FILESTREAM Storage in SQL Server 2008

SQL Server Technical Article

Writer: Paul S. Randal (SQLskills.com)

Technical Reviewer: Alexandru Chirica, Arkadi Brjazovski, Prem Mehra, Joanna Omel, Mike Ruthruff, Robin Dhamankar

Published: October 2008

Applies to: SQL Server 2008

Summary: This white paper describes the FILESTREAM feature of SQL Server 2008, which allows storage of and efficient access to BLOB data using a combination of SQL Server 2008 and the NTFS file system. It covers choices for BLOB storage, configuring Windows and SQL Server for using FILESTREAM data, considerations for combining FILESTREAM with other features, and implementation details such as partitioning and performance.

This white paper is targeted at architects, IT Pros, and DBAs tasked with evaluating or implementing FILESTREAM. It assumes the reader is familiar with Windows and SQL Server and has at least a rudimentary knowledge of database concepts such as transactions.

Introduction

In today's society, data is generated at incredible rates and often needs to be stored and accessed in a controlled and efficient way. There are various technologies to do this and the choice of technology often depends on the nature of the data being stored – *structured*, *semistructured*, or *unstructured*:

- Structured data is data that can be easily stored in a relational schema, such as that representing sales data for a company. This can be stored in a database with a table of information for products the company sells, another with information about customers, and another that details sales of products to customers. The data can be accessed and manipulated using a rich query language such as Transact-SQL.
- Semistructured data is data that conforms to a loose schema but does not lend itself well to being stored in a set of database tables, such as data where each data point could have radically different attributes. Semistructured data is often stored using the **xml** data type in the Microsoft® SQL Server® database software and accessed using an element-based query language such as XQuery.
- Unstructured data may have no schema at all (such as a piece of encrypted data) or may be a large amount of binary data (many MBs or even GBs) which may seem to have no schema but in reality has a very simple schema inherent to it, such as image files, streaming video, or sound clips. Binary data in this case means data that can have any value, not just those that can be entered at a keyboard. These data values are commonly known as Binary Large Objects, or more simply BLOBs.

This white paper describes the FILESTREAM feature of SQL Server 2008, which allows storage of and efficient access to BLOB data using a combination of SQL Server 2008 and the NTFS file system. It covers the FILESTREAM feature itself, choices for BLOB storage, configuring the Windows® operating system and SQL Server for using FILESTREAM data, considerations for combining FILESTREAM with other features, and implementation details such as partitioning and performance.

Choices for BLOB Storage

While structured and semistructured data can easily be stored in a relational database, the choice of where to store structured or BLOB data is more complicated. When deciding where to store BLOB data, consider the

following requirements:

- Performance: The way the data is going to be used is a critical factor. If streaming access is needed, storing the data inside a SQL Server database may be slower than storing it externally in a location such as the NTFS file system. Using file system storage, the data is read from the file and passed to the client application (either directly or with additional buffering). When the BLOB is stored in a SQL Server database, the data must first be read into SQL Server's memory (the buffer pool) and then passed back out through a client connection to the client application. Not only does this mean the data goes through an extra processing step, it also means that SQL Server's memory is unnecessarily "polluted" with BLOB data, which can cause further performance problems for SQL Server operations.
- Security: Sensitive data that needs to have tightly-managed access can be stored in a database and security can be controlled using the usual SQL Server access controls. If the same data is stored in the file system, different methods of security such as access control lists (ACLs) need to be implemented.
- Data size: Based on the research cited later in this white paper, BLOBs smaller than 256 kilobytes (KB) (such as widget icons) are better stored inside a database, and BLOBs larger than 1 megabyte (MB) are best stored outside the database. For those sized between 256 KB and 1 MB, the more efficient storage solution depends on the read vs. write ratio of the data, and the rate of "overwrite". Storing BLOB data solely within the database (e.g., using the **varbinary(max)** data type) is limited to 2 gigabytes (GB) per BLOB.
- Client access: The protocol that the client uses to access SQL Server data, such as ODBC, may not be suited to applications such as streaming large video files. This may necessitate storing the data in the file system.
- Transactional semantics: If the BLOB data has associated structured data that will be stored in the database, changes to the BLOB data will need to adhere to transactional semantics so the two sets of data remain synchronized. For example, if a transaction creates BLOB data and a row in a database table but then rolls back, the creation of the BLOB data should be rolled back as well as the creation of the table row. This can become very complex if the BLOB data is stored in the file system with no link to the database.
- Data fragmentation: Frequent updates and overwrites will cause the BLOBs to move, either within the SQL Server database files or within the file system, depending on where the data is stored. In this case, if the BLOBs are large, then they may become fragmented (i.e., not stored in one contiguous part of the disk). This fragmentation can be more easily addressed by using the file system than by using SQL Server.
- Manageability: A solution that uses multiple technologies that are not integrated will be more complex and costly to manage than an integrated solution.
- Cost: The cost of the storage solution varies depending on the technology used.

The explanations above around size and fragmentation are based on the well-known Microsoft Research paper titled *To BLOB or Not to BLOB: Large Object Storage in a Database or a Filesystem?* (Gray, Van Ingen, and Sears). The paper has more information on the trade-offs involved and can be downloaded from:

http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2006-45 [http://research.microsoft.com/research/pubs/view.aspx?msr_tr_id=MSR-TR-2006-45]

There are a variety of solutions for BLOB storage, each with pros and cons based on the requirements above. The following table compares three common options for storing BLOB data, including FILESTREAM, in SQL Server 2008.

Comparison point	**Storage solution**		
	File server / file system	**SQL Server (using varbinary(max))**	**FILESTREAM**
Maximum BLOB size	NTFS volume size	2 GB – 1 bytes	NTFS volume size
Streaming performance of large BLOBs	Excellent	Poor	Excellent

Security	Manual ACLs	Integrated	Integrated + automatic ACLs
Cost per GB	Low	High	Low
Manageability	Difficult	Integrated	Integrated
Integration with structured data	Difficult	Data-level consistency	Data-level consistency
Application development and deployment	More complex	More simple	More simple
Recovery from data fragmentation	Excellent	Poor	Excellent
Performance of frequent small updates	Excellent	Moderate	Poor

Table 1: Comparison of BLOB Storage Technologies Before SQL Server 2008

FILESTREAM is the only solution that provides transactional consistency of structured and unstructured data as well as integrated management, security, low-cost, and excellent streaming performance. This is accomplished by storing the structured data in the database files and the unstructured BLOB data in the file system, while maintaining transactional consistency between the two stores. More details of the FILESTREAM architecture are given in the "Overview of FILESTREAM" section later in this white paper.

Overview of FILESTREAM

FILESTREAM is a new feature in the SQL Server 2008 release. It allows structured data to be stored in the database and associated unstructured (i.e., BLOB) data to be stored directly in the NTFS file system. You can then access the BLOB data through the high-performance Win32® streaming APIs, rather than having to pay the performance penalty of accessing BLOB data through SQL Server.

FILESTREAM maintains transactional consistency between the structured and unstructured data at all times, even allowing point-in-time recovery of FILESTREAM data using log backups. Consistency is maintained automatically by SQL Server and does not require any custom logic in the application. The FILESTREAM mechanism does this by maintaining the equivalent of a database transaction log, which has many of the same management requirements (described in more details in the "Configuring FILESTREAM Garbage Collection" section later in this white paper). The combination of the database's transaction log along with the FILESTREAM transaction log allows the FILESTREAM and structured data to be transactionally recovered correctly.

Instead of being a completely new data type, FILESTREAM is a storage attribute of the existing **varbinary (max)** data type. FILESTREAM preserves the majority of the existing behavior of the **varbinary (max)** data type. It alters how the BLOB data is stored – in the file system rather than in the SQL Server data files. Because FILESTREAM is implemented as a **varbinary (max)** column and integrated directly into the database engine, most SQL Server management tools and functions work without modification for FILESTREAM data.

It should be noted that the behavior of the regular **varbinary (max)** data type remains completely unchanged in SQL Server 2008, including the 2-GB size limit. The addition of the **FILESTREAM** attribute makes a **varbinary (max)** column be essentially unlimited in size (in reality the size is limited to that of the underlying NTFS volume).

FILESTREAM data is stored in the file system in a set of NTFS directories called data containers, which correspond to special filegroups in the database. Transactional access to the FILESTREAM data is controlled by SQL Server and a file system filter driver that is installed as part of enabling FILESTREAM at the Windows level. The use of a file system filter driver also allows remote access to the FILESTREAM data through a UNC path. SQL Server maintains a link of sorts from table rows to the FILESTREAM files associated with them. This means that deleting or renaming any FILESTREAM files directly through the file system will result in database corruption.

The use of FILESTREAM requires several schema modifications to data tables (most notably the requirement that each row must have a unique row ID), and it also has some restrictions when it is combined with other features (such as the inability to encrypt FILESTREAM data). These are all described in detail in the "Configuring SQL Server for FILESTREAM" section later in this white paper.

