan Wilson	PS Vita	28 Rock Av, Denver	PlayStation News	Sony	(800) BUY-SONY	200
am_smith	Alan Smith	PlayStation 4	47 Campus Rd, Boston	PlayStation News	Sony	(800) BUY-SONY	300

2. **Second Normal Form:**
   - All attribute (Non-Key Columns) dependent on the key



Primary Key

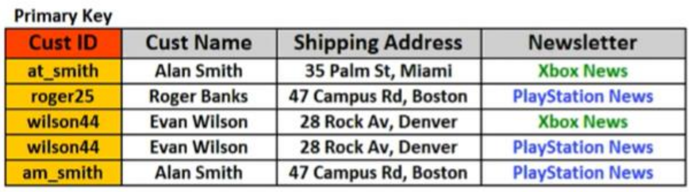
Cust ID	Cust Name	Shipping Address	Newsletter
at_smith	Alan Smith	35 Palm St, Miami	Xbox News
roger25	Roger Banks	47 Campus Rd, Boston	PlayStation News
wilson44	Evan Wilson	28 Rock Av, Denver	Xbox News
wilson44	Evan Wilson	28 Rock Av, Denver	PlayStation News
am_smith	Alan Smith	47 Campus Rd, Boston	PlayStation News

Primary Key

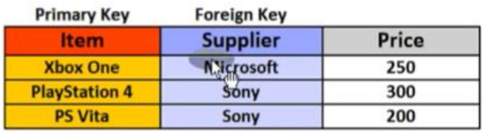
Item	Supplier	Supplier Phone	Price
Xbox One	Microsoft	(800) BUY-XBOX	250
PlayStation 4	Sony	(800) BUY-SONY	300
PS Vita	Sony	(800) BUY-SONY	200

Primary Key	Primary Key
Cust ID	Item
at_smith	Xbox One
roger25	PlayStation 4
wilson44	Xbox One
wilson44	PS Vita
am_smith	PlayStation 4

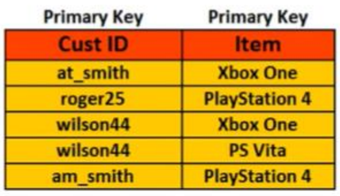
3. **Third Normal Form:**
   - All Fields (columns) can be determined Only by the Key in the table and no other column



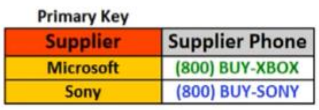
Cust ID (Primary Key)	Cust Name	Shipping Address	Newsletter
at_smith	Alan Smith	35 Palm St, Miami	Xbox News
roger25	Roger Banks	47 Campus Rd, Boston	PlayStation News
wilson44	Evan Wilson	28 Rock Av, Denver	Xbox News
wilson44	Evan Wilson	28 Rock Av, Denver	PlayStation News
am_smith	Alan Smith	47 Campus Rd, Boston	PlayStation News



Item (Primary Key)	Supplier (Foreign Key)	Price
Xbox One	Microsoft	250
PlayStation 4	Sony	300
PS Vita	Sony	200

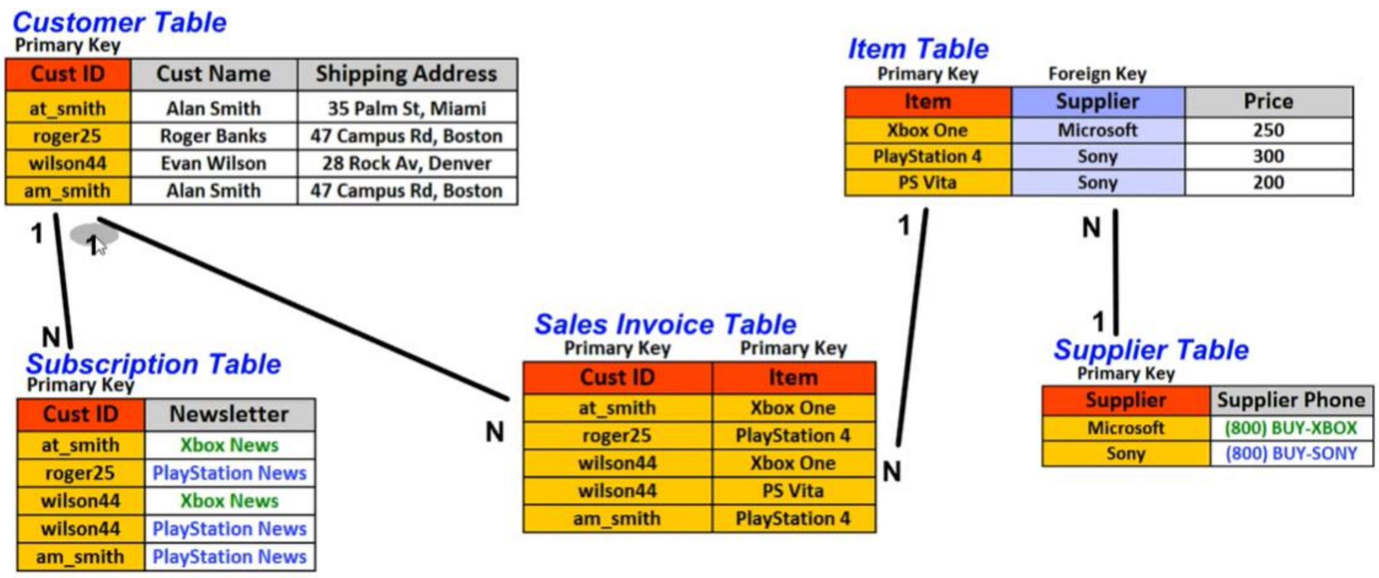


Cust ID (Primary Key)	Item (Primary Key)
at_smith	Xbox One
roger25	PlayStation 4
wilson44	Xbox One
wilson44	PS Vita
am_smith	PlayStation 4



Supplier (Primary Key)	Supplier Phone
Microsoft	(800) BUY-XBOX
Sony	(800) BUY-SONY

4. **Fourth Normal Form:**
   - No multi-valued dependencies



Customer Table

Cust ID (Primary Key)	Cust Name	Shipping Address
at_smith	Alan Smith	35 Palm St, Miami
roger25	Roger Banks	47 Campus Rd, Boston
wilson44	Evan Wilson	28 Rock Av, Denver
am_smith	Alan Smith	47 Campus Rd, Boston

Item Table

Item (Primary Key)	Supplier (Foreign Key)	Price
Xbox One	Microsoft	250
PlayStation 4	Sony	300
PS Vita	Sony	200

Subscription Table

Cust ID (Primary Key)	Newsletter
at_smith	Xbox News
roger25	PlayStation News
wilson44	Xbox News
wilson44	PlayStation News
am_smith	PlayStation News

Sales Invoice Table

Cust ID (Primary Key)	Item (Primary Key)
at_smith	Xbox One
roger25	PlayStation 4
wilson44	Xbox One
wilson44	PS Vita
am_smith	PlayStation 4

Supplier Table

Supplier (Primary Key)	Supplier Phone
Microsoft	(800) BUY-XBOX
Sony	(800) BUY-SONY

Entity-Relationship Model

The entity-relationship (ER) model is used to design relational databases by removing all existing redundancy in the data. The basic object of the ER model is an entity—that is, a real-world object. Each entity has several attributes, which are properties of the entity and therefore describe it. Based on its type, an attribute can be

Atomic (or single values)

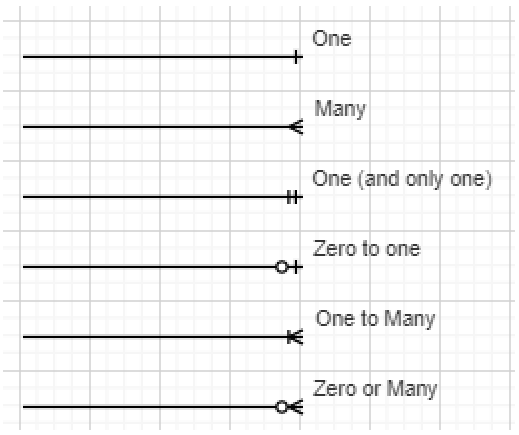
An atomic attribute is always represented by a single value for a particular entity**Multivalued**

A multivalued attribute may have one or more values for a particular entity.

Composite

Composite attributes are not atomic because they are assembled using some other atomic attributes.

A relationship exists when an entity refers to one or more other entities. Every existing relationship between two entities must be one of the following three types: 1:1, 1:N, or M:N.



