Advance Databases and SQL Querying

Stored Procedures:

A stored procedure is a set of [Structured Query Language (SQL)](https://searchsqlserver.techtarget.com/definition/SQL) statements with an assigned name, which are stored in a [relational database management system](https://searchdatamanagement.techtarget.com/definition/RDBMS-relational-database-management-system) as a group, so it can be reused and shared by multiple programs.

Benefits of Stored Procedures:

Stored procedures offer advantages over embedding [queries](https://searchsqlserver.techtarget.com/definition/query) in a [graphical user interface (GUI)](https://searchwindevelopment.techtarget.com/definition/GUI). Since stored procedures are modular, it is easier to troubleshoot when a problem arises in an application. Stored procedures are also tunable, which eliminates the need to modify the GUI [source code](https://searchmicroservices.techtarget.com/definition/source-code) to improve its performance. It's easier to code stored procedures than to build a query through a GUI. Stored procedures can access or modify [data](https://searchdatamanagement.techtarget.com/definition/data) in a [database](https://searchsqlserver.techtarget.com/definition/database)

Use of stored procedures can reduce network traffic between clients and [servers](https://whatis.techtarget.com/definition/server), because the commands are executed as a single batch of code. This means only the call to execute the procedure is sent over a [network](https://searchnetworking.techtarget.com/definition/network), instead of every single line of code being sent individually.We can create the check for SP, which user can see which SP.

Syntax:

Create Procedure MyTestProc

AS

Set NOCOUNT ON

Select \* from [HumanResource].[shift]

To Execute SP:

Exec MyTestProc

To Drop SP:

Drop Proc MyTestProc

Parameterized SP

Create Procedure MyFIrstParaProc

@paramname varchar(50)

AS

Set NOCOUNT ON

Select \* from [HumanResource].[shift] where Name= paramname

To Execute SP:

Exec MyFIrstParaProc @ paramname =’Dev’

* For multiple parameters the sequence of parameters should be same as in proc
* If dev fails to pass parameter it will give you an error,But to avoid it you can pass the default values such as

Create Procedure MyFIrstParaProc

@paramname varchar(50) = ‘Default’

AS

Set NOCOUNT ON

Select \* from [HumanResource].[shift] where Name= paramname

To Execute SP with parameters:

Exec MyFIrstParaProc @ paramname =’Dev’

Output parameters:

This parameters returns the value from the stored procedure

Create Procedure MyFIrstoutParaProc

@topshift varchar(50) output

AS

Set @topshift =Select top 1 (ShiftId) from [HumanResource].[shift]

Display Result:

Declare @outputResult varchar(50)

Exec MyFIrstoutParaProc @outputResult output

Select @ outputResult

UDF (User Defined Functions)

Like functions in programming languages, SQL Server user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

Benefits:

* They allow modular programming.

You can create the function once, store it in the database, and call it any number of times in your program. User-defined functions can be modified independently of the program source code.

* They allow faster execution.

Similar to stored procedures, Transact-SQL user-defined functions reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the user-defined function does not need to be reparsed and reoptimized with each use resulting in much faster execution times.

CLR functions offer significant performance advantage over Transact-SQL functions for computational tasks, string manipulation, and business logic. Transact-SQL functions are better suited for data-access intensive logic.

* They can reduce network traffic.

An operation that filters data based on some complex constraint that cannot be expressed in a single scalar expression can be expressed as a function. The function can then invoked in the WHERE clause to reduce the number or rows sent to the client.

SQL User defined functions

Types:

1. Scalar Value : Return single data value

User-defined scalar functions return a single data value of the type defined in the RETURNS clause. For an inline scalar function, there is no function body; the scalar value is the result of a single statement. For a multistatement scalar function, the function body, defined in a BEGIN...END block, contains a series of Transact-SQL statements that return the single value. The return type can be any data type except **text**, **ntext**, **image**, **cursor**, and **timestamp.**

Syntax**:**

Create Function myScaleFunc()

Returns Money AS

Begin

Declare @ytdSales Money

Select @ ytdSales=sum(YTDSales) FROM Sales.SalesTerritory

Return @ ytdSales

End

Show result:

Declare @ytdResults AS Money

Select @ytdResults = dbo. myScaleFunc()

Print @ytdResults

Parameterized Function:

Create Function myParamScaleFunc(@group Varchar(50))