The FILESTREAM data can be accessed and manipulated in two ways – either with the standard Transact-SQL programming model, or via the Win32 streaming APIs. Both mechanisms are fully transaction-aware and support most DML operations including insert, update, delete, and select. FILESTREAM data is also supported for maintenance operations such as backup, restore, and consistency checking. The major exception is that partial updates to FILESTREAM data are not supported. Any update to a FILESTREAM data value translates into creating a new copy of the FILESTREAM data file. The old file is asynchronously removed, as described in the "Configuring FILESTREAM Garbage Collection" section later in this white paper.

Dual Programming Model Access to BLOB Data

After data is stored in a FILESTREAM column, it can be accessed by using Transact-SQL transactions or by using Win32 APIs. This section gives some high-level details of the programming models and how to use them.

Transact-SQL Access

By using Transact-SQL, FILESTREAM data can be inserted, updated, and deleted as follows:

- FILESTREAM fields can be prepopulated using an insert operation (with an empty value or small non-null value). However, the Win32 interfaces are a more efficient way to stream a large amount of data.
- When FILESTREAM data is updated, the underlying BLOB data in the file system is modified. When a FILESTREAM field is set to NULL, the BLOB data associated with the field is deleted. Transact-SQL chunked updates implemented as UPDATE.Write() cannot be used to perform partial updates to FILESTREAM data.
- When a row that contains FILESTREAM data is deleted or a table that contains FILESTREAM data is deleted or truncated, the underlying BLOB data in the file system is also deleted. The actual physical removal of the FILESTREAM files is an asynchronous background process, as explained in the "Configuring FILESTREAM Garbage Collection" section later in this white paper.

For more information and examples of using Transact-SQL to access FILESTREAM data, see the SQL Server 2008 Books Online topic "Managing FILESTREAM Data by Using Transact-SQL" (http://msdn.microsoft.com/en-us/library/cc645962.aspx [http://msdn.microsoft.com/en-us/library/cc645962.aspx]).

Win32 Streaming Access

To allow transactional file system access to the FILESTREAM data, a new intrinsic function, GET_FILESTREAM_TRANSACTION_CONTEXT(), provides the token that represents the current transaction that the session is associated with. The transaction must have been started and not yet committed or rolled back. By obtaining a token, the application binds the FILESTREAM file system streaming operations with a started transaction. The function returns NULL in the case where there is no explicitly started transaction. A token must be obtained before FILESTREAM files can be accessed.

In FILESTREAM, the database engine controls the BLOB physical file system namespace. A new intrinsic function, PathName, provides the logical UNC path of the BLOB that corresponds to each FILESTREAM field in the table. The application uses this logical path to obtain the Win32 handle and operate on the BLOB data by using regular Win32 file system interfaces. The function returns NULL if the value of the FILESTREAM column is NULL. This emphasizes the fact that a FILESTREAM file must be precreated before it can be accessed at the Win32 level. This is done as described previously.

The Win32 streaming support works in the context of a SQL Server transaction. After it obtains a transaction token and a pathname, the Win32 OpenSqlFilestream API is used to obtain a Win32 file handle. Alternatively, the managed SqlFileStream API can be used. This handle can then be used by the Win32 streaming interfaces, such as ReadFile() and WriteFile(), to access and update the file by way of the file system. Again, note that FILESTREAM files cannot be directly deleted, and it cannot be renamed using the file system. Otherwise the link-level consistency will be lost between the database and the file system (i.e., the database essentially becomes corrupt).

The FILESTREAM file system access models a Transact-SQL statement by using file open and close. The statement starts when a file handle is opened and ends when the handle is closed. For example, when a write handle is closed, any possible AFTER trigger that is registered on the table fires as it would if an UPDATE statement were completed.

For more information and examples of using Win32 APIs to access FILESTREAM data, see the SQL Server 2008 Books Online topic "Managing FILESTREAM Data by Using Win32" (http://msdn.microsoft.com/en-us/library/cc645940.aspx [http://msdn.microsoft.com/en-us/library/cc645940.aspx]).

Transaction Semantics

All file handles must be closed before the transaction commits or rolls back. If a handle is left open when a transaction commit occurs, the commit will fail and additional reads and writes against the handle will cause a failure, as expected. The transaction must then be rolled back. Similarly, if the database or instance of the database engine shuts down, all open handles are invalidated.

Whenever a FILESTREAM file is opened for a write operation, a new zero-length file is created and the entire updated FILESTREAM data value is written to it. The old file is removed asynchronously as described in the "Configuring FILESTREAM Garbage Collection" section later in this white paper.

With FILESTREAM, the database engine ensures transaction durability on commit for FILESTREAM BLOB data that is modified from the file system streaming access. This is done using the FILESTREAM log mentioned earlier and an explicit flush of the FILESTREAM file contents to disk.

Isolation Semantics

The isolation semantics are governed by database engine transaction isolation levels. When FILESTREAM data is accessed through the Win32 APIs, only the read-committed isolation level is supported. Transact-SQL access also allows the repeatable-read and serializable isolation levels. Furthermore, using Transact-SQL access, dirty reads are permitted through the read-uncommitted isolation level, or the **NOLOCK** query hint, but such access will not show in-flight updates of FILESTREAM data.

The file system access open operations do not wait for any locks. Instead, the open operations fail immediately if they cannot access the data because of transaction isolation. The streaming API calls fail with ERROR_SHARING_VIOLATION if the open operation cannot continue because of isolation violation.

Partial Updates

To allow for partial updates to be made, the application can issue a device FS control (FSCTL_SQL_FILESTREAM_FETCH_OLD_CONTENT) to fetch the old content into the file that the opened handle references. This can also be done using the managed SqlFileStream API using the ReadWrite flag. This will trigger a server-side old content copy, as discussed earlier. For better application performance and to avoid running into potential time-outs when you are working with very large files, you should use asynchronous I/O.

If the FSCTL is issued after the handle has been written to, the last write operation will persist, and prior writes that were made to the handle are lost.

For more information on partial updates, see the SQL Server 2008 Books Online topic "FSCTL_SQL_FILESTREAM_FETCH_OLD_CONTENT" (http://technet.microsoft.com/en-us/library/cc627407.aspx [http://technet.microsoft.com/en-us/library/cc627407.aspx]).

Write-Through from Remote Clients

Remote file system access to FILESTREAM data is enabled over the **Server Message Block (SMB)** protocol. If the client is remote, caching of write operations depends on the options specified and the API used. For instance, the default for native code APIs is to perform write-through, whereas for the managed APIs the default is to use buffering. This difference reflects customer feedback on the various APIs and their uses through the prerelease CTP versions of SQL Server 2008.

It is recommended that applications that are running on remote clients consolidate small write operations (through buffering) to make fewer write operations using a larger data size. Also, if buffering is being used, an explicit flush should be issued by the client before the transaction is committed.

Creating memory mapped views (memory mapped I/O) by using a FILESTREAM handle is not supported. If memory mapping is used for FILESTREAM data, the database engine cannot guarantee consistency and durability of the data or the integrity of the database.

When to Use FILESTREAM

Even though FILESTREAM technology has many attractive features, it may not be the optimal choice in all situations. As mentioned earlier, the size of the BLOB data and the access patterns are the most significant factors when deciding whether to store the BLOB data wholly within the database or by using FILESTREAM.

Size affects the following:

- Efficiency with which the BLOB data can be accessed using either storage mechanism. As mentioned earlier, streaming access of large BLOB data is more efficient using FILESTREAM, but partial updates are (potentially much) slower.
- Efficiency of backing up the combined structured and BLOB data using either storage mechanism. A

backup that combines SQL Server database files and a large number of FILESTREAM files will be slower than a backup of just SQL Server database files of an equivalent total size. This is because of the extra overhead of backing up each NTFS file (one per FILESTREAM data value). This overhead becomes more noticeable when the FILESTREAM files are smaller (as the time overhead becomes a larger percentage of the total time to backup per MB of data).

As an example, the following graph shows the relative throughput of local reads of various sizes of BLOB data using **varbinary (max)**, FILESTREAM through Transact-SQL, and FILESTREAM through NTFS. It can be seen (on the blue line) that Win32 access of FILESTREAM data becomes several times faster than Transact-SQL access of **varbinary (max)** data as data size increases. Note that the throughput measurements are in megabits per second (Mbps).