Chapter 2

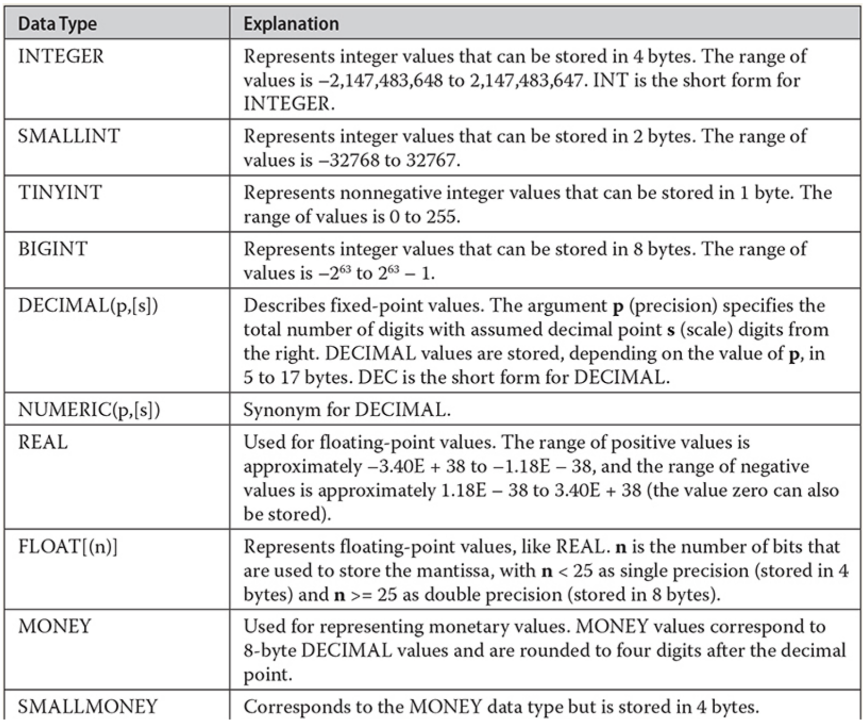
SQL's Basic Objects

The language of the Database Engine, Transact-SQL, has the same basic features as other common programming languages:

- **Literal Values:** A literal value is, for example, an alphanumerical, hexadecimal, or numeric constant.
- **Identifiers**: In Transact-SQL, identifiers are used to identify database objects such as databases, tables, and indices. They are represented by character strings that may include up to 128 characters and may contain letters, numerals, or the following characters: _, @, #, and $.
- **Delimiters:** double quotations marks is used for delimiters for so-called delimited identifier. It is special kind of identifier usually used to allow the use of reserved keywords as identifiers and also to allow spaces in the names of database objects.
- **Comments:** Using the pair of characters /* and */ marks the enclosed text as a comment

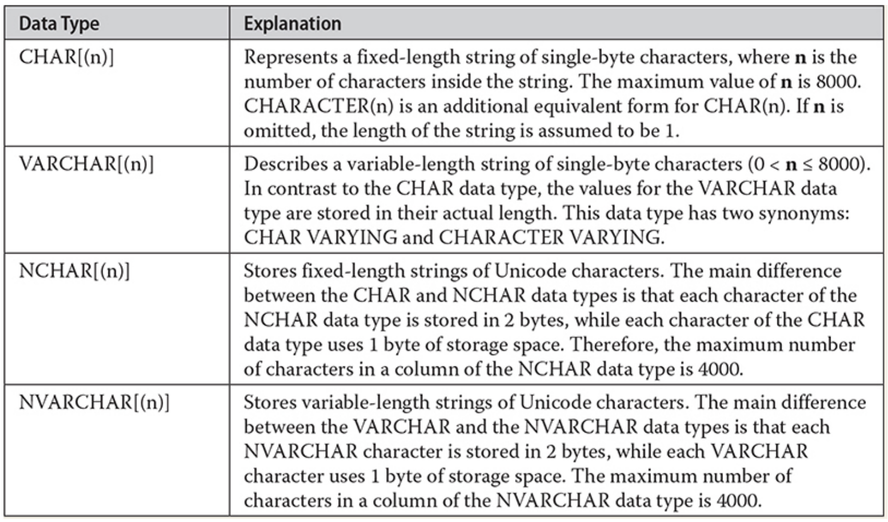
Data Types

Numeric data types: used to represent numbers



Data Type	Explanation
INTEGER	Represents integer values that can be stored in 4 bytes. The range of values is –2,147,483,648 to 2,147,483,647. INT is the short form for INTEGER.
SMALLINT	Represents integer values that can be stored in 2 bytes. The range of values is –32768 to 32767.
TINYINT	Represents nonnegative integer values that can be stored in 1 byte. The range of values is 0 to 255.
BIGINT	Represents integer values that can be stored in 8 bytes. The range of values is -2^{63} to $2^{63} - 1$.
DECIMAL(p,[s])	Describes fixed-point values. The argument **p** (precision) specifies the total number of digits with assumed decimal point **s** (scale) digits from the right. DECIMAL values are stored, depending on the value of **p**, in 5 to 17 bytes. DEC is the short form for DECIMAL.
NUMERIC(p,[s])	Synonym for DECIMAL.
REAL	Used for floating-point values. The range of positive values is approximately –3.40E + 38 to –1.18E – 38, and the range of negative values is approximately 1.18E – 38 to 3.40E + 38 (the value zero can also be stored).
FLOAT[(n)]	Represents floating-point values, like REAL. **n** is the number of bits that are used to store the mantissa, with **n** < 25 as single precision (stored in 4 bytes) and **n** >= 25 as double precision (stored in 8 bytes).
MONEY	Used for representing monetary values. MONEY values correspond to 8-byte DECIMAL values and are rounded to four digits after the decimal point.
SMALLMONEY	Corresponds to the MONEY data type but is stored in 4 bytes.

Character data types: strings of single-byte characters or strings of Unicode characters



Data Type	Explanation
CHAR[(n)]	Represents a fixed-length string of single-byte characters, where **n** is the number of characters inside the string. The maximum value of **n** is 8000. CHARACTER(n) is an additional equivalent form for CHAR(n). If **n** is omitted, the length of the string is assumed to be 1.
VARCHAR[(n)]	Describes a variable-length string of single-byte characters (0 < **n** ≤ 8000). In contrast to the CHAR data type, the values for the VARCHAR data type are stored in their actual length. This data type has two synonyms: CHAR VARYING and CHARACTER VARYING.
NCHAR[(n)]	Stores fixed-length strings of Unicode characters. The main difference between the CHAR and NCHAR data types is that each character of the NCHAR data type is stored in 2 bytes, while each character of the CHAR data type uses 1 byte of storage space. Therefore, the maximum number of characters in a column of the NCHAR data type is 4000.
NVARCHAR[(n)]	Stores variable-length strings of Unicode characters. The main difference between the VARCHAR and the NVARCHAR data types is that each NVARCHAR character is stored in 2 bytes, while each VARCHAR character uses 1 byte of storage space. The maximum number of characters in a column of the NVARCHAR data type is 4000.

Chapter 3 – Data Definition Language (DDL)

The DDL statements are divided into three groups, which are discussed in turn.

1. The first group includes statements that create objects,
2. The second group includes statements that modify the structure of objects, and
3. The third group includes statements that remove objects.

Creating Database Objects

Physical Objects

- The physical objects are related to the organization of the data on the physical device (disk).
- The Database Engine's physical objects are files and filegroups.

Logical Objects

- Logical objects represent a user's view of a database
- Databases, tables, columns, and views (virtual tables) are examples of logical objects.

1. Creation of a Database

To create the database graphically

1. Right Click on Databases folder in the Object explorer
2. Select New Database
3. In the New Database dialog box, enter the Database name and click OK.

To create the database using a query:

```
CREATE DATABASE DatabaseName
```

Whether, you create a database graphically using the designer or, using a query, the following 2 files gets generated.

.MDF file - Data File (Contains actual data)
.LDF file - Transaction Log file (Used to recover the database)

To alter a database, once it's created:

```
ALTER DATABASE DatabaseName MODIFY NAME = NewDatabaseName
```

Alternatively, you can also use system stored procedure

EXCURE SP_RENAMEDB 'OldDatabaseName','NewDatabaseName'

To Delete or Drop a database

```
DROP DATABASE DatabaseName
```

Dropping a database, deletes the LDF and MDF files. You cannot drop a database, if it is currently in use. You get an error stating - Cannot drop database "NewDatabaseName" because it is currently in use. So, if other users are connected, you need to put the database in single user mode and then drop the database.

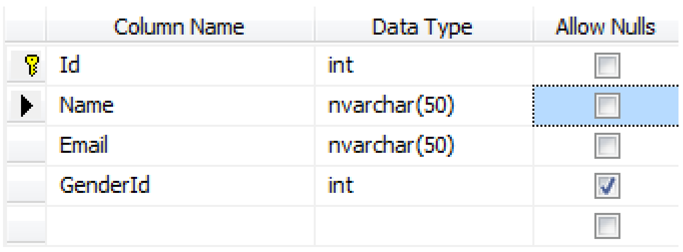
Alter Database DatabaseName Set SINGLE_USER With Rollback Immediate

With Rollback Immediate option, will rollback all incomplete transactions and closes the connection to the database

2. Creating and working with tables

To create tblPerson table, graphically, using SQL Server Management Studio

1. Right click on Tables folder in Object explorer window
2. Select New Table
3. Fill Column Name, Data Type and Allow Nulls, as shown below and save the table as tblPerson.



Column Name	Data Type	Allow Nulls
Id	int	☐
Name	nvarchar(50)	☐
Email	nvarchar(50)	☐
GenderId	int	✓
		☐

To create table using query:

```
CREATE TABLE tblGender (
       ID int NOT NULL PRIMARY KEY,
       Gender nvarchar(50) NOT NULL
)
```

The general formula:

```
ALTER TABLE ForeignKeyTable
ADD CONSTRAINT ForeignKeyTable_ForiegnKeyColumn_FK
FOREIGN KEY (ForiegnKeyColumn)
REFERENCES PrimaryKeyTable (PrimaryKeyColumn)
```

To add foreign key reference using a query onto existing table:

```
ALTER TABLE          tblPerson
ADD CONSTRAINT       tblPerson_GenderId_FK
FOREIGN KEY          (GenderId)
REFERENCES           tblGender(ID)
```

3. Updating values into exisiting rows and columns

```
UPDATE tblPerson
SET Age = 24
WHERE Name = 'Ahsan'
```

4. ALTER STATEMENT

Multiple columns to a table

```
ALTER TABLE TableName
ADD
       ColumnName1 DataType1 ColumnConstraint1,
       ColumnName2 DataType2 ColumnConstraint2,
       …,
       ColumnNamen DataTypen ColumnConstraintn,
```

5. Cascading referential integrity constraint

1. **No Action:** This is the default behavior. No Action specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, an error is raised and the DELETE or UPDATE is rolled back.

2. **Cascade:** Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are also deleted or updated.

3. **Set NULL:** Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to NULL.

4. **Set Default:** Specifies that if an attempt is made to delete or update a row with a key referenced by foreign keys in existing rows in other tables, all rows containing those foreign keys are set to default values.

6. CREATE TABLE and Declarative Integrity Constraints

All declarative constraints can be categorized into several groups:

- **DEFAULT clause** in the column definition specifies the default value of the column—that is, whenever a new row is inserted into the table, the default value for the particular column will be used if there is no value specified for it.