Returns Money AS

Begin

Declare @ytdSales Money

Select @ ytdSales=sum(YTDSales) FROM Sales.SalesTerritory where [Group] = @group

Return @ ytdSales

End

Show result:

Declare @ytdResults AS Money

Select @ytdResults = dbo. myParamScaleFunc(‘india’)

Print @ytdResults

1. Table Value : return data into tabular format.

User-defined table-valued functions return a **table** data type. For an inline table-valued function, there is no function body; the table is the result set of a single SELECT statement

Create Function myParamTabFunc(@TerritoryId int)

Returns Table AS

Return

Select \* FROM Sales.SalesTerritory where [TerritoryId] = @ TerritoryId

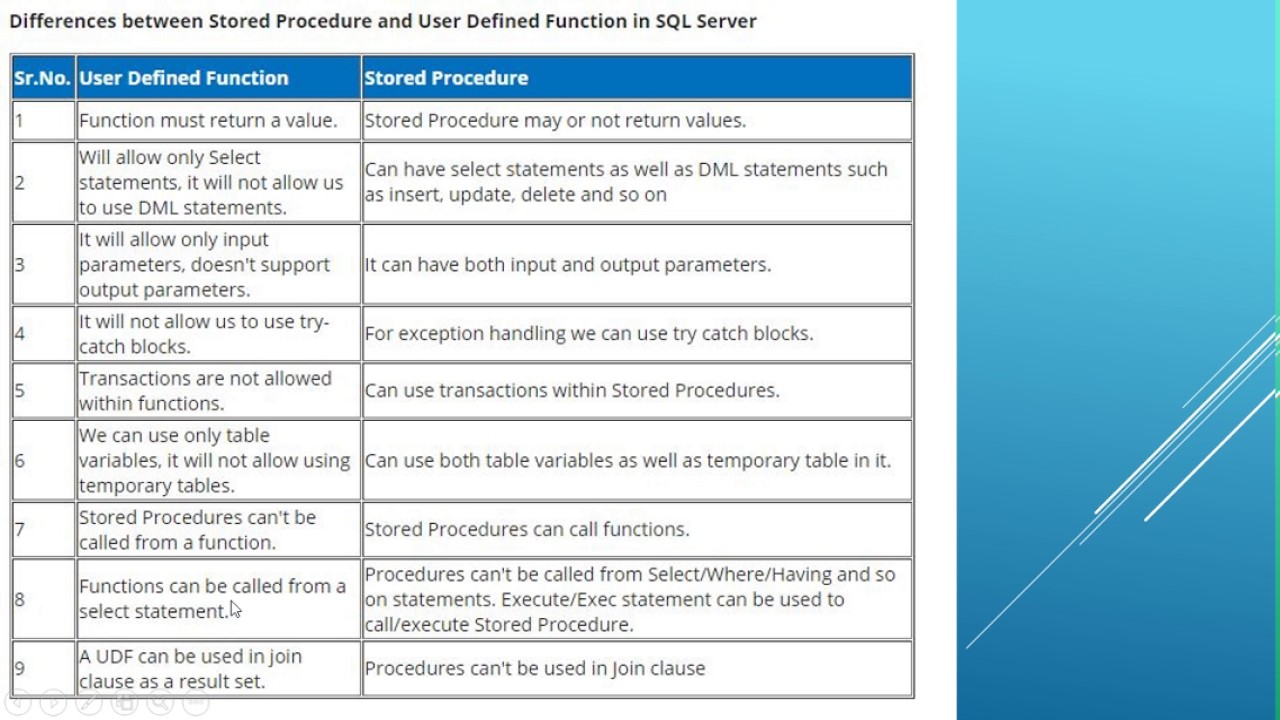
Result :

Select \* from dbo. myParamTabFunc(7)

1. System Function : System functions which are used to perform various operations.

SQL Server provides many system functions that you can use to perform a variety of operations. They cannot be modified.

Difference between SP and UDF



Triggers:

SQL Server triggers are special [stored procedures](http://www.sqlservertutorial.net/sql-server-stored-procedures/) that are executed automatically in response to the database object, database, and server events. SQL Server provides three type of triggers:

Data manipulation language (DML) triggers which are invoked automatically in response to [INSERT](http://www.sqlservertutorial.net/sql-server-basics/sql-server-insert/), [UPDATE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-update/), and [DELETE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-delete/) events against tables.