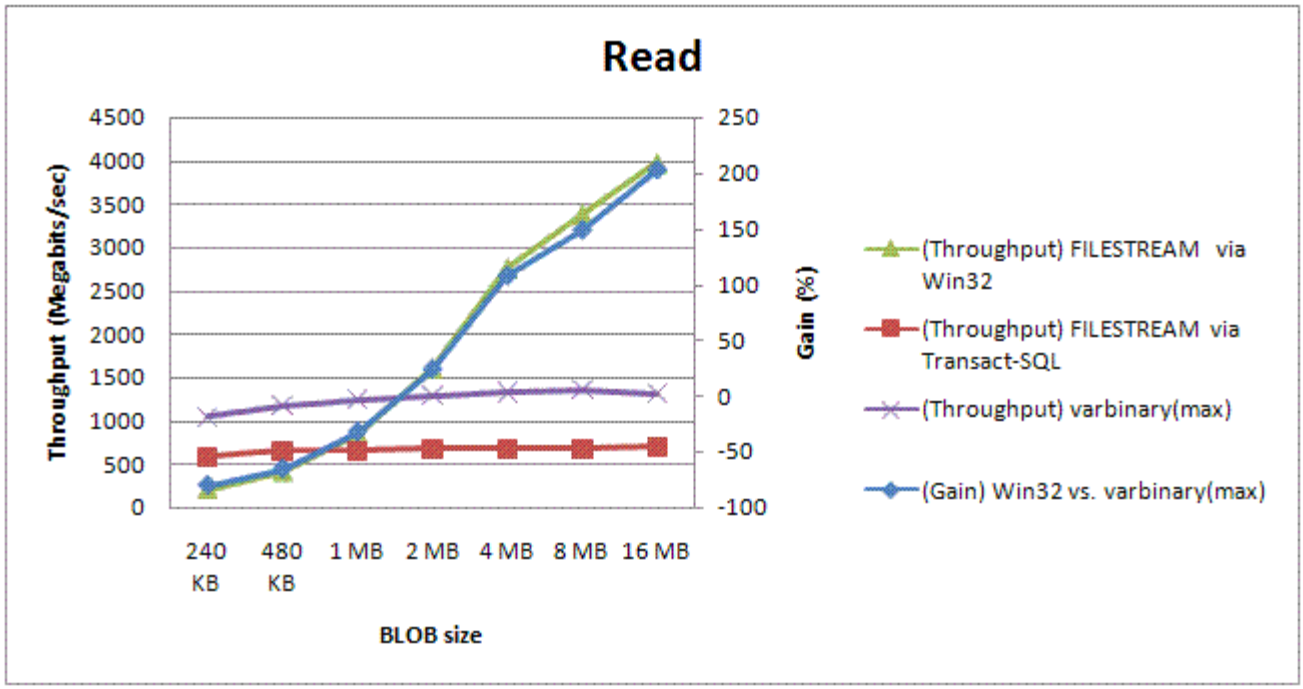

Figure 1: Read performance of various BLOB sizes

The NTFS numbers include the time necessary to start a transaction, retrieve the pathname and transaction context from SQL Server, and open a Win32 handle to the FILESTREAM data. Each test was performed using the same computer with four processor cores and a warm SQL Server buffer pool.

The other factor to consider is whether the client or mid-tier can be written (or altered) to use the Win32 streaming APIs as well as regular access to SQL Server. If this is not the case, FILESTREAM will not be appropriate, because the best performance is obtained using the Win32 streaming APIs.

Configuring Windows for FILESTREAM

As with any other deployment, before you deploy an application that uses FILESTREAM, it is important to prepare the Windows server that will be hosting the SQL Server database and associated FILESTREAM data containers. This section explains how to configure storage hardware and the NTFS file system in preparation for using FILESTREAM. It then shows how to enable FILESTREAM at the Windows level.

Hardware Selection and Configuration

One of the most common causes of a poorly performing workload is an unsuitable hardware configuration. Sometimes the cause is insufficient memory, leading to "thrashing" in SQL Server's buffer pool, and sometimes the cause is simply that the storage hardware does not have the I/O throughput capability the workload demands. For applications that will use FILESTREAM for high-performance streaming of BLOB data using the Win32 APIs, the choice and configuration of the storage hardware is critical.

The following sections describe some best practices around storage choice and layout. For a deeper discussion on this, see the TechNet white paper "Physical Database Storage Design" (http://www.microsoft.com/technet/prodtechnol/sql/2005/physdbstor.mspx [http://www.microsoft.com/technet/prodtechnol/sql/2005/physdbstor.mspx]). After you design an optimal layout, it is prudent to perform load testing to validate the

performance capacity of the I/O subsystem. This is discussed in detail in the TechNet SQL Server Best Practices Article "Predeployment I/O Best Practices" (http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/pdpliobp.mspx [http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/pdpliobp.mspx]).

Physical Storage Layout

Be sure to take into account the anticipated workload on a FILESTREAM data container when you are deciding where to place it, as well as the workloads on any co-located data containers or SQL Server files. Each FILESTREAM data container may need to be on its own volume, as having multiple data containers with heavy workloads on a single volume could lead to contention.

The point to take away here is that without thinking about the workloads involved, simply placing everything on one volume may lead to performance problems. The degree of separation required will vary from customer to customer.

It is also possible to create a table schema inside SQL Server that allows crude load balancing of FILESTREAM data between multiple volumes. This is described in the section "Load Balancing of FILESTREAM Data".

RAID Level Choice

The benefits of using RAID technology are well-known, and much has been written about choosing a RAID level appropriate to the application requirements, so this white paper will not attempt to repeat all that information. The aforementioned "Physical Database Storage Design" white paper has an excellent section on RAID levels and RAID level choice. What follows here is a simple overview of the factors to consider.

RAID levels differ in a variety of ways, most notably in terms of read/write performance, resilience to failure, and cost. For instance, RAID 5 is relatively low cost, it can handle the failure of only one drive in the RAID array, and it may be unsuitable for write-heavy workloads. On the other hand, RAID 10 provides excellent read and write performance, and it can handle multiple drive failures (depending on the degree of mirroring involved), but it is more expensive, given that at least 50 percent of the drives in the RAID array are redundant. These are the three main factors involved in choosing a RAID level. RAID level choice may be different for the volume on which each user database is stored, and it may even differ between the volume storing the data files and that storing the log files for a single database.

If the workload will involve high-performance streaming of FILESTREAM data, the immediate choice may be to have the FILESTREAM data container volume use the RAID level that gives the highest read performance. However, this may not provide a high degree of resilience against failures. On the other hand, the immediate choice may be to use the same RAID level as for the other volumes that store the data for the database, but this may not provide the requisite performance levels that the workload demands.

For this white paper, the point to stress is that a conscious RAID level choice should be made for the FILESTREAM data container volumes after evaluating the trade-offs involved, not just deciding based on a single factor.

Drive Interface Choice

In typical databases involving BLOB data, the total size of the BLOB data may be many times more than the total size of the structured data. When implementing a solution involving FILESTREAM data stored in separate volumes, you may want to use cheaper storage for the volume, such as IDE or SATA (hereafter simply called "SATA"), instead of more expensive SCSI storage. Before this choice is made, the trade-offs involved should be understood. This section gives an overview of the different characteristics of SCSI vs. IDE/SATA to enable an informed choice to be made based on performance and reliability as well as cost.

Capacity and Performance

SATA drives tend to have higher capacity than SCSI drives, but they have slower rotational speed (RPM) than SCSI drives. While there are some 10,000 RPM SATA drives, most are 5,400 or 7,200 RPM. High-performance SCSI drives are available that are 10,000 and up to 15,000 RPM. Although RPM can be a useful comparison metric, the two figures that must really be used for a comparison are the latency (how long until the disk head is at the right position over the disk surface) and average transfer rates (how much data can be transferred to/from the disk surface per second). It is also important that the drives are able to process complex I/O patterns efficiently. When choosing drives, ensure that SATA drives support Native Command Queue (NCQ) and SCSI drives support Command Tag Queue (CTQ), which allow them to process multiple and interleaved disk I/Os for better effective performance.

To summarize, SCSI drives usually have better latency and transfer rates and so will provide better streaming performance, but potentially at a higher cost.

Reliability

SQL Server relies on guaranteed write-ordering and durability to provide reliability and recoverability through its write-ahead logging mechanism. For more information on these I/O requirements, see the TechNet white paper "SQL Server I/O Basics" (http://www.microsoft.com/technet/prodtechnol/sql/2005/iobasics.mspx [http://www.microsoft.com/technet/prodtechnol/sql/2005/iobasics.mspx]).

For reliability, SCSI is usually better than SATA because it uniformly supports forcing data to be written to disk where SATA does not This is done either by supporting write-through, where the data to be written is not cached at all, or by supporting forcibly flushing cache contents to disk. The lack of either can impact recoverability after a hardware, software, or power failure. All the types of interfaces can support hot-swapping to allow repairs while maintaining availability.

The FILESTREAM feature relies on two write-ordering and durability guarantees:

- Data durability at transaction commit time
- Write-ahead logging for FILESTREAM file creation and deletion

Data durability is achieved by the FILESTREAM file system driver issuing an explicit flush of files that have been modified prior to a transaction committing (the details of the mechanism are beyond the scope of this white paper). This guarantees that in the event of a power failure, any disks that do not have sufficient battery-backed cache do not have committed, but unflushed FILESTREAM data that will be lost. If the SATA drives do not support a forced-flush operation, then recoverability may be impacted and data may be lost.

Write-ahead logging relies on the consistency of NTFS metadata. This in itself relies on the reliability of the underlying drives. There is no problem with SCSI, but if the SATA drives do not support forced-flushing, then some NTFS metadata changes may be lost in a power failure situation. This could result in a number of scenarios:

- NTFS cannot recover and the volume cannot be mounted (i.e., the FILESTREAM data container is essentially offline).
- NTFS recovers but NTFS metadata changes are lost and SQL Server does not know to roll back an uncommitted transaction that performs an insert of FILESTREAM data (i.e., the FILESTREAM data is "leaked").
- NTFS recovers but NTFS metadata changes are lost and SQL Server does not know to roll back an uncommitted transaction that performs a delete of FILESTREAM data (i.e., the FILESTREAM data is lost).