Altering an existing column to add a default constraint:

```
ALTER TABLE { TABLE_NAME }
ADD CONSTRAINT { CONSTRAINT_NAME }
DEFAULT { DEFAULT_VALUE } FOR {
EXISTING_COLUMN_NAME }
```

Adding a new column, with default value, to an existing table:

```
ALTER TABLE { TABLE_NAME }
ADD { COLUMN_NAME } { DATA_TYPE } { NULL | NOT
NULL }
CONSTRAINT { CONSTRAINT_NAME } DEFAULT {
DEFAULT_VALUE }
```

- **UNIQUE clause** is use to enforce uniqueness of a column i.e. the column should not allow any duplicate values.

Sometimes more than one column or group of columns of the table have unique values and therefore can be used as the primary key. All columns or groups of columns that qualify to be primary keys are called *candidate keys*. Each candidate key is defined using the UNIQUE clause in the CREATE TABLE or the ALTER TABLE statement.

The UNIQUE clause has the following form:

CONSTRAINT c_name
UNIQUE [CLUSTERED | NONCLUSTERED]
({col_name1}, …)

- **PRIMARY KEY** clause
- **CHECK clause** is used to limit the range of the values that can be entered for a column. – Boolean

The general formula for adding check constraint in SQL Server:

ALTER TABLE { TABLE_NAME }
ADD CONSTRAINT { CONSTRAINT_NAME } CHECK (BOOLEAN_EXPRESSION)

- **FOREIGN KEY** clause and referential integrity

7. Identity Column in SQL Server

A column with the IDENTITY property allows only integer values, which are usually implicitly assigned by the system.

Delete the row, that you have just inserted and insert another row. You see that the value for PersonId is 2. Now if you insert another row, PersonId is 3. A record with PersonId = 1, does not exist, and I want to fill this gap. To do this, we should be able to explicitly supply the value for identity column. To explicitly supply a value for identity column

1. First turn on identity insert - SET IDENTITY_INSERT tblPerson ON
2. In the insert query specify the column list
   INSERT INTO tblPerson(PersonId, Name) VALUES (2, 'John')

After, you have the gaps in the identity column filled, and if you wish SQL server to calculate the value, turn off Identity_Insert.

SET Identity_Insert tblPerson OFF

If you have deleted all the rows in a table, and you want to reset the identity column value, use DBCC CHECKIDENT command. This command will reset PersonId identity column.

DBCC CHECKIDENT(tblPerson, RESEED, 0)

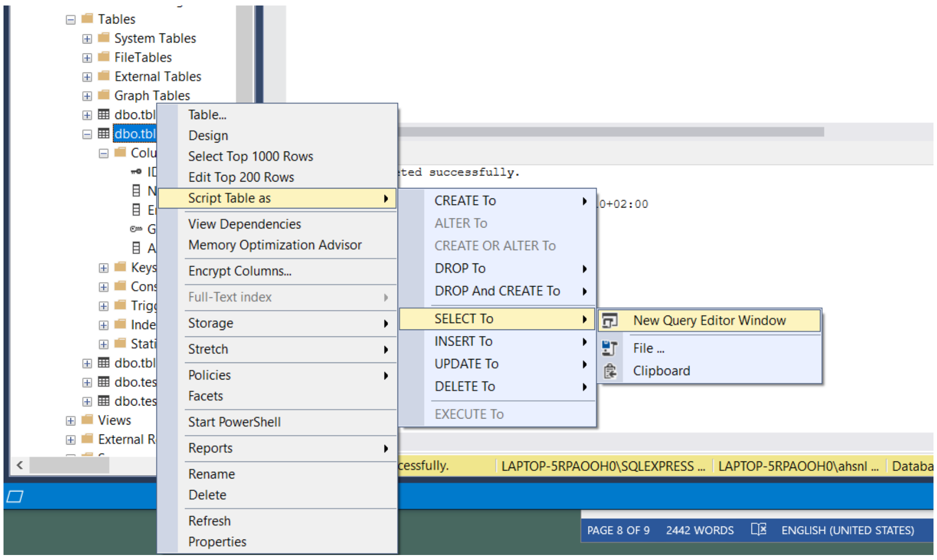
8. How to obtain the last generated identity column value in SQL server

Example queries for getting the last generated identity value:

SELECT SCOPE_IDENTITY() – same session and the same scope and the same scope
SELECT @@IDENTITY – same session and across any scope
SELECT IDENT_CURRENT('TableName') – specific table across any session and any scope

9. All about 'SELECT' Statement

- **Select specific or all columns**



- **Distinct Rows**
- **Filtering with where clause**
- **Wild cards in SQL Server**
- **Joining multiple conditions using AND and OR operators**
- **Sorting rows using order by**
- **Selecting top *n* or top *n* percentage of rows**
-

Operator	Purpose
=	Equal to
!= or <>	Not Equal to
> , <	Greater than , Less Than
>= , <=	Greater than or equal to, Less than or equal to
IN	Specify a list of values
BETWEEN	Specify a range
LIKE	Specify a pattern
NOT	Not in a list, range etc.
%	Specifies zero or more characters
_ (underscore)	Specify exactly one character
[ ] – numeric , '[ ]' - text	Any character with in the brackets
[^]	Not any character with in the brackets

Chapter 3 – extends of DDL in SQL

Group By

Group by clause is used to the group a selected set of rows into a set of summary rows by the values of one or more columns or expressions. It is always used in conjunction with one or more aggregate functions.

SQL server aggrated functions:

1. COUNT()
2. SUM()
3. AVG()

4. MIN()
5. MAX()

Example:

- Query for retrieving total salaries by city:

We apply the SUM() aggregate function on Salary Column, and grouping by city column.

```
SELECT City, SUM(Salary) AS TotalSalary
FROM tblPerson
GROUP BY City
```

- Query for retrieving total salaries by city and by gender:

```
SELECT City, GenderId, SUM(Salary) AS TotalSalary
FROM tblPerson
GROUP BY City, GenderId
ORDER BY city
```

- Query for retrieving total salaries and total number of employees by City, and by GenderId:

```
SELECT City, GenderId, SUM(Salary) AS TotalSalary, COUNT(ID) AS [Total Employees]
FROM tblPerson
GROUP BY City, GenderId
ORDER BY city
```

- Filtering Groups:
  - **WHERE clause** is used to filter rows before aggregation
  - **HAVING clause** is used to filter groups after aggregations.

WHERE clause:

```
    SELECT City, GenderId, SUM(Salary) AS TotalSalary, COUNT(ID) AS [Total Employees]
    FROM tblPerson
    WHERE City = 'Prague'
    GROUP BY City, GenderId
```

HAVING clause:

```
    SELECT City, GenderId, SUM(Salary) AS TotalSalary, COUNT(ID) AS [Total Employees]
    FROM tblPerson
    GROUP BY City, GenderId
    HAVING City = 'Prague'
```

Differences between WHERE and HAVING clause:

- WHERE clause can be used with – SELECT, INSERT, and UPDATE statement, where as HAVING clause can only be used by SELECT statement
- WHERE filters rows before aggregation (GROUP BY), where as HAVING filters groups, after the aggregations are perform

Joins

Joins are used to retrieve data from 2 or more related tables. In general, tables are related to each other using Foreign Key constraints

In SQL server, there are different types of JOINS:

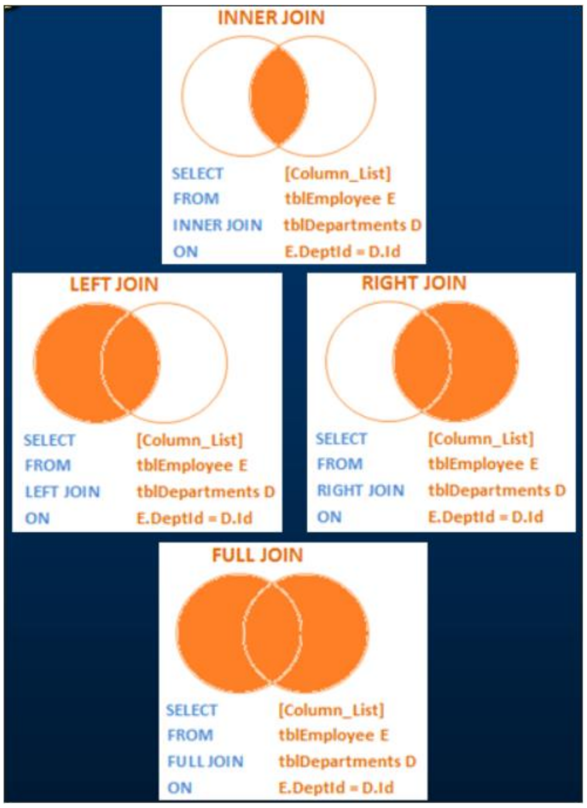
1. INNER JOIN – returns all the rows from multiple table where the join condition is satisfy. Non matching row are eliminated.
2. OUTER JOIN – returns al the records when they are matching in any of the table. Therefore, it return all the rows from LH and RH. It divided into 3 types:
   a. Left Join or Left Outer Join – returns all the matching rows + non-matching rows from the left table

b. Right Join or Right Outer Join
c. Full Join or Full Outer Join

3. CROSS JOIN - produce the Cartesian product of the 2 tables involved in the join. Cross join should not include ON clause

General Formula

```
SELECT column_list
FROM left_table_name
JoinType right_table_name
ON JoinCondition
```

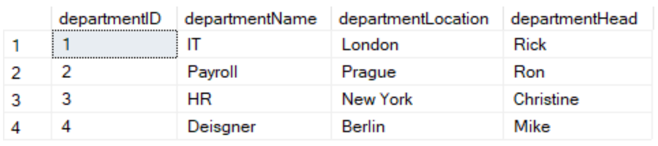


```
SELECT      [Column_List]
FROM        tblEmployee E
INNER JOIN  tblDepartments D
ON          E.DeptId = D.Id
```