Data definition language (DDL) triggers which fire in response to [CREATE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-create-table/), ALTER, and [DROP](http://www.sqlservertutorial.net/sql-server-basics/sql-server-drop-table/)statements. [DDL triggers](http://www.sqlservertutorial.net/sql-server-triggers/sql-server-ddl-trigger/) also fire in response to some system stored procedures that perform DDL-like operations.

Logon triggers which fire in response to LOGON events.

The CREATE TRIGGER statement allows you to create a new trigger that is fired automatically whenever an event such as [INSERT](http://www.sqlservertutorial.net/sql-server-basics/sql-server-insert/), [DELETE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-delete/), or [UPDATE](http://www.sqlservertutorial.net/sql-server-basics/sql-server-update/) occurs against a table.

The following illustrates the syntax of the CREATE TRIGGER statement:

|  |
| --- |
| CREATE TRIGGER [schema\_name.]trigger\_name  ON table\_name  AFTER  {[INSERT],[UPDATE],[DELETE]}  [NOT FOR REPLICATION]  AS  {sql\_statements} |

Transaction and error handling:

A transaction is a single unit of work. If transaction is successfully, the data will be committed and become a permanent part of the database else if any error occurs then all the modifications will be roll backed.

Properties:

ACID:

1. Atomicity: It requires that each transaction will be all or nothing
2. Consistency: Any transaction would bring the database from one valid state to another.
3. Integrity: Concurrent execution of transaction results in a system state that would be obtained if transactions were executed serially, one after another. Exclusive or rollover lock----
4. Durability: Once a transaction is committed, the data will remain forever in database.

Examples :

Select \* FROM Sales.SalesTerritory

Begin Transaction

Update Sales.SalesTerritory Set CostY=1 where TerritoryId=1

Commit Transaction

Declare @ErrorResult Varchar(50)

Begin Transaction

Insert Into Sales.SalesTerritory(name,group,rowguid,modifieddate)

Values

(‘anagha’,’BFSI’,’swuyyt5tyffhg65re6’,getdate())

SET @ErrorResult= @@ERROR

If (@ErrorResult =0)

BEGIN

PRINT ‘Successful!!!’

COMMIT Transaction

End

Else

Begin

Print ‘Transaction Failed‘

Rollback Transaction

End

Custom error Message –

Select \* FROM Sales.SalesTerritory

Begin Transaction

Update Sales.SalesTerritory Set CostY=1 where TerritoryId=1

Commit Transaction

Declare @ErrorResult Varchar(50)

Begin Transaction

Insert Into Sales.SalesTerritory(name,group,rowguid,modifieddate)

Values

(‘anagha’,’BFSI’,’swuyyt5tyffhg65re6’,getdate())

SET @ErrorResult= @@ERROR

If (@ErrorResult =0)

BEGIN

PRINT ‘Successful!!!’

COMMIT Transaction

End

Else

Begin

Raiserror(‘Statement Failed – custom message’,16,1)

Rollback Transaction

End

Try and Catch

Implements error handling for Transact-SQL that is similar to the exception handling in the Microsoft Visual C# and Microsoft Visual C++ languages. A group of Transact-SQL statements can be enclosed in a TRY block. If an error occurs in the TRY block, control is passed to another group of statements that is enclosed in a CATCH block.

Syntax:

BEGIN TRY

{ sql\_statement | statement\_block }

END TRY

BEGIN CATCH

[ { sql\_statement | statement\_block } ]

END CATCH

[ ; ]

Begin Try

Begin Transaction

Select \* FROM Sales.SalesTerritory

Begin Transaction

Update Sales.SalesTerritory Set CostY=1 where TerritoryId=1

Commit Transaction

Declare @ErrorResult Varchar(50)

Begin Transaction

Insert Into Sales.SalesTerritory(name,group,rowguid,modifieddate)

Values

(‘anagha’,’BFSI’,’swuyyt5tyffhg65re6’,getdate())

COMMIT Transaction

End Try

Begin Catch

PRINT ‘Catch Statement Entered’

Rollback Transaction

END Catch

Deadlock:

A blocking situation can be understood with the following example.

Let’s assume that we have two database users already connected to SQL Server using an application: **UserA** and **UserB**.