It should be noted that all three of these scenarios are no worse than if the BLOB data were stored outside the database on an NTFS volume with underlying SATA drives that did not support forcing data to disk. The use of FILESTREAM on a volume with underlying SATA drives in this case is actually *better* than storing the BLOB data in raw NTFS files on the same volume, as FILESTREAM's link-level consistency provides a mechanism to detect when such "corruptions" have occurred (through running DBCC CHECKDB on the database).

To summarize, FILESTREAM data can be reliably stored on volumes with underlying SATA storage, as long as the SATA drives support forcing data to disk through cache flushing.

NTFS Configuration

Even the most carefully designed I/O subsystem running on high-performance hardware may not work as desired if the file system (in this case NTFS) is not configured correctly. This section describes some of the configuration options that can affect a workload involving FILESTREAM data.

For a more complete overview of NTFS, see the TechNet Library articles "NTFS Technical Reference" (http://technet.microsoft.com/en-us/library/cc758691.aspx [http://technet.microsoft.com/en-us/library /cc758691.aspx]) and "Working with File Systems" (http://technet.microsoft.com/en-us/library/bb457112.aspx [http://technet.microsoft.com/en-us/library/bb457112.aspx]).

Optimizing NTFS Performance

By default, NTFS is not configured to handle a high-performance workload with tens of thousands of files in an individual file system directory (i.e., the FILESTREAM scenario). There are two NTFS options that need to be configured to facilitate FILESTREAM performance. It is especially important to set these options correctly prior to undertaking any performance benchmarking; otherwise, results will not be representative of true FILESTREAM performance.

The first configuration option is to disable the generation of 8.3 names when new files are created (or renamed). This process generates a secondary name for each file that is only for backward compatibility with 16-bit applications. The algorithm generates a new 8.3 name and then has to scan through all existing 8.3 file names in the directory to ensure the new name is unique. As the number of files in the directory grows large (generally

above 300,000), this process takes longer and longer. The time to create a file in turn rises and performance decreases, so turning off this process can significantly increase performance. To turn off this process, type the following at a command prompt and then restart the computer:

```
fsutil behavior set disable8dot3 1
```

Note: This option disables the generation of 8.3 names on all NTFS volumes on the server. If any volumes are being used by 16-bit applications, they may experience issues after you change this behavior.

The second option to turn off is updating the last access time for a file when it is accessed. If the workload briefly accesses many files, then a disproportionate amount of time is spent simply updating the last access time for each file. Turning off this option can also significantly increase performance. To turn off this process, type the following at a command prompt and then restart the computer:

```
fsutil behavior set disablelastaccess 1
```

Cluster Size

All Windows file systems have the concept of a "cluster", which is the unit of allocation when disk space is allocated. As a cluster is the smallest amount of disk space that can be allocated, if a file is very small, some of the cluster may be unused (essentially wasted). The cluster size is therefore typically quite small so that small files do not waste disk space.

Large files may have many clusters allocated to them, or files may grow over time and have clusters allocated as they grow. If a file grows a lot, but in small chunks, then the allocated clusters are likely to not be contiguous on disk (i.e., they are "fragments"). This means that the smaller the clusters are, and the more a file grows, the more "fragmented" it will become.

Cluster size is therefore a trade-off between wasting disk space and reducing fragmentation. More details can be found about the various clusters sizes on the Windows file systems in the Knowledge Base article "Default Cluster Size for FAT and NTFS" (http://support.microsoft.com/kb/140365 [http://support.microsoft.com/kb/140365]).

The recommendation for using FILESTREAM is that the individual units of BLOB data be 1 MB in size or higher. If this is the case, it is recommended that the NTFS cluster size for the FILESTREAM data container volume is set to 64 KB to reduce fragmentation. This must be done manually as the default for NTFS volumes up to 2 terabytes (TB) is 4 KB. This can be done using the /A option of the format command. For example, at a command prompt type:

```
format F: /FS:NTFS /V:MyFILESTREAMContainer /A:64K
```

This setting should be combined with large buffer sizes, as described in the "Performance Tuning and Benchmarking Considerations" section later in this white paper.

Managing Fragmentation

As described earlier, when many files on a volume grow, they become fragmented. This means that the collection of clusters allocated to the file is not contiguous. When the file is read sequentially, the underlying disk heads need to read all the clusters in sequence, which may mean they have to read different portions of the disk. Even if files do not grow once they have been created, if they were created on a volume where the available free space is not in a single contiguous chunk, they may be fragmented immediately, because the necessary clusters to accommodate them are not available contiguously.

This fragmentation causes the sequential read performance to be lower than when there is no (or little) fragmentation. The problem is very similar to that of index fragmentation within a database slowing down query range scan performance.

It is therefore essential that fragmentation be periodically removed using a disk defragmenting tool to maintain sequential read performance. Also, if the volume that will be used to host the FILESTREAM data container was previously used, or if it still contains other data, the fragmentation level should be checked and fixed if necessary.

Compression

Data stored in NTFS can be compressed to save disk space, but at the expense of extra CPU to compress and decompress the data when it is written or read, respectively. Compression is also not useful if the data is essentially uncompressible. For example, random data, encrypted data, or data that has already been compressed will not compress well, but it must still be passed through the NTFS compression algorithm and incur CPU overhead.

For these reasons, it only makes sense to enable compression when the data can be heavily compressed, and when the extra CPU required will not cause the workload performance to decrease. It should also be noted that compression can only be enabled when the NTFS cluster size is 4,096 bytes or less.

Compression can be enabled on the FILESTREAM data container volume when it is formatted, using the /C option of the format command. For example:

```
format F: /FS:NTFS /V:MyFILESTREAMContainer /A:4096 /C
```

An existing volume can also be enabled for compression using the following steps:

1. In My Computer or Windows Explorer, right-click the volume to compress or uncompress.
2. Click **Properties** to display the **Properties** dialog box.
3. On the **General** tab, select or clear the **Compress drive to save disk space** check box, and then click **OK**.
4. In the **Confirm Attribute Changes** dialog box, select whether to make the compression apply to the entire volume or only to the root folder.

This is shown in the following figure.

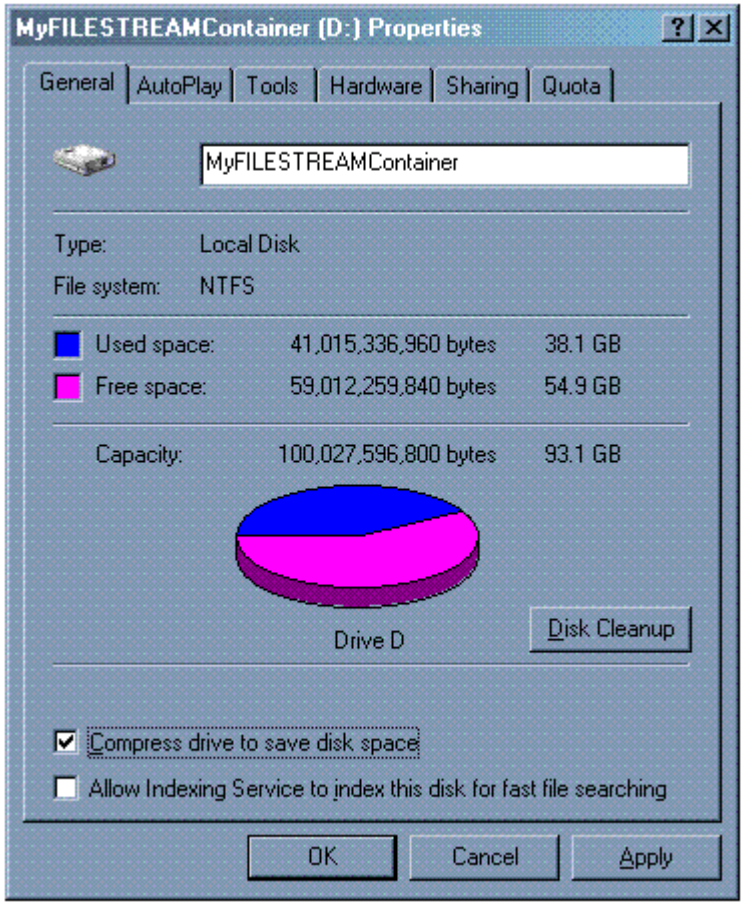

Figure 2: Compressing an existing volume using Windows Explorer

Space Management

Although multiple FILESTREAM data containers can be placed on a single NTFS volume, there are reasons for having a 1:1 mapping between data containers and NTFS volumes. Apart from the potential for workload-dependent contention, there is no way to manage the FILESTREAM data container space usage from within SQL Server, so use of NTFS disk quotas is necessary if this is required. Disk quotas are tracked on a per-user, per-volume basis, so having multiple FILESTREAM data containers on a single volume makes it hard to tell which data container is using more disk space. Note that all the FILESTREAM files will be created under the SQL Server service account. If this is changed, then disk space will start to be charged to the new service account.