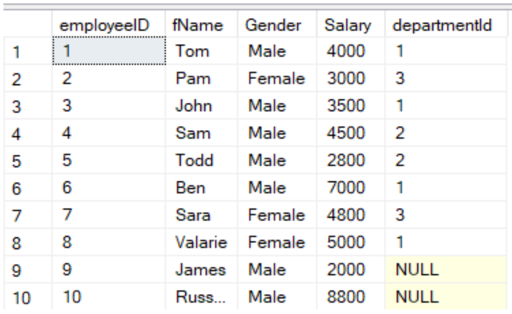
```
SELECT      [Column_List]
FROM        tblEmployee E
LEFT JOIN   tblDepartments D
ON          E.DeptId = D.Id
```

```
SELECT      [Column_List]
FROM        tblEmployee E
RIGHT JOIN  tblDepartments D
ON          E.DeptId = D.Id
```

```
SELECT      [Column_List]
FROM        tblEmployee E
FULL JOIN   tblDepartments D
ON          E.DeptId = D.Id
```



	departmentID	departmentName	departmentLocation	departmentHead
1	1	IT	London	Rick
2	2	Payroll	Prague	Ron
3	3	HR	New York	Christine
4	4	Deisgner	Berlin	Mike

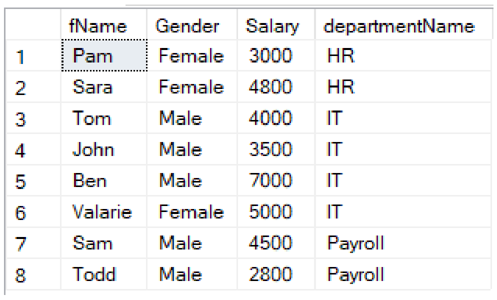


	employeeID	fName	Gender	Salary	departmentId
1	1	Tom	Male	4000	1
2	2	Pam	Female	3000	3
3	3	John	Male	3500	1
4	4	Sam	Male	4500	2
5	5	Todd	Male	2800	2
6	6	Ben	Male	7000	1
7	7	Sara	Female	4800	3
8	8	Valarie	Female	5000	1
9	9	James	Male	2000	NULL
10	10	Russ...	Male	8800	NULL

Example:

INNER JOINS:

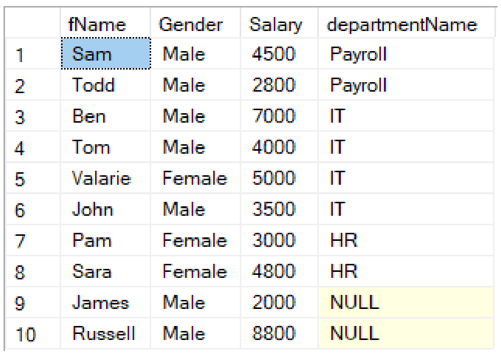
```
SELECT fName, Gender, Salary, departmentName
FROM EmployeeTbl
INNER JOIN DepartmentTbl
ON EmployeeTbl.departmentId =
DepartmentTbl.departmentID
order by departmentName
```



	fName	Gender	Salary	departmentName
1	Pam	Female	3000	HR
2	Sara	Female	4800	HR
3	Tom	Male	4000	IT
4	John	Male	3500	IT
5	Ben	Male	7000	IT
6	Valarie	Female	5000	IT
7	Sam	Male	4500	Payroll
8	Todd	Male	2800	Payroll

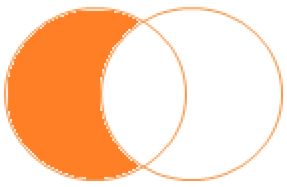
LEFT OUTER JOINS:

```
SELECT fName, Gender, Salary, departmentName
FROM EmployeeTbl
LEFT JOIN DepartmentTbl
ON EmployeeTbl.departmentId =
DepartmentTbl.departmentID
order by departmentName DESC
```

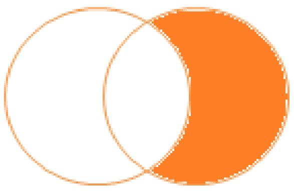


	fName	Gender	Salary	departmentName
1	Sam	Male	4500	Payroll
2	Todd	Male	2800	Payroll
3	Ben	Male	7000	IT
4	Tom	Male	4000	IT
5	Valarie	Female	5000	IT
6	John	Male	3500	IT
7	Pam	Female	3000	HR
8	Sara	Female	4800	HR
9	James	Male	2000	NULL
10	Russell	Male	8800	NULL

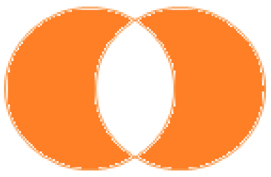
Advance or intelligent joins



```
SELECT      [Column_List]
FROM        tblEmployee E
LEFT JOIN   tblDepartments D
ON          E.DeptId = D.Id
WHERE       D.Id IS NULL
```



```
SELECT      [Column_List]
FROM        tblEmployee E
RIGHT JOIN  tblDepartments D
ON          E.DeptId = D.Id
WHERE       E.DeptId IS NULL
```

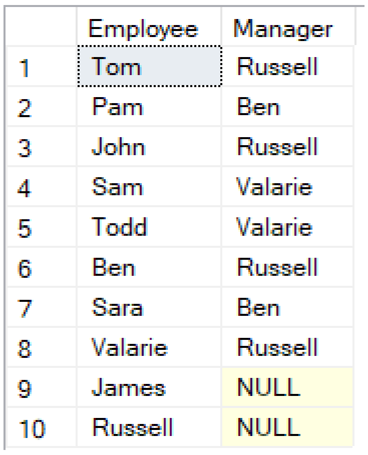


```
SELECT      [Column_List]
FROM        tblEmployee E
FULL JOIN   tblDepartments D
ON          E.DeptId = D.Id
WHERE       E.DeptId IS NULL
OR          D.Id IS NULL
```

Self Joins

Self Join EmployeeTbl table:

```
SELECT         E.fName as Employee, M.fName as Manager
FROM           EmployeeTbl E
LEFT JOIN      EmployeeTbl M
ON                    E.ManagerID = M.employeeID
```

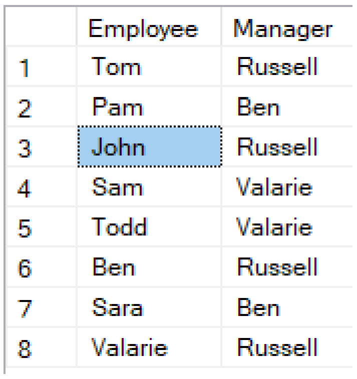


	Employee	Manager
1	Tom	Russell
2	Pam	Ben
3	John	Russell
4	Sam	Valarie
5	Todd	Valarie
6	Ben	Russell
7	Sara	Ben
8	Valarie	Russell
9	James	NULL
10	Russell	NULL

In short, joining a table with itself is called as SELF JOIN. SELF JOIN is not a different type of JOIN. It can be classified under any type of JOIN - INNER, OUTER or CROSS Joins. The above query is, LEFT OUTER SELF Join.

Inner Self Join EmployeeTbl table:

```
SELECT         E.fName as Employee, M.fName as Manager
FROM           EmployeeTbl E
INNER JOIN     EmployeeTbl M
ON             E.ManagerID = M.employeeID
```



	Employee	Manager
1	Tom	Russell
2	Pam	Ben
3	John	Russell
4	Sam	Valarie
5	Todd	Valarie
6	Ben	Russell
7	Sara	Ben
8	Valarie	Russell

Cross Self Join EmployeeTbl table:

Different ways to replace NULL in sql server

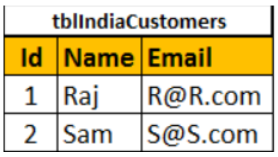
```
-- Replacing NULL value using ISNULL() function:
SELECT        E.fName as Employee, ISNULL(M.fName, 'Not A Manager') as Manager
FROM          EmployeeTbl E
LEFT JOIN     EmployeeTbl M
ON            E.ManagerID = M.employeeID

-- Replacing NULL value using CASE Statement:
SELECT        E.fName as Employee, CASE WHEN M.fName IS NULL THEN 'Not A Manager' ELSE M.fName END
as Manager
FROM          EmployeeTbl E
LEFT JOIN     EmployeeTbl M
ON            E.ManagerID = M.employeeID

-- Replacing NULL value using COALESCE() function: Return the first NON NULL values
SELECT        E.fName as Employee, COALESCE(M.fName, 'Not A Manager') as Manager
FROM          EmployeeTbl E
LEFT JOIN     EmployeeTbl M
ON            E.ManagerID = M.employeeID
```

Coalesce() function in sql server

https://csharp-video-tutorials.blogspot.com/2012/08/coalesce-function-in-sql-server-part-16.html

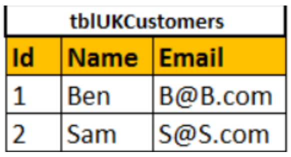
Union and Union All

UNION and UNION ALL operators in SQL Server, are used to combine the result-set of two or more SELECT queries.



tblIndiaCustomers

Id	Name	Email
1	Raj	R@R.com
2	Sam	S@S.com

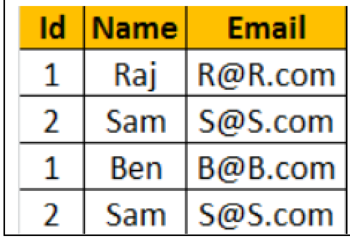


tblUKCustomers

Id	Name	Email
1	Ben	B@B.com
2	Sam	S@S.com

UNION ALL

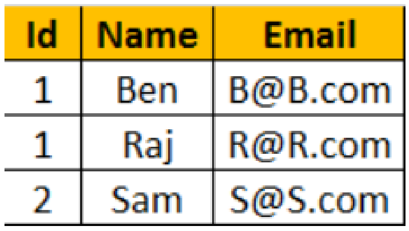
```
Select Id, Name, Email from tblIndiaCustomers
UNION ALL
Select Id, Name, Email from tblUKCustomers
```



Id	Name	Email
1	Raj	R@R.com
2	Sam	S@S.com
1	Ben	B@B.com
2	Sam	S@S.com

UNION

```
Select Id, Name, Email from tblIndiaCustomers
UNION
Select Id, Name, Email from tblUKCustomers
```



Id	Name	Email
1	Ben	B@B.com
1	Raj	R@R.com
2	Sam	S@S.com

Differences between UNION and UNION ALL :

- UNION removes duplicate rows, whereas UNION ALL does not
- UNION has to perform distinct sort to removes duplicates, which makes it less faster than UNION ALL

Differences between JOINS and UNIONS:

- UNION combines the result-set of two or more selected queries into a single result-set which includes all the rows from all the queries in the union

- JOINS retrieves data from two or more tables based on logical relationships between the tables.
- **In short,** UNION combines row from 2 or more table and JOINS combines columns from 2 or more table.