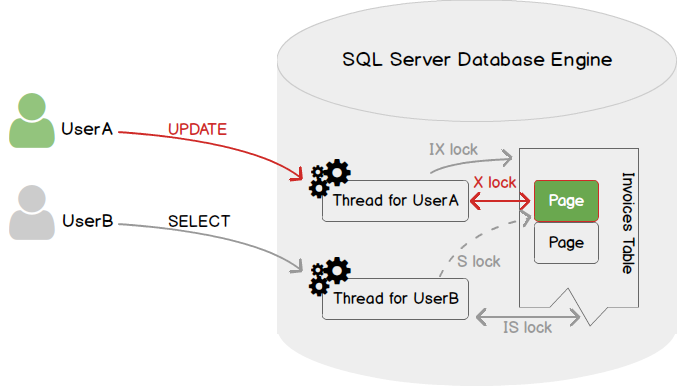
UserA is currently editing an invoice, which implies an **UPDATE** statement against an **Invoice** table with a **WHERE** clause that restricts to a particular value of **InvoiceId** column of that table. To perform this operation, the thread associated to that session inside SQL Server database engine has to acquire and hold:

* An [Intent-Exclusive](https://technet.microsoft.com/en-us/library/ms175519%28v=sql.105%29.aspx) (IX) lock on the **Invoice** table and on the page, that contain the row **userA** is editing. This lock is used to establish a lock hierarchy in order to perform data modifications.
* An [Exclusive](https://technet.microsoft.com/en-us/library/ms175519%28v=sql.105%29.aspx) (X) lock on the row **userA** is editing. This means that the session will be the only one allowed to modify that row until it releases this lock.

At the same time, **UserB** wants to get a list of the invoices for current month and unfortunately, the invoice **UserA** is editing sits in that list. **UserB**’s thread will:

* Acquire an [Intent Shared](https://technet.microsoft.com/en-us/library/ms175519%28v=sql.105%29.aspx) (IS) lock on the **Invoice** table. This lock is used to establish a lock hierarchy in order to perform read-only operations. This will work as IX and IS on a table are [compatible](https://technet.microsoft.com/en-us/library/ms186396%28v=sql.105%29.aspx).
* Try to attempt a [shared](https://technet.microsoft.com/en-us/library/ms175519%28v=sql.105%29.aspx) (S) lock on the pages needed to display the list. Among them, the page with an X lock acquired by **UserA**. However, shared and exclusive locks are incompatible (even semantically). This means that thread taking care of **UserB**’s session has to wait for **UserA**’s session to release this X lock. Until that happen, we can say that **UserB**’s session is **blocked** by **UserA**’s session.

The situation can be graphically summarized as follows:



This situation will end as soon as the **UPDATE** query has been completed and **UserA** has committed their transaction.

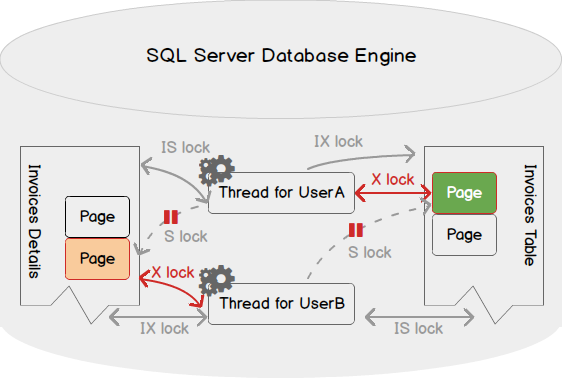
As we may expect, we can encounter a lot more complex situations, involving a session holding multiple locks on multiple resources.

Although it’s based on the same principles, deadlocks are a different from blocking. Actually, when a deadlock situation happens, there is no identifiable head blocker as both sessions implied holds incompatible locks on objects the other session needs to access. It’s a circular blocking chain.

For better understanding, we will go back to the situation we used for blocking presentation and add some complexity to that situation.

Let’s say that in order to modify a row in **Invoice** table, UserA must also read from an **InvoiceDetails** table to get the total that is billed to customer. Let’s say that, no matter the reason, **UserB** has already acquired an exclusive lock on a page containing a row of **InvoiceDetails** table that **UserA** needs to read.

