SQL Server new data types

* **Date and Time**: Four new date and time data types have been added, making working with time much easier than it ever has in the past. They include: DATE, TIME, DATETIME2, and DATETIMEOFFSET.
* **Spatial**: Two new spatial data types have been added--GEOMETRY and GEOGRAPHY--which you can use to natively store and manipulate location-based information, such as Global Positioning System (GPS) data.
* **HIERARCHYID**: The HIERARCHYID data type is used to enable database applications to model hierarchical tree structures, such as the organization chart of a business.
* **FILESTREAM**: FILESTREAM is not a data type as such, but is a variation of the VARBINARY(MAX) data type that allows unstructured data to be stored in the file system instead of inside the SQL Server database. Because this option requires a lot of involvement from both the DBA administration and development side, I will spend more time on this topic than the rest.

Each of these data types are available in all editions of SQL Server 2008. Let’s look at each of these, one at a time.

**Date and Time**

In SQL Server 2005 and earlier, SQL Server only offered two date and time data types: DATETIME and SMALLDATETIME. While they were useful in many cases, they had a lot of limitations, including:

* Both the date value and the time value are part of both of these data types, and you can’t choose to store one or the other. This often causes a lot of wasted storage (because you store data you don’t need or want); adds unwanted complexity to many queries because the data types often had to be converted to a different form to be useful; and often reduces performance because WHERE clauses with these data and time data types often had to include functions to convert them to a more useful form, preventing these queries from using indexes.
* They are not time-zone aware, which often requires extra coding for time-aware applications.
* Precision is only .333 seconds, which is often not granular enough for some applications.
* The range of supported dates is not adequate for some applications, and the range does not match the range of .NET CLR DATETIME data type, which requires additional conversion code.

To overcome these problems, SQL Server 2008 introduces four new date and time data types, which include:

* **DATE**: As you can imagine, the DATE data type only stores a date in the format of *YYYY-MM-DD.* It has a range of 0001-01-01 through 9999-12-32, which should be adequate for most business and scientific applications. The accuracy is 1 day, and it only takes 3 bytes to store the date.
* **TIME**: TIME is stored in the format: *hh:mm:ss.nnnnnnn*, with a range of 00:00:00.0000000 through 23:59:59:9999999 and is accurate to 100 nanoseconds. Storage depends on the precision and scale selected, and runs from 3 to 5 bytes.
* **DATETIME2**: DATETIME2 is very similar to the older DATETIME data type, but has a greater range and precision. The format is *YYYY-MM-DD hh:mm:ss:nnnnnnnm* with a range of 0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999, and an accuracy of 100 nanoseconds. Storage depends on the precision and scale selected, and runs from 6 to 8 bytes.
* **DATETIMEOFFSET**: DATETIMEOFFSET is similar to DATETIME2, but includes additional information to track the time zone. The format is *YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm* with a range of 0001-01-01 00:00:00.0000000 through 0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 (in UTC), and an accuracy of 100 nanoseconds. Storage depends on the precision and scale selected, and runs from 8 to 10 bytes.

All of these new date and time data types work with SQL Server 2008 date and time functions, which have been enhanced in order to properly understand the new formats. In addition, some new date and time functions have been added to take advantage of the new capabilities of these four new data types.

**Spatial**

While spatial data has been stored in many SQL Server databases for many years (using conventional data types) SQL Server 2008 includes the introduction of two specific spatial data types which can make it easier for developers to integrate spatial data in their SQL Server-based applications. In addition, by storing spatial data in relational tables, it becomes much easier to combine spatial data with other kinds of business data. For example, by combining spatial data (such as longitude and latitude) with the physical address of a business, applications can be created to map business locations on a map.

They include:

* **GEOMETRY**: The GEOMETRY data type is used to store planar (flat-earth) data. It is generally used to store XY coordinates that represent points, lines, and polygons in a two-dimensional space. For example storing XY coordinates in the GEOMETRY data type can be used to map the exterior of a building.
* **GEOGRAPHY**: The GEOGRAPHY data type is used to store ellipsoidal (round-earth) data. It is used to store latitude and longitude coordinates that represent points, lines, and polygons on the earth’s surface. For example, GPS data that represents the lay of the land is one example of data that can be stored in the GEOGRAPHY data type.