Stored procedures

A stored procedure is group of T-SQL (Transact SQL) statements. If you have a situation, where you write the same query over and over again, you can save that specific query as a stored procedure and call it just by it's name.

Creating a simple stored procedure without any parameters: This stored procedure, retrieves Name and Gender of all the employees. To create a stored procedure we use, **CREATE PROCEDURE** or **CREATE PROC** statement.

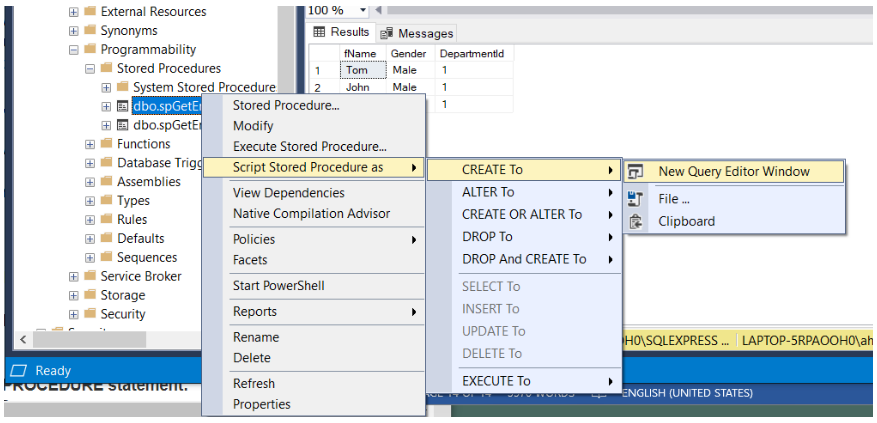
```
CREATE PROCEDURE spGetEmployees
AS
BEGIN
	SELECT fName, Gender FROM EmployeeTbl
END

-- to execute store procedure:
spGetEmployees
EXEC spGetEmployees
EXECUTE spGetEmployees
```

Creating a stored procedure with input parameters:

```
CREATE PROC spGetEmployeesByGenderAndDepartment
@Gender nvarchar(20),
@DepartmentId int
AS
BEGIN
	SELECT fName, Gender, DepartmentId FROM EmployeeTbl WHERE Gender = @Gender
	and DepartmentId = @DepartmentId
END

spGetEmployeesByGenderAndDepartment 'Male', 1
```

To view the text, of the stored procedure:

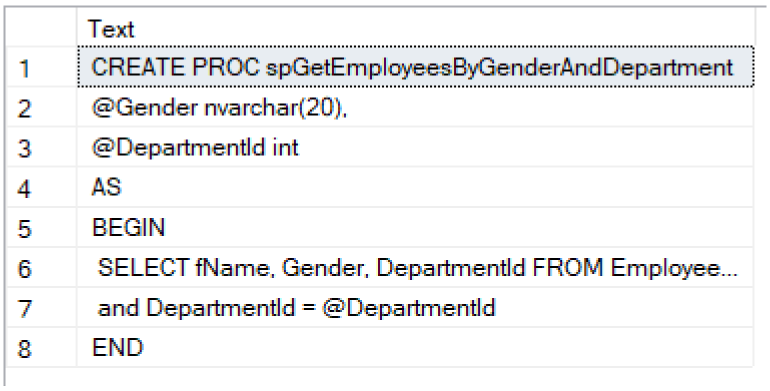
1. Object Explorer



	fName	Gender	DepartmentId
1	Tom	Male	1
2	John	Male	1
			1

2. Use system stored procedure `sp_helptext spName`

Result:



	Text
1	CREATE PROC spGetEmployeesByGenderAndDepartment
2	@Gender nvarchar(20),
3	@DepartmentId int
4	AS
5	BEGIN
6	SELECT fName, Gender, DepartmentId FROM Employee...
7	and DepartmentId = @DepartmentId
8	END

To change the stored procedure, use ALTER PROCEDURE statement:

```
ALTER PROCEDURE spGetEmployees
AS
BEGIN
       SELECT fName, Gender FROM EmployeeTbl ORDER BY fName
END
```

To encrypt the text of the SP, use the `WITH ENCRYPTION`. Once, encrypted, you cannot view the text of the procedure, using sp_helptext system stored procedure.

```
{CREATE | ALTER} PROC spGetEmployeesByGenderAndDepartment
@Gender nvarchar(20),
@DepartmentId int
WITH ENCRYPTION
AS
BEGIN
 SELECT fName, Gender, DepartmentId FROM EmployeeTbl WHERE Gender = @Gender
 and DepartmentId = @DepartmentId
END
```

Stored procedures with output parameters

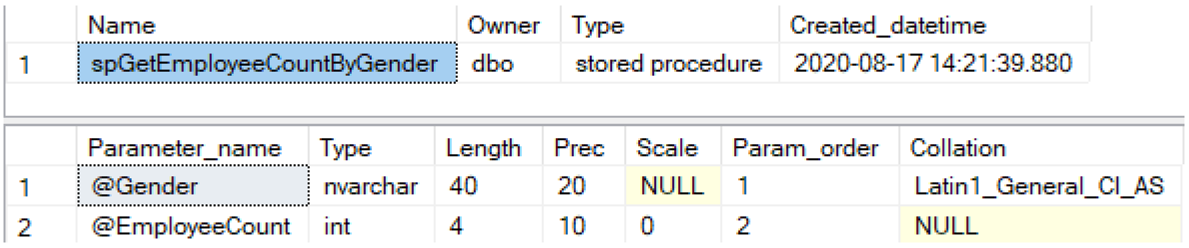
To create an SP with output parameter, we use the keywords OUT or OUTPUT.

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

```
CREATE PROCEDURE spGetEmployeeCountByGender
@Gender nvarchar(20),
@EmployeeCount int OUTPUT
AS
BEGIN
       SELECT @EmployeeCount = COUNT(employeeID) FROM EmployeeTbl WHERE Gender = @Gender
END

-- To executre this SP with OUTPUT parameter
DECLARE @TotalCount int
EXECUTE spGetEmployeeCountByGender 'Male', @TotalCount OUTPUT
PRINT @TotalCount
```

The following system stored procedures, are extremely useful when working procedures.
sp_help SP_Name : View the information about the stored procedure, like parameter names, their datatypes etc. sp_help can be used with any database object, like tables, views, SP's, triggers etc. Alternatively, you can also press ALT+F1, when the name of the object is highlighted.

Example: `sp_help spGetEmployeeCountByGender`



	Name	Owner	Type	Created_datetime
1	spGetEmployeeCountByGender	dbo	stored procedure	2020-08-17 14:21:39.880

	Parameter_name	Type	Length	Prec	Scale	Param_order	Collation
1	@Gender	nvarchar	40	20	NULL	1	Latin1_General_CI_AS
2	@EmployeeCount	int	4	10	0	2	NULL

sp_helptext SP_Name : View the Text of the stored procedure

sp_depends SP_Name : View the dependencies of the stored procedure. This system SP is very useful, especially if you want to check, if there are any stored procedures that are referencing a table that you are abput to drop. sp_depends can also be used with other database objects like table etc.

Stored procedure output parameters or return values

What are stored procedure status variables?

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

Example: **OUTPUT** variable

The following procedure returns total number of employees in the Employees table, using output parameter:

```
CREATE PROC spGetTotalCountOfEmployees
@TotalCount int OUTPUT
AS
BEGIN
       SELECT @TotalCount = COUNT(employeeID) FROM EmployeeTbl
END

DECLARE @TotalEmployee int
EXECUTE spGetTotalCountOfEmployees @TotalEmployee OUTPUT
SELECT @TotalEmployee AS TotalEmployee
```

We are retrieving the Name of the employee, based on their Id, using the output parameter

```
CREATE PROC spGetNameById
@employeeId int,
@fName nvarchar(20) OUTPUT
AS
BEGIN
       SELECT @fName = fName FROM EmployeeTbl WHERE employeeID = @employeeId
END

DECLARE       @fName nvarchar(20)
EXECUTE spGetNameById 10, @fName OUTPUT
PRINT  'First Name = ' + @fName
```

Example: **RETURN** variable

```
CREATE PROC spGetTotalCountOfEmployees2
AS
BEGIN
       RETURN (SELECT COUNT(employeeID) FROM EmployeeTbl)
END

DECLARE @TotalEmployee int
EXECUTE @TotalEmployee = spGetTotalCountOfEmployees2
PRINT @TotalEmployee
```

In general, RETURN values are used to indicate success or failure of stored procedure, especially when we are dealing with nested stored procedures. Return a value of 0, indicates success, and any nonzero value indicates failure.

Difference between return values and output parameters

RETURN status values	OUTPUT Parameters
Only Integers Datatype	Any Datatype
Only one value	More than one value
Use to convey success or failure	Use to return values like name, count, etc..