In such a case, we are in the situation depicted by following figure. (As a reminder, green is used to refer to **UserA** and orange for **UserB**)



As you can see in the figure above, both threads are waiting for a lock that won’t be ever released as the activity of one is suspended until the other releases its acquired locks. There can be more complicated in real life situations.

Here is some further information about the deadlock monitor thread:

* It runs every 5 seconds by default
* When it detects a deadlock, this interval falls from 5 seconds to as low as 100 milliseconds based on frequency of deadlock occurrences
* When it finally finds no deadlock, it put the interval to its default of 5 seconds
* Once the deadlock victim is chosen, it will roll back the transaction of this victim and return a 1205 error message to user. The error message looks like follows

Transaction (Process ID 89) was deadlocked on resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

* By default, the deadlock victim is chosen based on the estimated amount of resource consumption for rolling back. It’s the least expensive one that is chosen. We can use SET DEADLOCK\_PRIORITY <Value>statement to influence the choice of deadlock victim.

SQL Grouping Sets and Common Table Expressions(CTE) :

SQL Grouping Sets:

By definition, a grouping set is a group of columns by which you group. Typically, a single query with an aggregate defines a single grouping set.

Common Table Expressions(CTE): Specifies a temporary named result set, known as a common table expression (CTE). This is derived from a simple query and defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE or MERGE statement. This clause can also be used in a CREATE VIEW statement as part of its defining SELECT statement. A common table expression can include references to itself. This is referred to as a recursive common table expression.

Example:

Select \* FROM Sales.SalesTerritory

WITH CTE\_Sales as

Select Name, CountryCoderegion FROM Sales.SalesTerritory

Select \* from CTE\_Sales where name like ‘North%’

Groupby

Select name,NULL,NULL,sum(SaleyId) FROM Sales.SalesTerritory Group by Name

Union ALL

Select name, CountryCoderegion,NULL,sum(SaleyId) FROM Sales.SalesTerritory

Group by Name, CountryCoderegion

Union ALL

Select name, CountryCoderegion,[Group] ,sum(SaleyId) FROM Sales.SalesTerritory

Group by Name, CountryCoderegion,[Group]

To avoid these much complexity we use Grouping Sets

Select name, CountryCoderegion,[Group] ,sum(SaleyId) FROM Sales.SalesTerritory

Group by Grouping Sets (Name),

(Name,CountryCoderegion),

(Name,CountryCoderegion,[Group])

Here also we had written too much code like name then again name etc. To avoid these we use Rollup

Select name, CountryCoderegion,[Group] ,sum(SaleyId) FROM Sales.SalesTerritory

Group by ROLLUP (Name,CountryCoderegion,[Group])

Cube:

Select name, CountryCoderegion,[Group] ,sum(SaleyId) FROM Sales.SalesTerritory

Group by Cube (Name,CountryCoderegion,[Group])

SQL Ranking Functions :

It allow us to rank the records depending upon what criteria we have.

Ranking functions return a ranking value for each row in a partition. Depending on the function that is used, some rows might receive the same value as other rows. Ranking functions are nondeterministic.

* Rank (): Returns the rank of each row within the partition of a result set. The rank of a row is one plus the number of ranks that come before the row in question.
* ROW\_NUMBER and RANK are similar. ROW\_NUMBER numbers all rows sequentially (for example 1, 2, 3, 4, 5). RANK provides the same numeric value for ties (for example 1, 2, 2, 4, 5).
* Dense\_Rank():This function returns the rank of each row within a result set partition, with no gaps in the ranking values. The rank of a specific row is one plus the number of distinct rank values that come before that specific row.
* Ntile():Distributes the rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one. For each row, NTILE returns the number of the group to which the row belongs.

Select \* from [person].[Address]

Select PostalCode,Row\_number() over (order by PostalCode) as ‘Row\_Number’,

Rank() over (order by PostalCode) as ‘Rank’ ,

Dense\_Rank()over (order by PostalCode) as ‘Dense Rank’ ,

Ntile(10) over (order by PostalCode) as ‘Ntile’

from [person].[Address]

WHERE PostalCode IN(98055,98027,98052,97205)

Special data

SQL XML data types-

If your data is highly structured with known schema, the relational model is likely to work best for data storage. SQL Server provides the required functionality and tools you may need. On the other hand, if the structure is semi-structured or unstructured, or unknown, you have to give consideration to modeling such data.