GEOMETRY and GEOGRAPHY data types are implemented as .NET CLR data types, which means they can support various properties and methods specific to the data. For example, a method can be used to calculate the distance between two GEOMETRY XY coordinates, or the distance between two GEOGRAPHY latitude and longitude coordinates. Another example is a method to see if two spatial objects intersect or not. Methods defined by the Open Geospatial Consortium standard, and Microsoft extensions to that standard, can be used. To take full advantage of these methods, you will have to be an expert in spatial data, a topic that well beyond the scope of this chapter.

Another feature of spatial data types is that they support special spatial indexes. Unlike conventional indexes, spatial indexes consist of a grid-based hierarchy in which each level of the index subdivides the grid sector that is defined in the level above. But like conventional indexes, the SQL Server query optimizer can use spatial indexes to speed up the performance of queries that return spatial data.

Spatial data is an area unfamiliar to most DBAs. If this is a topic you want to learn more about, you will need a good math background, otherwise you will get lost very quickly.

**HIERARCHYID**

While hierarchical tree structures are commonly used in many applications, SQL Server has not made it easy to represent and store them in relational tables. In SQL Server 2008, the HIERARCHYID data type has been added to help resolve this problem. It is designed to store values that represent the position of nodes of a hierarchal tree structure.

For example, the HIERARCHYID data type makes it easier to express these types of relationships without requiring multiple parent/child tables and complex joins.

* Organizational structures
* A set of tasks that make up a larger projects (like a GANTT chart)
* File systems (folders and their sub-folders)
* A classification of language terms
* A bill of materials to assemble or build a product
* A graphical representation of links between web pages

Unlike standard data types, the HIERARCHYID data type is a CLR user-defined type, and it exposes many methods that allow you to manipulate the date stored within it. For example, there are methods to get the current hierarchy level, get the previous level, get the next level, and many more. In fact, the HIERARCHYID data type is only used to store hierarchical data; it does not automatically represent a hierarchical structure. It is the responsibility of the application to create and assign HIERARCHYID values in a way that represents the desired relationship. Think of a HIERARCHYID data type as a place to store positional nodes of a tree structure, not as a way to create the tree structure.

**FILESTREAM**

SQL Server is great for storing relational data in a highly structured format, but it has never been particularly good at storing unstructured data, such as videos, graphic files, Word documents, Excel spreadsheets, and so on. In the past, when developers wanted to use SQL Server to manage such unstructured data, developers essentially had two choices. They could store unstructured data in VARBINARY(MAX) columns inside the database; or they could store the data outside of the database as part of the file system, and include pointers inside a column that pointed to the file’s location. This allowed an application that needed access to the file to find it by looking up the file’s location from inside a SQL Server table.

Neither of these options was a perfect solution. Storing unstructured data in VARBINARY(MAX) columns offers less than ideal performance, has a 2 GB size limit, and can dramatically increase the size of a database.

Storing unstructured data in the file system requires that the files have a unique naming system that allows hundreds, if not thousands of files to be keep track of; it requires managing folders to store the data; security is a problem and often requires using NTFS permissions to keep people from accessing the files inappropriately; it requires separate backups of the database and the files; and it doesn’t prevent problems that arise when outside files are modified or moved and the database is not updated to reflect this.

To help resolve these problems, SQL Server 2008 has introduced what is called FILESTREAM storage, which is essentially a hybrid approach that combines the best features of the previous two options.

FILESTREAM storage is implemented in SQL Server 2008 by storing VARBINARY(MAX) binary large objects (BLOBs) outside of the database and in the NTFS file system. While this sounds very similar to the older method of storing unstructured data in the file system and pointing to it from a column, it is much more sophisticated. Instead of a simple link from a column to an outside file, the SQL Server Database Engine has been integrated with the NTFS file system for optimum performance and ease of administration. For example, FILESTREAM data uses the Windows OS system cache for caching data instead of the SQL Server buffer pool. This allows SQL Server to do what it does best: manage structured data; and allows the Windows OS to do what is does best: manage large files. In addition, SQL Server handles all of the links between database columns and the files, so we don’t have to.