The following advantages of using Stored Procedures over adhoc queries (inline SQL)
1. Execution plan retention and reusability - Stored Procedures are compiled and their execution plan is cached and used again, when the same SP is executed again. Although adhoc queries also create and reuse plan, the plan is reused only when the query is textual match and the datatypes are matching with the previous call. Any change in the datatype or you have an extra space in the query then, a new plan is created.

2. Reduces network traffic - You only need to send, EXECUTE SP_Name statement, over the network, instead of the entire batch of adhoc SQL code.

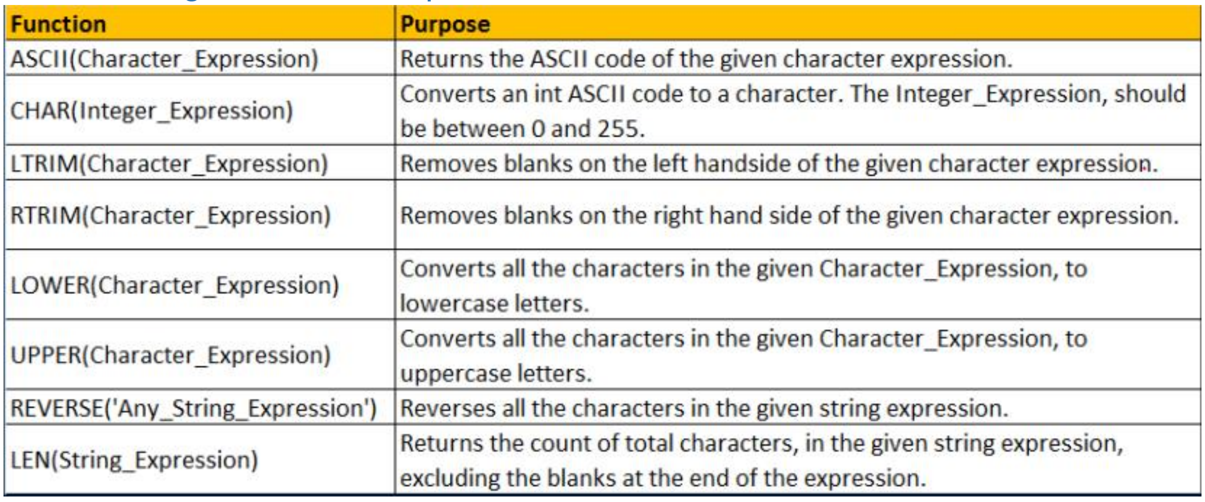
3. Code reusability and better maintainability - A stored procedure can be reused with multiple applications. If the logic has to change, we only have one place to change, where as if it is inline sql, and if you have to use it in multiple applications, we end up with multiple copies of this inline sql. If the logic has to change, we have to change at all the places, which makes it harder maintaining inline sql.

4. Better Security - A database user can be granted access to an SP and prevent them from executing direct "select" statements against a table. This is fine grain access control which will help control what data a user has access to.

5. Avoids SQL Injection attack - SP's prevent sql injection attack.

Chapter 4 – FUNCTIONS

Built in string functions in sql server

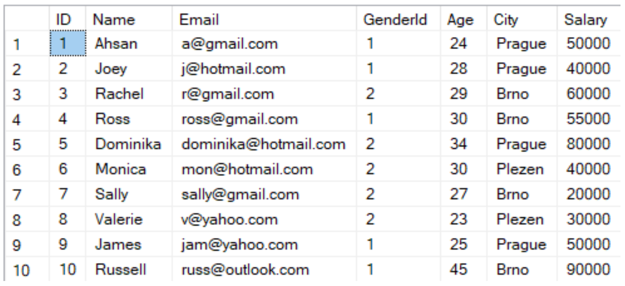


Function	Purpose
ASCII(Character_Expression)	Returns the ASCII code of the given character expression.
CHAR(Integer_Expression)	Converts an int ASCII code to a character. The Integer_Expression, should be between 0 and 255.
LTRIM(Character_Expression)	Removes blanks on the left handside of the given character expression.
RTRIM(Character_Expression)	Removes blanks on the right hand side of the given character expression.
LOWER(Character_Expression)	Converts all the characters in the given Character_Expression, to lowercase letters.
UPPER(Character_Expression)	Converts all the characters in the given Character_Expression, to uppercase letters.
REVERSE('Any_String_Expression')	Reverses all the characters in the given string expression.
LEN(String_Expression)	Returns the count of total characters, in the given string expression, excluding the blanks at the end of the expression.

String functions

Function	Purpose
LEFT(Character_Expression, Integer_Expression)	Returns the specified number of characters from the left hand side of the given character expression.
RIGHT(Character_Expression, Integer_Expression)	Returns the specified number of characters from the right hand side of the given character expression.
CHARINDEX('Expression_To_Find', 'Expression_To_Search', 'Start_Location')	Returns the starting position of the specified expression in a character string
SUBSTRING('Expression', 'Start', 'Length')	Returns substring (part of the string), from the given expression

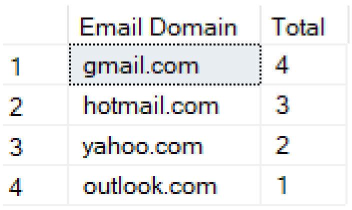
Real time example, where we can use LEN(), CHARINDEX() and SUBSTRING() functions. Let us assume we have table as shown below.



	ID	Name	Email	GenderId	Age	City	Salary
1	1	Ahsan	a@gmail.com	1	24	Prague	50000
2	2	Joey	j@hotmail.com	1	28	Prague	40000
3	3	Rachel	r@gmail.com	2	29	Brno	60000
4	4	Ross	ross@gmail.com	1	30	Brno	55000
5	5	Dominika	dominika@hotmail.com	2	34	Prague	80000
6	6	Monica	mon@hotmail.com	2	30	Plezen	40000
7	7	Sally	sally@gmail.com	2	27	Brno	20000
8	8	Valerie	v@yahoo.com	2	23	Plezen	30000
9	9	James	jam@yahoo.com	1	25	Prague	50000
10	10	Russell	russ@outlook.com	1	45	Brno	90000

Write a query to find out total number of emails, by domain. The result of the query should be as shown below.

```
SELECT
       SUBSTRING(Email, CHARINDEX('@', Email) + 1,
       LEN(Email) - CHARINDEX('@', Email)) AS [Email Domain],
       COUNT(Email) AS Total
FROM tblPerson
GROUP BY
       SUBSTRING(Email, CHARINDEX('@', Email) + 1,
       LEN(Email) - CHARINDEX('@', Email))
ORDER BY Total DESC
```



	Email Domain	Total
1	gmail.com	4
2	hotmail.com	3
3	yahoo.com	2
4	outlook.com	1

Replicate, Space, Patindex, Replace and Stuff functions

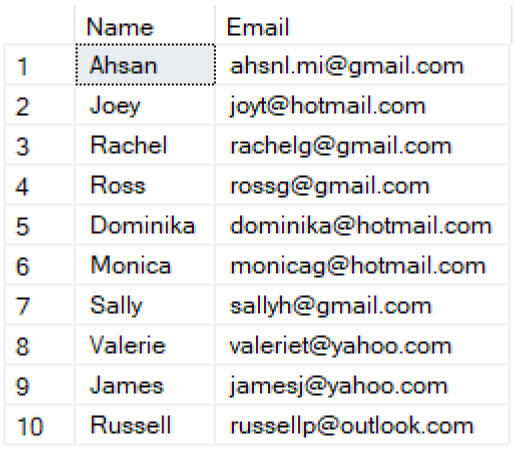
Replication Function

Definition: The REPLICATE() function repeats a string a specified number of times.

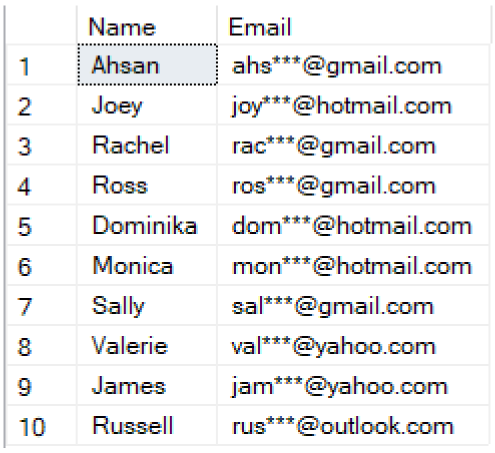
REPLICATE(*stringToBeReplicate, numberOfTImesToReplicate*)

Usage:

```
SELECT Name,
       SUBSTRING(Email, 1, 3) + REPLICATE('*',3) +
       SUBSTRING(Email, CHARINDEX('@', Email), LEN(Email) - CHARINDEX('@', Email) +1) as Email
FROM tblPerson
```



	Name	Email
1	Ahsan	ahsnl.mi@gmail.com
2	Joey	joyt@hotmail.com
3	Rachel	rachelg@gmail.com
4	Ross	rossg@gmail.com
5	Dominika	dominika@hotmail.com
6	Monica	monicag@hotmail.com
7	Sally	sallyh@gmail.com
8	Valerie	valeriet@yahoo.com
9	James	jamesj@yahoo.com
10	Russell	russellp@outlook.com



	Name	Email
1	Ahsan	ahs***@gmail.com
2	Joey	joy***@hotmail.com
3	Rachel	rac***@gmail.com
4	Ross	ros***@gmail.com
5	Dominika	dom***@hotmail.com
6	Monica	mon***@hotmail.com
7	Sally	sal***@gmail.com
8	Valerie	val***@yahoo.com
9	James	jam***@yahoo.com
10	Russell	rus***@outlook.com

Space Function

Definition: The SPACE() function returns a string of the specified number of space characters.