XML is a good choice if you want a platform-independent model in order to ensure portability of the data by using structural and semantic markup. Additionally, it is an appropriate option if some of the following properties are satisfied:

* Your data is sparse or you do not know the structure of the data, or the structure of your data may change significantly in the future.
* Your data represents containment hierarchy, instead of references among entities, and may be recursive.
* Order is inherent in your data.
* You want to query into the data or update parts of it, based on its structure.

If none of these conditions is met, you should use the relational data model. For example, if your data is in XML format but your application just uses the database to store and retrieve the data, an [n]varchar(max) column is all you require. Storing the data in an XML column has additional benefits. This includes having the engine determine that the data is well formed or valid, and also includes support for fine-grained query and updates into the XML data.

Reasons for Storing XML Data in SQL Server

Following are some of the reasons to use native XML features in SQL Server instead of managing your XML data in the file system:

* You want to share, query, and modify your XML data in an efficient and transacted way. Fine-grained data access is important to your application. For example, you may want to extract some of the sections within an XML document, or you may want to insert a new section without replacing your whole document.
* You have relational data and XML data and you want interoperability between both relational and XML data within your application.
* You need language support for query and data modification for cross-domain applications.
* You want the server to guarantee that the data is well formed and also optionally validate your data according to XML schemas.
* You want indexing of XML data for efficient query processing and good scalability, and the use of a first-rate query optimizer.
* You want SOAP, ADO.NET, and OLE DB access to XML data.
* You want to use administrative functionality of the database server for managing your XML data. For example, this would be backup, recovery, and replication.

If none of these conditions is satisfied, it may be better to store your data as a non-XML, large object type, such as [n]varchar(max) or varbinary(max).

To fill data into Table from XML

1. Create table with column type xml
2. Insert data into table with insert query

Insert into XMLdatatable (xmldata) values (‘<note><to>Tove</to><from>Jani</from><heading>Reminder</heading><body>Dont forget me this weekend!</body></note>’)

To convert data in table to XML

Select \* FROM Sales.SalesTerritory

For xml auto,elements,root(‘Salesterritory’)

Select \* FROM Sales.SalesTerritory

For xml raw,elements,root(‘Salesterritory’)

To fetch particular data from the XML in table

Select [xmldata].query(‘\node\to’) as [TO] from XMLdatatable

Just want the actual value from the tag

Select [xmldata].value(‘(\node\to)[1]’,’varchar(10)’) as [TO] from XMLdatatable

Give me top 10 territoryId from salesterritory in XML

Select top 10 territoryId FROM Sales.SalesTerritory

For xml auto,elements,root(‘Salesterritory’)

Represent as XML and display it in Tabular format:

Select \* FROM Sales.SalesTerritory

For xml auto,elements,root(‘Salesterritory’)

Declare @xmlhandle int

Declare @xmldocument xml

Set @xmldocument= Select \* FROM Sales.SalesTerritory

For xml auto,elements,root(‘Salesterritory’)

Exec sp\_xml\_preparedocument @xmlhandle output, @xmldocument

Select \* from openxml(@xmlhandle,’salesTerritory/sales.salesTerritory’,2)

With (TerritoryId int,SalesYtd Money)

Exec sp\_xml\_removedocument @xmlhandle

SQL Partitions :

Why is partitioning important??

Limiting the scope of data retrieval and speeds up your query as well as maintains your datatable properly.

Partitioning is not supported in SQL Express works in Enterprise edition.

Create DB as [DB Name]

Create partition function cust\_part\_func(int)

As range right

For values(1000,2000,3000,4000)

Create partition scheme cust\_part\_scheme

As partition cust\_part\_func

To(fgp1,fgp2,fgp3,fgp4,fgp5,fgp6)

1. Right click on db 🡪 properties 🡪 filegroup
2. Add file group as fgp1,fgp2,fgp3,fgp4,fgp5,fgp6
3. Add files as fg1,fg2,fg3,fg4,fg5,fg6
4. Select the file type of above files as fgp1,fgp2,fgp3,fgp4,fgp5,fgp6
5. We can see that the mdf named with the name as fgp1,fgp2,fgp3,fgp4,fgp5,fgp6

Create table partition

(EmpId int identity(1,1) not null,

EmpDate Dattime,null)

On cust\_part\_func(EmpId)

Declare @i int

Set @i=0

While i<10000

Begin

Insert into [partition].(EmpDate) values (getdate())

Set @i=@i+1

End

Select $partition cust\_part\_func(EmpId) as [PartitionNumber],\* from partition

There is aUI technige also

Click on the table 🡪 Storage🡪

Create partition,scheme etc.