There is a single FILESTREAM file system filter driver for each NTFS volume that has a FILESTREAM data

container, and there is also one for each version of SQL Server that has a FILESTREAM data container on the volume. Each filter driver is responsible for managing all FILESTREAM data containers for that volume, for all instances that use a particular version of SQL Server.

For example, an NTFS volume that is hosting three FILESTREAM data containers, one for each of three SQL Server 2008 instances, will have only one SQL Server 2008 FILESTREAM file system filter driver.

Security

There are two security requirements for using the FILESTREAM feature. Firstly, SQL Server must be configured for integrated security. Secondly, if remote access will be used, then the SMB port (445) must be enabled through any firewall systems. This is the same as required for regular remote share access. For more information, see the Knowledge Base article "Service Overview and Network Port Requirements for the Windows Server System" (http://support.microsoft.com/kb/832017 [http://support.microsoft.com/kb/832017]).

Antivirus Considerations

Antivirus software is ubiquitous in today's environment. FILESTREAM cannot block antivirus software from scanning the files in the FILESTREAM data container (this would create security problems). The software usually has a policy setting on what to do to a file that is suspected as being contaminated with a virus: Either delete the file or restrict access to it (known as putting the file in "quarantine"). In both cases, access to the BLOB data in the affected file will be prevented, and to SQL Server the file will appear to have been deleted.

It is recommended that the antivirus software be set to quarantine files, not delete them. DBCC CHECKDB can be used inside SQL Server to discover which files seem to be missing, and then the Windows administrator can correlate the file names against the antivirus software's log and take corrective action.

Enabling FILESTREAM in Windows

FILESTREAM is a hybrid feature that requires both the Windows administrator and the SQL Server administrator to perform actions before the feature is enabled. This is necessary to preserve the separation of duties between the two administrators, especially if the SQL Server administrator is not also the Windows administrator. Enabling FILESTREAM at the Windows level installs a file system filter driver, which is something that only a Windows administrator has privileges to do.

At the Windows level, FILESTREAM is enabled either during the installation of SQL Server 2008 or by running SQL Server Configuration Manager. Here are the steps to follow:

1. On the **Start** menu, point to **All Programs**, point to **Microsoft SQL Server 2008**, point to **Configuration Tools**, and then click **SQL Server Configuration Manager**.
2. In the list of services, right-click **SQL Server Services**, and then click **Open**.
3. In the **SQL Server Configuration Manager** snap-in, locate the instance of SQL Server on which you want to enable FILESTREAM.
4. Right-click the instance and then click **Properties**.
5. In the **SQL Server Properties** dialog box, click the **FILESTREAM** tab.
6. Select the **Enable FILESTREAM for Transact-SQLAccess** check box.
7. If you want to read and write FILESTREAM data from Windows, click **Enable FILESTREAM for file I/O streaming access**. Enter the name of the Windows share in the **Windows Share Name** box.
8. If remote clients must access the FILESTREAM data that is stored on this share, select **Allow remote clients to have streaming access to FILESTREAM data**.
9. Click **Apply**.

The following figure shows the FILESTREAM tab as described in the procedure.

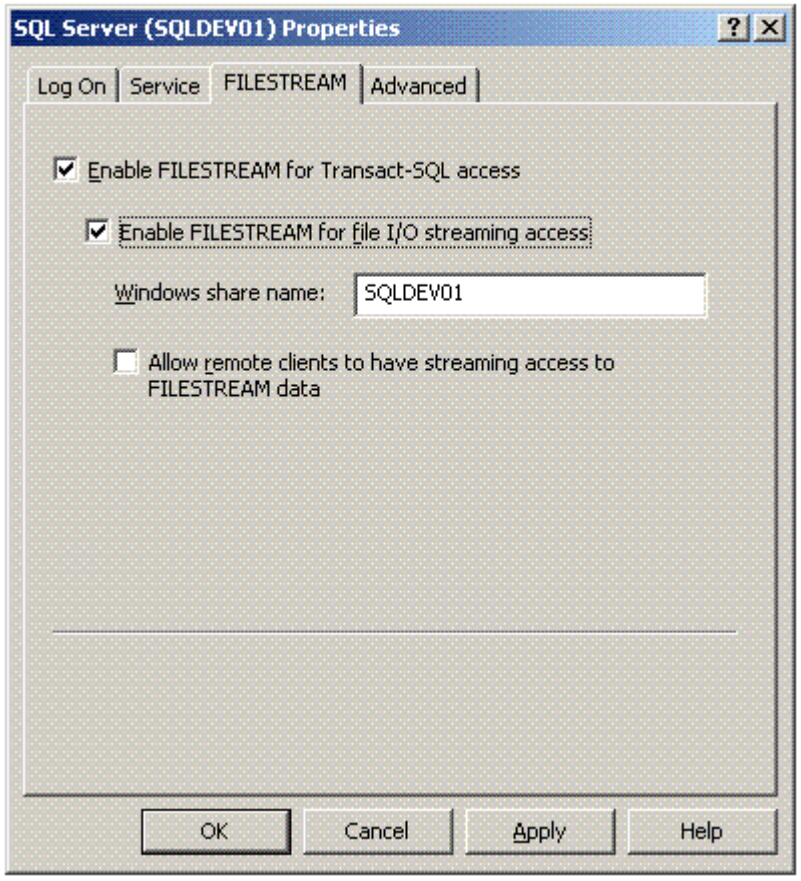

Figure 3: Configuring FILESTREAM using SQL Server Configuration Manager

This procedure must be completed for each SQL Server instance that will use the FILESTREAM feature before it can be used by SQL Server. Note that there is no specification of the FILESTREAM data container at this stage – that is done when a FILESTREAM filegroup is created in a database after FILESTREAM has been enabled within SQL Server.

Note that it is possible to disable FILESTREAM access at the Windows level even while SQL Server has it enabled. In that case, after the SQL Server instance is restarted, all FILESTREAM data will be unavailable. The following warning will be displayed.

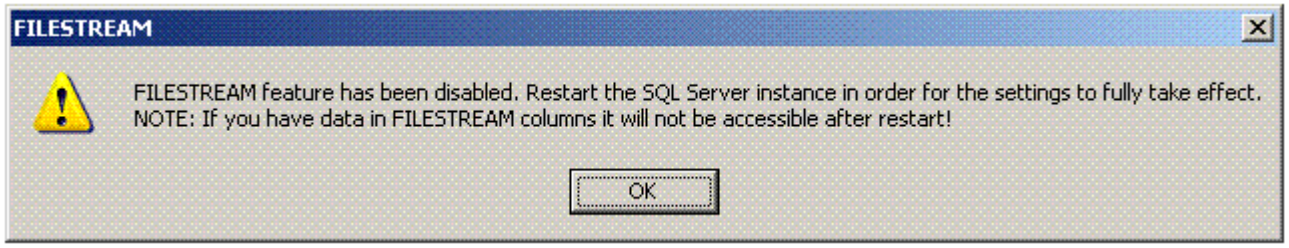

Figure 4: Warning displayed when disabling FILESTREAM using SQL Server Configuration Manager

Configuring SQL Server for FILESTREAM

Each SQL Server instance that will use the FILESTREAM feature must be separately configured, both at the Windows and the SQL Server level. After FILESTREAM has been enabled, a database must be configured to store FILESTREAM data, and only then can tables be defined that include FILESTREAM columns. This section describes how to configure FILESTREAM at the SQL Server level and how to create FILESTREAM-enabled databases and tables, and it explains how FILESTREAM interacts with other features in SQL Server 2008.

Security Considerations

FILESTREAM requires that integrated security (i.e., Windows Authentication) be used. When an application using Win32 attempts to access FILESTREAM data, the Windows user is validated through SQL Server. If the user has

Transact-SQL access to the FILESTREAM data, access will be granted at the Win32 level also, as long as the transaction token is obtained in the security context of the Windows user who is performing the file open.

The requirement for Windows Authentication comes from the nature of Windows File I/O APIs. The only way to pass the client's identity from the client application to SQL Server during a File I/O operation is to use the Windows token associated with the client's thread.

When the FILESTREAM data container is created, it is automatically secured so that only the SQL Server service account and members of the **builtin/Administrators** group can access the data container directory tree. Care should be taken that the data container contents are never changed except through supported transactional methods, as change through other methods will lead to the container becoming corrupt.

Enabling FILESTREAM in SQL Server

The second step of enabling FILESTREAM is performed within the SQL Server 2008 instance. This should not be done until FILESTREAM has been enabled at the Windows level, and the NTFS volume that will store the FILESTREAM data has been adequately prepared (as described in the "Configuring Windows for FILESTREAM" section earlier).

FILESTREAM access is controlled within SQL Server using **sp_configure** to set the filestream_access_level configuration option to one of three settings. The possible settings are:

- 0 – disable FILESTREAM support for this instance
- 1 – enable FILESTREAM for Transact-SQL access only
- 2 – enable FILESTREAM for Transact-SQL and Win32 streaming access

The following example shows enabling FILESTREAM for Transact-SQL and Win32 streaming access.

Transact-SQL Copy Code

```
EXEC sp_configure filestream_access_level, 2;
GO
RECONFIGURE;
GO
```

The RECONFIGURE statement is necessary for the newly configured value to take effect. Note that if FILESTREAM has not been enabled at the Windows level, it will not be enabled at the SQL Server level when the code above is run. The current configured value can be found using the following code.