In addition, FILESTREAM storage offers these additional benefits:

* Transact-SQL can be used to SELECT, INSERT, UPDATE, DELETE FILESTREAM data.
* By default, FILESTREAM data is backed up and restored as part of the database file. If you want, there is an option available so you can backup a database without the FILESTREAM data.
* The size of the stored data is only limited by the available space of the file system. Standard VARBINARY(MAX) data is limited to 2 GB.

As you might expect, using FILESTREAM storage is not right for every situation, for example, it is best used under the following conditions:

* When the BLOB file sizes average 1MB or higher.
* When fast read access is important to your application.
* When applications are being built that use a middle layer for application logic.
* When encryption is not required, as it is not supported for FILESTREAM data.

If your application doesn’t meet the above conditions, then using the standard VARBINARY(MAX) data type might be your best option. Because this technology is so new, you will want to thoroughly test your options before implementing one option or the other in any new applications you build.

**How to Implement FILESTREAM Storage**

Enabling SQL Server to use FILESTREAM data is a multiple-step process, which includes:

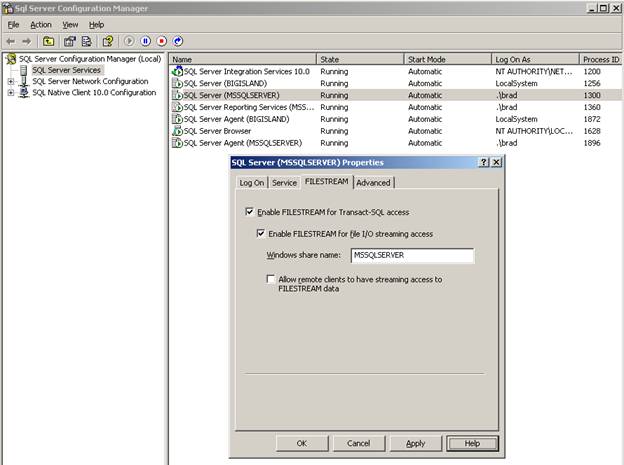
1. Enabling the SQL Server instance to use FILESTREAM data
2. Enabling a SQL Server database to use FILESTREAM data
3. When creating FILESTREAM-enabled columns in a table, specifying the “VARBINARY(MAX) FILESTREAM” data type.

Let’s look at each of these steps, one step at a time.

By default, FILESTREAM storage is not turned on after you install a new SQL Server 2008 instance. If you want to take advantage of it, you must enable it, which is a two step process.

The first step can be performed using the SQL Server 2008 Configuration Manager (demoed here), or by using the sp\_filestream\_configure system stored procedure.

To begin enabling FILESTREAM storage at the instance level, start the SQL Server 2008 Configuration Manager, click on SQL Server Services in the left window, and then in the right window, right-click on the SQL Server instance you want to enable FILESTREAM storage on, choose Properties, then click on the FILESTREAM tab, and the following dialog box appears.



**Figure 1:** Enabling FILESTREAM storage offers several options.

When you enable FILESTREAM storage, you have several options. The first one is “Enable FILESTREAM for Transact-SQL access. This option must be selected if you want to use FILESTREAM storage. If you want to allow local WIN32 streaming access to FILESTREAM storage data, then you must also select the “Enable FILESTREAM for file I/O streaming access” option. In addition, selecting this option requires that you enter a Windows share name where you want the FILESTREAM data to be stored. And last, if you want to allow remote clients to access the FILESTREAM data, then you must select the “Allow remote clients to have streaming access to FILESTREAM data. Keep in mind that you only want to implement those options that you will use, as choosing additional options can increase additional server resource usage. Once you choose your options, click OK.