Dynamic Queries and Pivots

A string that you generate, and when we execute it, it will work as a query.

In some applications, having hard coded SQL statements is not appealing because of the dynamic nature of the queries being issued against the database server. Because of this sometimes there is a need to dynamically create a SQL statement on the fly and then run that command.

SQL Server offers a few ways of running a dynamically built SQL statement. Here are a few options:

1. Writing a query with parameters
2. Using [EXEC](https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/execute-a-stored-procedure?view=sql-server-2017)
3. Using [sp\_executesql](https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-executesql-transact-sql?view=sql-server-2017" \t "_blank)

Pivot:

PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output, and runs aggregations where they're required on any left over column values that are wanted in the final output. UNPIVOT carries out the opposite operation to PIVOT by rotating columns of a table-valued expression into column values.

Select \* FROM Sales.SalesTerritory

Select CountryRegionCode,[Group], salesYTd FROM Sales.SalesTerritory

We want result as :

---COuntryRegionCode NorthAmerica Europe

US 23 --

Pivot:

Select CountryRegionCode,[NorthAmerica], [Europe ] FROM Sales.SalesTerritory

Pivot(

sum(SalesYtd) for [group]

in ([NorthAmerica], [Europe ],[Pacific])

) as pvt

Dynamic query:

Declare @sqlstring varchar(2000)

@sqlstring=’ Select CountryRegionCode,[Group],’

@sqlstring=@sqlstring + ‘salesYTd FROM Sales.SalesTerritory’

Print @sqlstring

Exec (@sqlstring)

SQL FileStream

FILESTREAM enables SQL Server-based applications to store unstructured data, such as documents and images, on the file system. Applications can leverage the rich streaming APIs and performance of the file system and at the same time maintain transactional consistency between the unstructured data and corresponding structured data.

FILESTREAM integrates the SQL Server Database Engine with an NTFS or ReFS file systems by storing varbinary(max) binary large object (BLOB) data as files on the file system. Transact-SQL statements can insert, update, query, search, and back up FILESTREAM data. Win32 file system interfaces provide streaming access to the data.

FILESTREAM uses the NT system cache for caching file data. This helps reduce any effect that FILESTREAM data might have on Database Engine performance. The SQL Server buffer pool is not used; therefore, this memory is available for query processing.

FILESTREAM is not automatically enabled when you install or upgrade SQL Server. You must enable FILESTREAM by using SQL Server Configuration Manager and SQL Server Management Studio. To use FILESTREAM, you must create or modify a database to contain a special type of filegroup. Then, create or modify a table so that it contains a varbinary(max) column with the FILESTREAM attribute. After you complete these tasks, you can use Transact-SQL and Win32 to manage the FILESTREAM data.

When to Use FILESTREAM

In SQL Server, BLOBs can be standard varbinary(max) data that stores the data in tables, or FILESTREAM varbinary(max) objects that store the data in the file system. The size and use of the data determines whether you should use database storage or file system storage.

If the following conditions are true, you should consider using FILESTREAM:

* Objects that are being stored are, on average, larger than 1 MB.
* Fast read access is important.
* You are developing applications that use a middle tier for application logic.

For smaller objects, storing varbinary(max) BLOBs in the database often provides better streaming performance.

### Transact-SQL Access

By using Transact-SQL, you can insert, update, and delete FILESTREAM data:

* You can use an insert operation to prepopulate a FILESTREAM field with a null value, empty value, or relatively short inline data. However, a large amount of data is more efficiently streamed into a file that uses Win32 interfaces.
* When you update a FILESTREAM field, you modify the underlying BLOB data in the file system. When a FILESTREAM field is set to NULL, the BLOB data associated with the field is deleted. You cannot use a Transact-SQL chunked update, implemented as UPDATE\*\*.\*\*Write(), to perform partial updates to the data.
* When you delete a row or delete or truncate a table that contains FILESTREAM data, you delete the underlying BLOB data in the file system.