Transact-SQL Copy Code

```
EXEC sp_configure filestream_access_level;
GO
```

If FILESTREAM is not configured at the Windows level then the "config_value" in the **sp_configure** output will be different (i.e., 0) from the "run_value" after the RECONFIGURE statement has been executed.

Creating a Database Enabled for FILESTREAM

Once FILESTREAM is enabled at the Windows and SQL Server levels, a FILESTREAM data container can be defined. This is done by defining a FILESTREAM filegroup in a database. There is a 1:1 mapping between FILESTREAM filegroups and FILESTREAM data containers.

A FILESTREAM filegroup can be defined when a database is created, or it can be created separately by using an ALTER DATABASE statement. The following example creates a FILESTREAM filegroup in an existing database.

Transact-SQL Copy Code

```
ALTER DATABASE Production ADD
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM;
GO
```

The CONTAINS FILESTREAM clause is necessary to distinguish the new filegroup from regular database

filegroups. If the FILESTREAM feature is disabled, this statement will fail with the following error.

Transact-SQL Copy Code

```
Msg 5591, Level 16, State 3, Line 1
FILESTREAM feature is disabled.
```

Assuming FILESTREAM is enabled at the Windows and SQL Server levels, the filegroup will be created. At this point, the FILESTREAM data container is defined by adding a single file to the filegroup. The pathname specified is the pathname of the directory that will be created as the root of the data container. The entire pathname up to, but not including, the final directory name must already exist. The following example defines the data container for the FileStreamGroup1 filegroup created earlier.

Transact-SQL Copy Code

```
ALTER DATABASE Production ADD FILE (
       NAME = FSGroup1File,
       FILENAME = 'F:\Production\FSDATA')
TO FILEGROUP FileStreamGroup1;
GO
```

At this point, the FSDATA directory will be created. It will be empty apart from two things:

- The file filestream.hdr. This is the FILESTREAM metadata for the data container.
- The directory $FSLOG. This is the FILESTREAM equivalent of a database's transaction log.

It should be noted that a database can have multiple FILESTREAM filegroups. This can be useful to separate the BLOB storage for multiple tables in the database.

Creating a Table for Storing FILESTREAM Data

Once the database has a FILESTREAM filegroup, tables can be created that contain FILESTREAM columns. As mentioned earlier, a FILESTREAM column is defined as a **varbinary (max)** column that has the **FILESTREAM** attribute. The following code creates a table with a single FILESTREAM column.

Transact-SQL Copy Code

```
USE Production;
GO
CREATE TABLE DocumentStore (
       DocumentID INT IDENTITY PRIMARY KEY,
       Document VARBINARY (MAX) FILESTREAM NULL,
       DocGUID UNIQUEIDENTIFIER NOT NULL ROWGUIDCOL
              UNIQUE DEFAULT NEWID ())
FILESTREAM_ON FileStreamGroup1;
GO
```

A table can have multiple FILESTREAM columns, but the data from all FILESTREAM columns in a table must be stored in the same FILESTREAM filegroup. If the FILESTREAM_ON clause is not specified, whichever FILESTREAM filegroup is set to be the default will be used. This may not be the desired configuration and could lead to performance problems.

Once the table is created, the FILESTREAM data container will contain another directory, corresponding to the table, with a subdirectory that corresponds to the FILESTREAM column in the table. This subdirectory will contain the data files once data is entered into the table. The directory structure will vary depending on the number of FILESTREAM columns a table has, and whether the table is partitioned or not.

Note that for a table to have one or more FILESTREAM columns, it must also have a column of the **uniqueidentifier** data type that has the ROWGUIDCOL attribute. This column must not allow null values and must have either a **UNIQUE** or **PRIMARY KEY** single-column constraint. The GUID value for the column must be supplied either by an application when inserting data or by a **DEFAULT** constraint that uses the NEWID() function (or NEWSEQUENTIALID() if merge replication is configured, as mentioned in the "Feature Combinations and Restrictions" section later).

For more information about details and restrictions on the required table schema and options, see the SQL Server 2008 Books Online topic "CREATE TABLE (Transact-SQL)" (http://msdn.microsoft.com/en-us/library/ms174979.aspx [http://msdn.microsoft.com/en-us/library/ms174979.aspx]).

Configuring FILESTREAM Garbage Collection

FILESTREAM data files in the FILESTREAM data container cannot be partially updated. This means that any change to the BLOB data in the FILESTREAM column will create a whole new FILESTREAM data file. The "old" file must be preserved until it is no longer needed for recovery purposes. Files representing deleted FILESTREAM data, or rolled-back inserts of FILESTREAM data, are similarly preserved.

Files that are no longer needed are removed by a garbage collection process. This process is automatic, unlike Windows SharePoint® Services, where garbage collection must be implemented manually on the external BLOB store.

All FILESTREAM file operations are mapped to a Log Sequence Number (LSN) in the database transaction log. As long as the transaction log has been truncated past the FILESTREAM operation LSN, the file is no longer needed and can be garbage collected. Therefore, anything that can prevent transaction log truncation may also prevent a FILESTREAM file being physically deleted. Some examples are:

- Log backups have not been taken, in the FULL or BULK_LOGGED recovery model.
- There is a long-running active transaction.
- The replication log reader job has not run.

FILESTREAM garbage collection is a background task that is triggered by the database checkpoint process. A checkpoint is automatically run when enough transaction log has been generated. For more information, see the SQL Server 2008 Books Online topic "CHECKPOINT and the Active Portion of the Log" (http://msdn.microsoft.com/en-us/library/ms189573.aspx [http://msdn.microsoft.com/en-us/library/ms189573.aspx]). Given that FILESTREAM file operations are minimally logged in the database's transaction log, it may take a while before the number of transaction log records generated triggers a checkpoint process and garbage collection occurs. If this becomes a problem, you can force garbage collection by using the CHECKPOINT statement.

Partitioning Considerations

If the table that contains FILESTREAM data is partitioned, the FILESTREAM_ON clause must specify a partitioning scheme involving FILESTREAM filegroups and based on the table's partitioning function. This is necessary because the regular partitioning scheme will involve the regular filegroups that cannot be used to store FILESTREAM data. The table definition (either in a CREATE TABLE or CREATE CLUSTERED INDEX … WITH DROP_EXISTING statement) then specifies both partitioning schemes.

The FILESTREAM partitioning scheme can specify that all the partitions are mapped to a single filegroup, but this is not recommended as it can lead to performance problems.

The following (contrived) example shows this syntax.

Transact-SQL Copy Code

```
CREATE PARTITION FUNCTION DocPartFunction (INT)
AS RANGE RIGHT FOR VALUES (100000, 200000);
GO
CREATE PARTITION SCHEME DocPartScheme AS
PARTITION MyPartFunction TO (Data_FG1, Data_FG2, Data_FG3);
GO
CREATE PARTITION SCHEME DocFSPartScheme AS
PARTITION MyPartFunction TO (FS_FG1, FS_FG2, FS_FG3);
GO
CREATE TABLE DocumentStore (
        DocumentID INT IDENTITY PRIMARY KEY,
        Document VARBINARY (MAX) FILESTREAM NULL,
        DocGUID UNIQUEIDENTIFIER NOT NULL ROWGUIDCOL
                UNIQUE DEFAULT NEWID () ON Data_FG1)
ON DocPartScheme (DocumentID)
FILESTREAM_ON DocFSPartScheme;
GO
```

Notice that to use the **DocumentID** column as the partitioning column, the underlying nonclustered index that enforces the **UNIQUE** constraint on the DocGUID must be explicitly placed on a filegroup so that the

DocumentID column can be the partitioning column. This means that partition switching is only possible if the **UNIQUE** constraints are disabled before performing the partition switch, as they are unaligned indexes, and then re-enabled afterwards.

Continuing the previous example, the following code creates a table and then attempts a partition switch.

Transact-SQL Copy Code

```
CREATE TABLE NonPartitionedDocumentStore (
       DocumentID INT IDENTITY PRIMARY KEY,
       Document VARBINARY (MAX) FILESTREAM NULL,
       DocGUID UNIQUEIDENTIFIER NOT NULL ROWGUIDCOL
              UNIQUE DEFAULT NEWID ());
GO
ALTER TABLE DocumentStore SWITCH PARTITION 2 TO NonPartitionedDocumentStore;
GO
```

The partition switch fails with the following message.

Msg 7733, Level 16, State 4, Line 1

'ALTER TABLE SWITCH' statement failed. The table 'FileStreamTestDB.dbo.DocumentStore' is partitioned while index 'UQ_Document_8CC1617F60ED59' is not partitioned.

Disabling the unique index in the source table and trying again gives the following code.

Transact-SQL Copy Code

```
ALTER INDEX [UQ__Document__8CC331617F60ED59] ON DocumentStore DISABLE;
GO
ALTER TABLE FileStreamTest3 SWITCH PARTITION 2 TO NonPartitionedFileStreamTest3;
GO
```

This also fails, with the following message.