The next step is to open SQL Server Management Studio (SSMS) and run the following Transact-SQL code from a query window.

EXEC sp\_configure filestream\_access\_level, 2

RECONFIGURE

FILESTREAM storage has now been enabled for the SQL Server instance.

The next step is to enable FILESTREAM storage for a particular database. You can do this when you first create a database, or after the fact using ALTER DATABASE. For this example, we will be creating a new database using Transact-SQL.

The Transact-SQL code used to create a FILESTREAM-enabled database looks like this:

CREATE DATABASE FILESTREAM\_Database

ON

PRIMARY ( NAME = Data1,

    FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\FILESTREAM\_Database.mdf'),

FILEGROUP FileStreamGroup CONTAINS FILESTREAM( NAME = FILESTREAM\_Data,

    FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\FILESTREAM\_Data')

LOG ON  ( NAME = Log1,

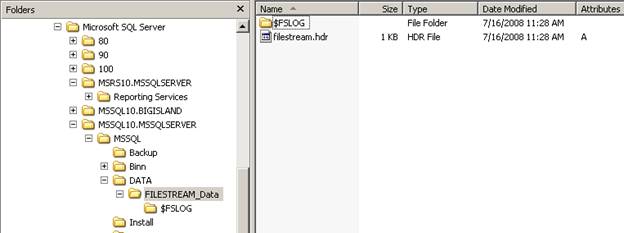
    FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\FILESTREAM\_Database.ldf')

GO

The above code looks similar to the code used to create a regular SQL Server database, except that you can see that there has been the addition of a new filegroup that will be used to store the FILESTREAM data. In addition, when creating the FILESTREAM filegroup, you will be adding the clause “CONTAINS FILESTREAM.”

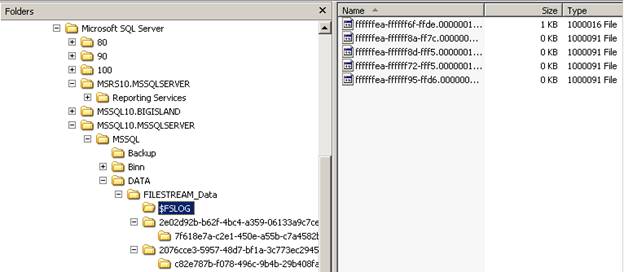
After the above code runs, and the database is created, a new sub-folder is created with the name of “FILESTREAM\_Data.” Notice that this sub-folder name is based on the name I assigned it in the above code.

Inside this newly created folder is a called “filestream.hdr” and an empty sub-folder called $FSLOG. It is very important that you do not delete, modify, or move the“filestream.hdr” file, as it is used to keep track of the FILESTREAM data.



**Figure 2:** When a FILESTREAM-enabled database is created, additional sub-folders, and a system file, are created.

Later, when you create FILESTREAM-enabled columns in tables, and begin adding FILESTREAM data, additional subfolders will be created under the folder created when FILESTREAM was enabled for the database, like you see below.



**Figure 3:** As you create new FILESTREAM-enabled tables, new sub-folders are created.

As you can imagine, you do not want to change any of these folders or the files inside of them. They are all managed by SQL Server.

At this point, our database is FILESTREAM-enabled, and you begin adding new tables that include the VARBINARY(MAX) data type. The only difference between creating a standard VARBINARY(MAX) column in a table and a FILESTREAM-enabled VARBINARY(MAX) column is to add the keyword FILESTREAM after the VARBINARY(MAX). For example, to create a very simple table that can store FILESTREAM data, you can use code similar to this:

CREATE TABLE dbo.FILESTREAM\_Table

(

       DATA\_ID UNIQUEIDENTIFIER ROWGUIDCOL NOT NULL UNIQUE ,

       DATA\_Name varchar(100),

       Catalog VARBINARY(MAX) FILESTREAM,

)

This simple table includes three columns, of which the last one, named “Catalog” can store FILESTREAM data. At this point, you can SELECT, INSERT, UPDATE, and DELETE FILESTREAM data similarly as you would any column in a SQL Server table.