Geometry (for 2 dimention) and Geography (for 3 dimentions) Datatype

SQL server furnishes us with the geometry and geography data types for conserving spatial data, which allows us to render graphical data. To be more specific, it is beneficial for creating, analyzing, comparing and retrieving spatial data.

Spatial data, also known as geospatial data, is a particular type of information about either the physical object or physical location of data that can be constituted by numerical values in geographic collaborate systems.

Generally, the use of spatial data is for representing the location, size, and shape of an object on planet Earth such as a mountain, lake, flat ground, township or so forth. On the top of that, spatial data may include attributes which can be given more information of the entity that is being represented.

Geographic Information Systems (GIS) or other specialized software applications should be appropriate to access, manipulate, visualize and dip analyze geospatial data.

There are two major supported data-type is SQL server namely geometry data type and geography data type.

* Geometry spatial data type

It is substantially a two-dimensional rendering of an object and also useful in case of represented as points on a planar, or flat-earth data. A good example of it is (10, 2) where the first number ‘10’ identifies that point’s position on the horizontal (x) axis and the number ‘2’ represents the point’s position on the vertical axis (y).

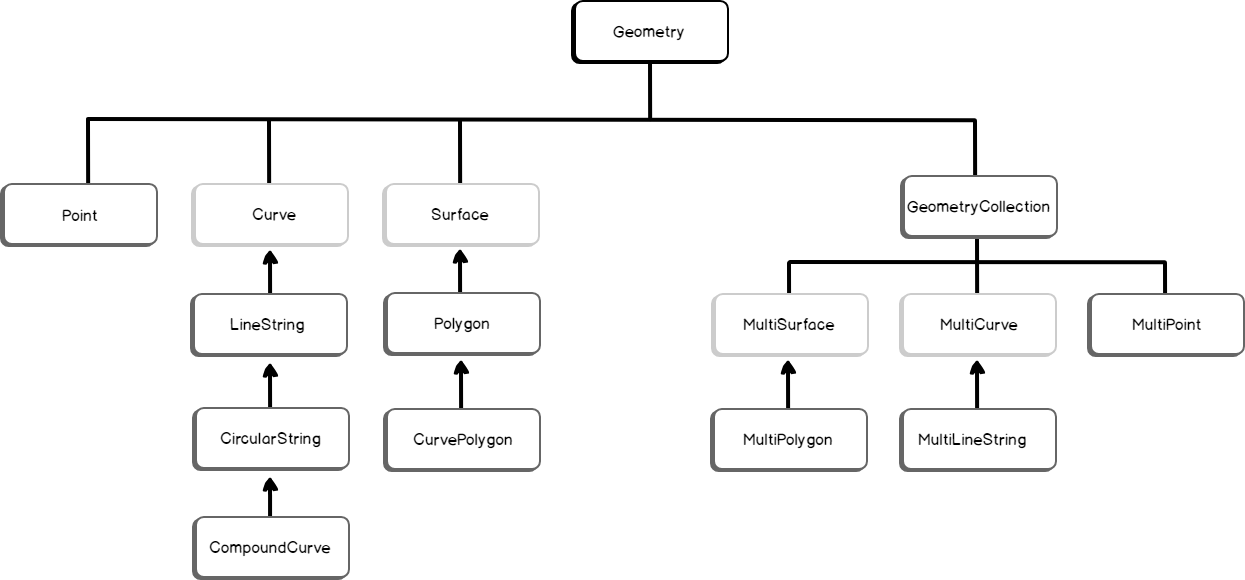
A common use case of the Geometry type is for a three-dimensional object, such as a building.

* Geography spatial data types

These are represented as latitudinal and longitudinal degrees, as on a round-earth coordinate system.

The common use case of the Geography type is to store an application’s GPS data.

In SQL Server, both data types have been implemented in the .NET common language runtime (CLR).



The following example is for creating a geometry instance.

|  |
| --- |
| DECLARE @g geometry;    SET @g = geometry::STPointFromText('POINT (100 100)', 0);    SELECT @g.ToString(); |

SQL Server return type: geography

If you define the same instance using the STGeomFromText() method,

|  |
| --- |
| DECLARE @g geometry;    SET @g = geometry::STGeomFromText('LINESTRING (100 100, 20 180, 180 180)', 0);    SELECT @g.ToString();  Refrence link: <https://www.sqlshack.com/spatial-data-types-in-sql-server/> |