Msg 4947, Leve 16, State 1, Line 1

ALTER TABLE SWITCH statement failed. There is no identical index in source table 'FileStreamTestDB.dbo.DocumentStore' for the index 'UQ_NonParti_8CC3316103317E3D' in target table 'FileStreamTestDB.dbo.NonPartitionedDocumentStore'.

The unique indexes in both the partitioned and nonpartitioned tables must be disabled before the switch can proceed.

Transact-SQL Copy Code

```
ALTER INDEX [UQ__NonParti__8CC3316103317E3D] ON NonPartitionedDocumentStore DISABLE;
GO
ALTER TABLE DocumentStore SWITCH PARTITION 2 TO NonPartitionedDocumentStore;
GO
ALTER INDEX [UQ__NonParti__8CC3316103317E3D] ON
NonPartitionedDocumentStore REBUILD WITH (ONLINE = ON);
ALTER INDEX [UQ__Document__8CC331617F60ED59] ON
NonPartitionedDocumentStore REBUILD WITH (ONLINE = ON);
GO
```

More information on partitioning FILESTREAM data will be included in the upcoming white paper on partitioning in SQL Server 2008.

Load Balancing of FILESTREAM Data

Partitioning can also be used to create a table schema that allows crude load balancing of FILESTREAM data between multiple volumes. This may be desirable for a variety of reasons such as hardware limitations or to allow hot-spots in a table to be stored on different volumes.

The following code shows a partitioning function and schema based on the **uniqueidentifier** column that effectively spreads the FILESTREAM data across 16 volumes, while striping the structured data across two filegroups.

Transact-SQL Copy Code

```
USE master;
GO
-- Create the database
CREATE DATABASE Production ON PRIMARY
       (NAME = 'Production', FILENAME = 'E:\Production\Production.mdf'),
FILEGROUP DataFilegroup1
       (NAME = 'Data_FG1', FILENAME = 'F:\Production\Data_FG1.ndf'),
FILEGROUP DataFilegroup2
       (NAME = 'Data_FG2', FILENAME = 'G:\Production\Data_FG2.ndf'),
FILEGROUP FSFilegroup0 CONTAINS FILESTREAM
       (NAME = 'FS_FG0', FILENAME = 'H:\Production\FS_FG0'),
FILEGROUP FSFilegroup1 CONTAINS FILESTREAM
       (NAME = 'FS_FG1', FILENAME = 'I:\Production\FS_FG1'),
FILEGROUP FSFilegroup2 CONTAINS FILESTREAM
       (NAME = 'FS_FG2', FILENAME = 'J:\Production\FS_FG2'),
FILEGROUP FSFilegroup3 CONTAINS FILESTREAM
       (NAME = 'FS_FG3', FILENAME = 'K:\Production\FS_FG3'),
FILEGROUP FSFilegroup4 CONTAINS FILESTREAM
       (NAME = 'FS_FG4', FILENAME = 'L:\Production\FS_FG4'),
FILEGROUP FSFilegroup5 CONTAINS FILESTREAM
       (NAME = 'FS_FG5', FILENAME = 'M:\Production\FS_FG5'),
FILEGROUP FSFilegroup6 CONTAINS FILESTREAM
       (NAME = 'FS_FG6', FILENAME = 'N:\Production\FS_FG6'),
FILEGROUP FSFilegroup7 CONTAINS FILESTREAM
       (NAME = 'FS_FG7', FILENAME = 'O:\Production\FS_FG7'),
FILEGROUP FSFilegroup8 CONTAINS FILESTREAM
       (NAME = 'FS_FG8', FILENAME = 'P:\Production\FS_FG8'),
FILEGROUP FSFilegroup9 CONTAINS FILESTREAM
       (NAME = 'FS_FG9', FILENAME = 'Q:\Production\FS_FG9'),
FILEGROUP FSFilegroupA CONTAINS FILESTREAM
       (NAME = 'FS_FGA', FILENAME = 'R:\Production\FS_FGA'),
FILEGROUP FSFilegroupB CONTAINS FILESTREAM
       (NAME = 'FS_FGB', FILENAME = 'S:\Production\FS_FGB'),
FILEGROUP FSFilegroupC CONTAINS FILESTREAM
       (NAME = 'FS_FGC', FILENAME = 'T:\Production\FS_FGC'),
FILEGROUP FSFilegroupD CONTAINS FILESTREAM
       (NAME = 'FS_FGD', FILENAME = 'U:\Production\FS_FGD'),
FILEGROUP FSFilegroupE CONTAINS FILESTREAM
       (NAME = 'FS_FGE', FILENAME = 'V:\Production\FS_FGE'),
FILEGROUP FSFilegroupF CONTAINS FILESTREAM
       (NAME = 'FS_FGF', FILENAME = 'W:\Production\FS_FGF');
GO
USE Production;
GO
-- Create a partition function based on the last 6 bytes of the GUID
CREATE PARTITION FUNCTION LoadBalance_PF (UNIQUEIDENTIFIER)
AS RANGE LEFT FOR VALUES (
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-100000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-200000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-300000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-400000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-500000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-600000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-700000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-800000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-900000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-a00000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-b00000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-c00000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-d00000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-e00000000000'),
      CONVERT (uniqueidentifier, '00000000-0000-0000-0000-f00000000000'));
GO
```

```
-- Create a filestream partitioning scheme that allows mapping to 16 filestream filegroups
CREATE PARTITION SCHEME LoadBalance_FS_PS
AS PARTITION LoadBalance_PF TO (
       FSFileGroup0, FSFileGroup1, FSFileGroup2, FSFileGroup3,
       FSFileGroup4, FSFileGroup5, FSFileGroup6, FSFileGroup7,
       FSFileGroup8, FSFileGroup9, FSFileGroupA, FSFileGroupB,
       FSFileGroupC, FSFileGroupD, FSFileGroupE, FSFileGroupF);
GO
-- Create a data partitioning scheme to round-robin between two filegroups
CREATE PARTITION SCHEME LoadBalance_Data_PS
AS PARTITION LoadBalance_PF TO (
       DataFileGroup1, DataFileGroup2, DataFileGroup1, DataFileGroup2,
       DataFileGroup1, DataFileGroup2, DataFileGroup1, DataFileGroup2,
       DataFileGroup1, DataFileGroup2, DataFileGroup1, DataFileGroup2,
       DataFileGroup1, DataFileGroup2, DataFileGroup1, DataFileGroup2);
GO
-- Create the partitioned table
CREATE TABLE DocumentStore (
       DocumentID INT IDENTITY,
       Document VARBINARY (MAX) FILESTREAM NULL,
       DocGUID UNIQUEIDENTIFIER NOT NULL ROWGUIDCOL
              DEFAULT NEWID (),
       CONSTRAINT DocStorePK PRIMARY KEY CLUSTERED (DocGUID),
       CONSTRAINT DocStoreU UNIQUE (DocGUID))
ON LoadBalance_Data_PS (DocGUID)
FILESTREAM_ON LoadBalance_FS_PS;
GO
```

The load balancing can be easily tested using the following code.

Transact-SQL Copy Code

```
SET NOCOUNT ON;
GO
-- Insert 10000 rows to test load balancing
DECLARE @count INT = 0;
WHILE (@count < 10000)
BEGIN
       INSERT INTO DocumentStore DEFAULT VALUES;
       SET @count = @count + 1;
END;
GO
-- Check the distribution
SELECT COUNT ($PARTITION.LoadBalance_PF (DocGUID))
FROM DocumentStore
GROUP BY $PARTITION.LoadBalance_PF (DocGUID);
GO
```

The results of a sample run of this test were 631, 641, 661, 640, 649, 637, 618, 618, 576, 608, 595, 645, 640, 616, 602, and 623 rows in each of the FILESTREAM filegroups FS_FG0 through FS_FGF.

Feature Combinations and Restrictions

Because the FILESTREAM feature stores data within the file system, there are some restrictions and considerations when FILESTREAM is combined with other SQL Server features. This section gives an overview of the feature combinations to be aware of. For more information, see the SQL Server 2008 Books Online topic "Using FILESTREAM with Other SQL Server Features" (http://msdn.microsoft.com/en-us/library/bb895334.aspx [http://msdn.microsoft.com/en-us/library/bb895334.aspx]).

Replication

Both transactional replication and merge replication support FILESTREAM data, but there are many considerations such as:

- When the replication topology involves instances using different versions of SQL Server, there are limitations on the size of data than can be sent to down-level instances.

- The replication filter options determine whether the FILESTREAM attribute is replicated or not using transactional replication.
- The varbinary (max) maximum data value size that can be replicated in transactional replication without replicating the FILESTREAM attribute is 2 GB.
- When merge replication is used, both it and FILESTREAM require a **uniqueidentifier** column. Care must be taken with the table schema when using merge replication so that the GUIDs are sequential (i.e., use NEWSEQUENTIALID() rather than NEWID()).

Database Mirroring

Database mirroring does not support FILESTREAM. A FILESTREAM filegroup cannot be created on the principal server. Database mirroring cannot be configured for a database that contains FILESTREAM filegroups.

Encryption

FILESTREAM data cannot be encrypted using SQL Server encryption methods. If transparent data encryption is enabled, FILESTREAM data is not encrypted.