SPACE(*number*)

Usage:

```
SELECT Name + SPACE(10) + City as [Name and City] FROM tblPerson
```

Patindex Function

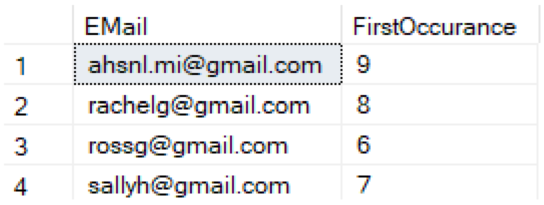
Definition:

- Returns the starting position of the first occurrence of a pattern in a specified expression. It takes two arguments, the pattern to be searched and the expression.
- PATINDEX() is similar to CHARINDEX(). With CHARINDEX() we cannot use wildcards, whereas PATINDEX() provides this capability. If the specified pattern is not found, PATINDEX() returns ZERO.

PATINDEX(*'%Pattern%', Expression*)

Usage:

```
SELECT EMail, PATINDEX('%@gmail.com', Email) as FirstOccurance
FROM tblPerson
WHERE PATINDEX('%@gmail.com', Email) > 0
```



	EMail	FirstOccurance
1	ahsnl.mi@gmail.com	9
2	rachelg@gmail.com	8
3	rossg@gmail.com	6
4	sallyh@gmail.com	7

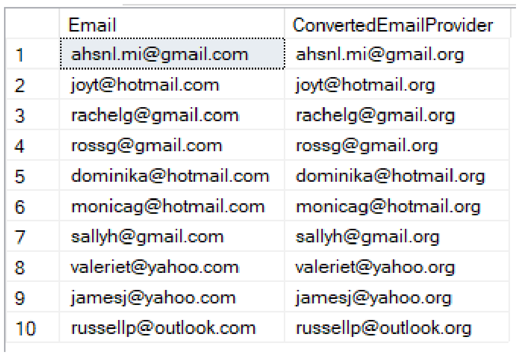
Replace Function

Definition: replaces all occurrences of a specified string values with another string value

REPLACE(*stringExpression, Pattern, replacementValue*)

Usage:

```
SELECT Email, REPLACE(Email, '.com', '.org') as
ConvertedEmailProvider
FROM tblPerson
```



	Email	ConvertedEmailProvider
1	ahsnl.mi@gmail.com	ahsnl.mi@gmail.org
2	joyt@hotmail.com	joyt@hotmail.org
3	rachelg@gmail.com	rachelg@gmail.org
4	rossg@gmail.com	rossg@gmail.org
5	dominika@hotmail.com	dominika@hotmail.org
6	monicag@hotmail.com	monicag@hotmail.org
7	sallyh@gmail.com	sallyh@gmail.org
8	valeriet@yahoo.com	valeriet@yahoo.org
9	jamesj@yahoo.com	jamesj@yahoo.org
10	russellp@outlook.com	russellp@outlook.org

Stuff Function

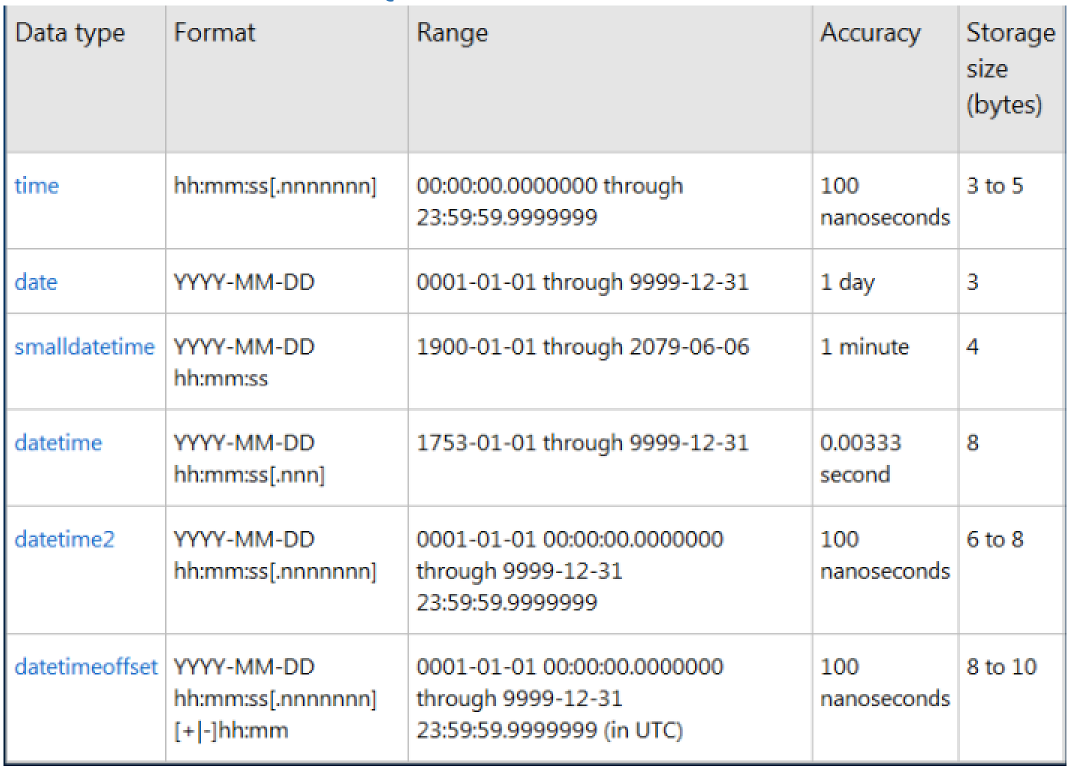
Definition: STUFF() function inserts replaceExpression, at the start position specified, along with removing the characters specified using Length parameter.

STUFF(*orginalExpression, Start, Length, replacementExpression*)

Usage:

```
Select Name,Email, STUFF(Email, 2, 3, '*****') as StuffedEmail
From tblPerson
```

DateTime functions in SQL Server



Data type	Format	Range	Accuracy	Storage size (bytes)
time	hh:mm:ss[.nnnnnnn]	00:00:00.0000000 through 23:59:59.9999999	100 nanoseconds	3 to 5
date	YYYY-MM-DD	0001-01-01 through 9999-12-31	1 day	3
smalldatetime	YYYY-MM-DD hh:mm:ss	1900-01-01 through 2079-06-06	1 minute	4
datetime	YYYY-MM-DD hh:mm:ss[.nnn]	1753-01-01 through 9999-12-31	0.00333 second	8
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999	100 nanoseconds	6 to 8
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 (in UTC)	100 nanoseconds	8 to 10

There are several built-in DateTime functions available in SQL Server. All the following functions can be used to get the current system date and time, where you have sql server installed.

Function	Date Time Format	Description
GETDATE()	2012-08-31 20:15:04.543	Commonly used function
CURRENT_TIMESTAMP	2012-08-31 20:15:04.543	ANSI SQL equivalent to GETDATE
SYSDATETIME()	2012-08-31 20:15:04.5380028	More fractional seconds precision
SYSDATETIMEOFFSET()	2012-08-31 20:15:04.5380028 + 01:00	More fractional seconds precision + Time zone offset
GETUTCDATE()	2012-08-31 19:15:04.543	UTC Date and Time
SYSUTCDATETIME()	2012-08-31 19:15:04.5380028	UTC Date and Time, with More fractional seconds precision

IsDate, Day, Month, Year and DateName, DateTime functions in SQL Server

IsDate Function

ISDATE() – Checks if the given value, is a valid date, time, or datetime. Returns 1 for success, and 0 for failure

Usage:

```
Select ISDATE('PRAGIM') -- returns 0
Select ISDATE(Getdate()) -- returns 1
Select ISDATE('2012-08-31 21:02:04.167') -- returns 1
```

Day, Month, and Year Function

DAY() - Returns the **'Day number of the Month'** of the given date

Usage:

```
Select DAY(GETDATE()) -- Returns the day number of the month, based on current system datetime.
Select DAY('01/31/2012') -- Returns 31
```

MONTH() - Returns the **'Month number of the year'** of the given date

Usage:

```
Select DAY(GETDATE()) -- Returns the day number of the month, based on current system datetime.
Select DAY('01/31/2012') -- Returns 31
```

YEAR() - Returns the **'Year number'** of the given date

DateName Function

DateName(*datePart, date*) - Returns a string, that represents a part of the given date. This functions takes 2 parameters. The first parameter **'DatePart'** specifies, the part of the date, we want. The second parameter, is the actual date, from which we want the part of the Date.

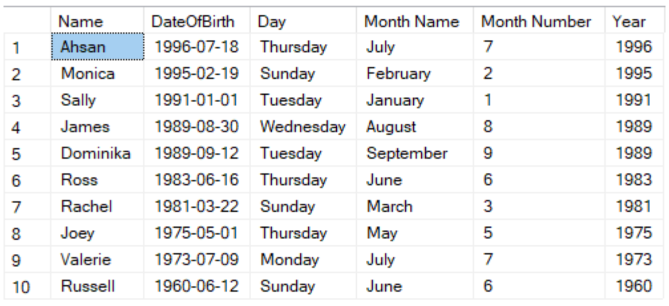
DatePart	Abbreviations
Year	yy, yyyy
Quarter	qq, q
Month	mm, m
DayOfYear	dy, y
Day	dd, d
Week	wk, ww
Weekday	dw
Hour	hh
Minute	mi, n
Second	ss, s
Millisecond	ms
Microseconds	mcs
nanoseconds	ns
TZoffset	tz

Usage:

```
Select DATENAME(Day, GETDATE()) -- Returns 30
Select DATENAME(WEEKDAY, '2012-09-30 12:43:46.837') -- Returns Sunday
Select DATENAME(MONTH, '2012-09-30 12:43:46.837') -- Returns September
```

Example: Write a query, which returns Name, DateOfBirth, Day, MonthNumber, MonthName, and Year as shown below.

```
SELECT Name, DateOfBirth,
       DATENAME(WEEKDAY, DateOfBirth) as [Day],
       DATENAME(MONTH, DateOfBirth) as [Month
Name],
       MONTH(DateOfBirth) as [Month Number],
       YEAR(DateOfBirth) as [Year]
FROM tblPerson ORDER BY [Year] DESC
```



	Name	DateOfBirth	Day	Month Name	Month Number	Year
1	Ahsan	1996-07-18	Thursday	July	7	1996
2	Monica	1995-02-19	Sunday	February	2	1995
3	Sally	1991-01-01	Tuesday	January	1	1991
4	James	1989-08-30	Wednesday	August	8	1989
5	Dominika	1989-09-12	Tuesday	September	9	1989
6	Ross	1983-06-16	Thursday	June	6	1983
7	Rachel	1981-03-22	Sunday	March	3	1981
8	Joey	1975-05-01	Thursday	May	5	1975
9	Valerie	1973-07-09	Monday	July	7	1973
10	Russell	1960-06-12	Sunday	June	6	1960

DatePart, DateAdd and DateDiff functions in SQL Server

DatePart Function

DATEPART(DatePart, Date) - Returns an integer representing the specified DatePart. This function is similar to DateName(). DateName() returns nvarchar, whereas DatePart() returns an integer.