Failover Clustering

FILESTREAM is fully supported with failover clustering. All nodes in the cluster must have FILESTREAM enabled at the Windows level, and the FILESTREAM data container(s) must be placed on shared storage so the data is available to all nodes. For more information, see the SQL Server 2008 Books Online topic: "How to: Set Up FILESTREAM on a Failover Cluster" (http://msdn.microsoft.com/en-us/library/cc645886.aspx [http://msdn.microsoft.com/en-us/library/cc645886.aspx]).

Full-Text

Full-text indexing works with a FILESTREAM column in exactly the same way that it does with a **varbinary (max)** column. The table must have an extra column that contains the file name extension for the BLOB data being stored in the FILESTREAM column.

Database Snapshots

SQL Server does not support database snapshots for FILESTREAM data containers. If a FILESTREAM data file is included in a CREATE DATABASE ON clause, the statement will fail and an error will be raised.

If a database contains FILESTREAM data, a database snapshot of regular filegroups can still be created. In this case, a warning message will be returned and the FILESTREAM filegroups will be marked as offline in the database snapshot. Queries will work as expected against the database snapshot unless they attempt to access the FILESTREAM data. If this happens an error will result.

A database cannot be reverted to a snapshot if the database contains FILESTREAM data, because there is no way to tell what state the FILESTREAM data was in at the point in time represented by the database snapshot.

Views, Indexes, Statistics, Triggers, and Constraints

FILESTREAM columns cannot be part of an index key or specified as an INCLUDE column in a nonclustered index. A computed column can be defined that references a FILESTREAM column, but the computed column cannot be indexed.

Statistics cannot be created on FILESTREAM columns.

PRIMARY KEY, FOREIGN KEY, and **UNIQUE** constraints cannot be created on FILESTREAM columns.

Indexed views cannot contain FILESTREAM columns; however, nonindexed views can.

Instead-of triggers cannot be defined on tables that contain FILESTREAM columns.

Isolation Levels

When FILESTREAM data is accessed through the Win32 APIs, only the read-committed isolation level is supported. Transact-SQL access also allows the repeatable-read and serializable isolation levels. Furthermore, using Transact-SQL access, dirty reads are permitted through the read-uncommitted isolation level, or the **NOLOCK** query hint, but such access will not show in-flight updates of FILESTREAM data.

Backup and Restore

FILESTREAM works with all recovery models and all forms of backup and restore (full, differential, and log). In a disaster situation, if the CONTINUE_AFTER_ERROR option is specified on either a BACKUP or RESTORE option, the FILESTREAM data may not recover with zero data loss (similar to recovery of regular data when CONTINUE_AFTER_ERROR is specified).

Security

The SQL Server instance must be configured to use integrated security if Win32 access to the FILESTREAM data is required.

Log Shipping

Log shipping supports FILESTREAM. Both the primary and secondary servers must be running SQL Server 2008, or a later version, and have FILESTREAM enabled at the Windows level.

SQL Server Express

SQL Server Express supports FILESTREAM. The 4-GB database size limit does not include the FILESTREAM data container.

However, if FILESTREAM data is being sent to or from the SQL Server Express instance using Service Broker, then care must be taken, because Service Broker does not support storing data as FILESTREAM in the transmission or target queues. This means that if either queue builds up, the 4-GB database size limit may be reached.

An alternative in this case is to use a scheme where the Service Broker conversation carries notifications that FILESTREAM data needs to be sent or received. Then the actual transmission of the FILESTREAM data is done using remote access through the FILESTREAM share of the SQL Server Express instance's FILESTREAM data container.

Performance Tuning and Benchmarking Considerations

There are several important considerations when tuning a FILESTREAM workload:

- Ensure that the hardware is configured correctly for FILESTREAM.
- Ensure that the generation of 8.3 names is disabled in NTFS.
- Ensure that last access time tracking is disabled in NTFS.
- Ensure that the FILESTREAM data container is not on a fragmented volume.
- Ensure that the BLOB data size is appropriate for storing with FILESTREAM.
- Ensure that the FILESTREAM data containers have their own, dedicated volumes.

One important factor to point out is the buffer size used by the SMB protocol that is used for buffering reads of FILESTREAM data. In testing using the Windows Server® 2003 operating system, larger buffer sizes tend to get better throughput, with buffer sizes of a multiple of approximately 60 KB. Larger buffer sizes may be more efficient on other operating systems.

There are *additional* considerations when comparing a FILESTREAM workload against other storage options (once the FILESTREAM workload has been tuned):

- Ensure that the storage hardware and RAID level is the same for both.
- Ensure that the volume compression setting is the same for both.
- Take into account whether FILESTREAM is performing write-through based on the API in use and options specified.

Data Migration Considerations

A common scenario using SQL Server 2008 will be the migration of existing BLOB data into FILESTREAM storage. While providing a complete tool or set of code to perform such migrations is beyond the scope of this white paper, the following is a simple workflow to follow:

- Review the data size considerations for using FILESTREAM to ensure that the average data sizes involved are such that FILESTREAM storage is appropriate.
- Review the available information on feature combinations and limitations to ensure that FILESTREAM storage will work with all other requirements of the application.

- Follow the recommendations in the "Performance Tuning and Benchmarking Considerations" section earlier.
- Ensure that the SQL Server instance is using integrated security and that FILESTREAM has been enabled at the Windows and SQL Server levels.
- Ensure that the target FILESTREAM data container location has enough disk space to store the migrated BLOB data.
- Create the required FILESTREAM filegroups.
- Duplicate the table schemas involved, changing the required BLOB columns to be FILESTREAM.
- Migrate all non-BLOB data to the new schema.
- Migrate all BLOB data into the new FILESTREAM columns.

FILESTREAM Usage Best Practices

This section is a collection of best practices that have emerged from FILESTREAM usage during the prerelease internal and public testing of the feature. As with all best practices, these are generalizations and they may not apply to every situation and scenario. The best practices are as follows, in no particular order:

- Multiple small appends to a FILESTREAM file should be avoided if possible, as each append creates a whole new FILESTREAM file. This could be very expensive for large FILESTREAM files. If possible, multiple appends should be bunched together into a **varbinary (max)** column and then appended to the FILESTREAM column when a size threshold is reached.
- With a heavily multithreaded write workload, consider setting the AllocationSize parameter for the OpenSqlFilestream or SqlFilestream APIs. Higher initial allocation sizes will limit the potential for fragmentation at the file-system level, especially when combined with a large NTFS cluster size as described previously.
- If FILESTREAM files are large, avoid Transact-SQL updates that append or pre-pend data to a file. This will (generally) spool data into **tempdb** and back into a new physical file, which will affect performance.
- When reading a FILESTREAM value, consider the following:
    - If reads require only reading the first few bytes, than consider substring functionality.
    - If entire file is to be read, consider Win32 access.
    - If random portions of the file are to be read, consider opening the file handle using SetFilePointer.
    - When reading an entire file, specify the FILE_SEQUENTIAL_ONLY flag.
    - Use buffers sized in multiples of 60 KB (as described earlier).

The size of a FILESTREAM file can be obtained without having to open a handle to the file by adding a persisted computed column to the table that stores the FILESTREAM file size. The computed column is updated while the file is already open for write operations.

Conclusion

This white paper has described the FILESTREAM feature of SQL Server 2008 that allows storage of and efficient access to BLOB data using a combination of SQL Server 2008 and the NTFS file system. To conclude, it is worth reiterating the major points made during the white paper.

FILESTREAM storage is not appropriate in all cases. Based on prior research and FILESTREAM feature behavior, BLOB data of size 1 MB and larger that will not be accessed through Transact-SQL is best suited to storing as FILESTREAM data.

Consideration must also be given to the update workload, as any partial update to a FILESTREAM file will generate a complete copy of the file. With a particularly heavy update workload, the performance may be such that FILESTREAM is not appropriate.

The details of feature combinations should be studied to ensure that the deployment succeeds. For instance, in SQL Server 2008 RTM, database mirroring cannot be used FILESTREAM data, nor can either flavor of snapshot isolation. Most other feature combinations are supported, but some may have limitations (such as replication). This white paper does not provide an exhaustive taxonomy of features and their interaction so the most recent SQL Server Books Online sections should be checked prior to deployment, especially as some limitations are likely to be lifted in future releases.

Finally, if FILESTREAM is deployed without correctly configuring Windows and SQL Server, the anticipated performance levels may not be reached. The best practices and configuration details described above should be used to help avoid performance problems.

For more information:

http://www.microsoft.com/sqlserver/ [http://www.microsoft.com/sqlserver/default.aspx] : SQL Server Web site

http://technet.microsoft.com/en-us/sqlserver/ [http://technet.microsoft.com/en-us/sqlserver/default.aspx] : SQL Server TechCenter

http://msdn.microsoft.com/en-us/sqlserver/ [http://msdn.microsoft.com/en-us/sqlserver/default.aspx] : SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

Send feedback [mailto://microsoft.com:25 /default.aspx?subject=White%20Paper%20Feedback:%20FILESTREAM%20Storage%20in%20SQL%20Server%202008] .