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

DateAdd Function
DATEADD(*datePart, numberToAdd, date*) - Returns the DateTime, after adding specified NumberToAdd, to the datepart specified of the given date.

Usage:

```
Select DateAdd(MONTH, 20, '2012-08-30 19:45:31.793')
-- Returns 2012-09-19 19:45:31.793
Select DateAdd(DAY, -20, '2012-08-30 19:45:31.793')
-- Returns 2012-08-10 19:45:31.793
```

DateDiff
DATEDIFF(*datePart, startDate, endDate*) - Returns the count of the specified datepart boundaries crossed between the specified startdate and enddate.

Usage:

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

Example: Write a query to compute the age of a person, when the date of birth is given

The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL

```
--Write a query to compute the age of a person:
DECLARE @DOB dateTIME, @tempDate dateTIME, @years int, @months int, @days int
SET @DOB = '07/18/1996'

SELECT @tempDate = @DOB

SELECT @years = DATEDIFF(YEAR, @tempDate, GETDATE()) -
       CASE
              WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR
              (MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) > DAY(GETDATE()))
              THEN 1 ELSE 0
       END
SELECT @tempDate = DATEADD(YEAR, @years, @tempDate)

SELECT @months = DATEDIFF(MONTH, @tempDate, GETDATE()) -
       CASE
              WHEN (DAY(@DOB) > DAY(GETDATE()))
              THEN 1 ELSE 0
       END
SELECT @tempDate = DATEADD(MONTH, @months, @tempDate)

SELECT @days = DATEDIFF(DAY, @tempDate, GETDATE())

SELECT @years as Years, @months as Months, @days as [Days]

--Dynamic computations(Create a Function):
CREATE FUNCTION fnComputeAge(@DOB DATETIME)
RETURNS NVARCHAR(50)
AS
BEGIN
       DECLARE @tempDate DATETIME, @years int, @months int, @days int

       SELECT @tempDate = @DOB

       SELECT @years = DATEDIFF(YEAR, @tempDate, GETDATE()) -
```

```
            CASE
                   WHEN (MONTH(@DOB) > MONTH(GETDATE())) OR
                   (MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) > DAY(GETDATE()))
                   THEN 1 ELSE 0
            END
       SELECT @tempDate = DATEADD(YEAR, @years, @tempDate)

       SELECT @months = DATEDIFF(MONTH, @tempDate, GETDATE()) -
            CASE
                   WHEN (DAY(@DOB) > DAY(GETDATE()))
                   THEN 1 ELSE 0
            END
       SELECT @tempDate = DATEADD(MONTH, @months, @tempDate)

       SELECT @days = DATEDIFF(DAY, @tempDate, GETDATE())

       DECLARE @Age nvarchar(50)
       SET @Age =
                   CAST(@years AS nvarchar(4)) + ' Years ' +
                   CAST(@months AS nvarchar(2)) + ' Months ' +
                   CAST(@days AS nvarchar(2)) + ' Days old '
       RETURN @Age
END

--TO EXECUTE
SELECT ID, Name, DateOfBirth, [dbo].[fnComputeAge](DateOfBirth) AS Age FROM tblPerson
```

Cast and Convert Functions in SQL Server

Syntax of CAST and CONVERT functions from MSDN:
CAST (expression AS data_type [(length)])
CONVERT (data_type [(length)] , expression [, style])

Continue again: https://csharp-video-tutorials.blogspot.com/2012/09/cast-and-convert-functions-in-sql.html

Mathematical Function in SQL

ABS (T-SQL)

A mathematical function that returns the absolute (positive) value of the specified numeric expression. (ABS changes negative values to positive values. ABS has no effect on zero or positive values.)

Syntax: ABS(*numericExpression*)

CEILING/FLOOR (T-SQL)

CEILING - This function returns the smallest integer greater than, or equal to, the specified numeric expression.

FLOOR - returns the largest integer less than or equal to the parameter.

Syntax:
CEILING(*numericExpression*)
FLOOR(*numericExpression*)

Examples:
Select CEILING(15.2) -- Returns 16
Select CEILING(-15.2) -- Returns -15

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

POWER, SQUARE, SQRT

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

Example: The following example calculates '2 TO THE POWER OF 3' = 2*2*2 = 8
Select POWER(2,3) -- Returns 8

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

Example:
Select SQUARE(9) -- Returns 81

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

Example:
Select SQRT(81) -- Returns 9

RAND() functions

RAND([*seedValue*]) – returns a random float number between 0 and 1. Rand() function takes an optional seed parameter. When seed value is supplied the RAND() function, it always return the same value for the same seed

E.g:

```
--RAND() function:
SELECT RAND(1)
-- Generate a random number between 1-10:
DECLARE @Counter INT
SET @Counter = 1
WHILE (@Counter <= 10)
BEGIN
       PRINT FLOOR(RAND() * 10)
       SET @Counter += 1
END
```

ROUND() function

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

1. *numericExpression* – is the number that we want to round
2. *Length* parameters specifies the number of the digits that we want to round to. If the length is a positive number, then the rounding is applied for the decimal part, whereas if the length is negative, then the rounding is applied to the number before the decimal part
3. *The optional function parameter* – is used to indicate rounding or *truncation* operations. **TRUNCATE** TABLE removes all rows from a table, but the table structure and its columns, constraints, indexes, and so on remain.
   a. Zero – indicate rounding
   b. Non-zero – indicate truncation.

Example:

```
-- Round to 2 places after (to the right) the decimal point
Select ROUND(850.556, 2) -- Returns 850.560

-- Truncate anything after 2 places, after (to the right) the decimal point
Select ROUND(850.556, 2, 1) -- Returns 850.550

-- Round to 1 place after (to the right) the decimal point
Select ROUND(850.556, 1) -- Returns 850.600

-- Truncate anything after 1 place, after (to the right) the decimal point
Select ROUND(850.556, 1, 1) -- Returns 850.500

-- Round the last 2 places before (to the left) the decimal point
Select ROUND(850.556, -2) -- 900.000

-- Round the last 1 place before (to the left) the decimal point
Select ROUND(850.556, -1) -- 850.000
```

Scalar user defined functions

In SQL Server there are 3 types of User Defined functions
1. Scalar functions
2. Inline table-valued functions
3. Multistatement table-valued functions

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

To create function, use the following Syntax:

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

Example: create a function which calculates and returns the age of a person

```
CREATE FUNCTION fnCalcAge(@DOB DATE)
RETURNS INT
AS
BEGIN
       --FUNCTION BODY
       DECLARE @Age INT
       SET @Age = DATEDIFF(YEAR, @DOB, GETDATE()) -
                            CASE
                                   WHEN   (MONTH(@DOB) > MONTH(GETDATE())) OR
                                          (MONTH(@DOB) = MONTH(GETDATE()) AND DAY(@DOB) >
DAY(GETDATE()))
                                   THEN 1
                                   ELSE 0
                            END
       RETURN @Age
END

SELECT ID, Name, DateOfBirth, [dbo].[fnCalcAge](DateOfBirth) AS Age FROM tblPerson
```

Inline table value functions

Definition: Scalar function, returns a single value while Inline Table function return a table.

Usage/Syntax for *Inline Table Value Function:*

```
CREATE FUNCTION functionName(@param1 dataType, @param2 dataType..., @paramN dataType)
RETURNS TABLE
AS
RETURN (selectStatement)
```

Example: Create Inline Table Function

```
CREATE FUNCTION fn_EmployeeByGender (@GenderId int)
RETURNS TABLE
AS
RETURN (
       SELECT ID, Name, DateOfBirth, GenderId, DepartmentId
       FROM tblPerson
       WHERE GenderId = @GenderId
)
```

This function is similar to SCALAR function based on:

1. The type of RETURN function. Instead of DataType, we return a Table
2. The function body is not enclose with BEGIN and END statement (block)
3. The *structure of the table* gets returned by the SELECT statement

Calling the function:

```
SELECT * FROM fn_EmployeeByGender('1') WHERE Name = 'Ahsan'

-- joins table based of Keys:
```

JOINING the Gender returned by the function, with department table:

```
SELECT Name, Gender
FROM fn_EmployeeByGender('1') E
JOIN tblGender G
ON      G.ID = E.GenderId
```

Multi-Statement Table Valued Functions

Differences between MTVF and ITVF

	ITVF	MSTVF
The RETURNS syntax	Simply state the RETURNS TABLE and the return table's definition will be based on the function's SELECT statement. No need to specify the structure of the return table	Your RETURNS syntax explicitly specifies the structure of the return table. This is done by declaring a TABLE variable that will be used to stored and accumulate the rows that are returned as the value of the function
The BEGIN/END syntax	Do not use	Use
Performance	Fast	Slow
Data updates	Possibility of updating the underlying table	Cannot be updated once created

Stack Overflow Questions.

1. How to insert columns at a specific position in existing table?

Not possible in MS SQL, only MySQL