.

Technical Report

Microsoft SQL Server Relational Engine: Storage Fundamentals for NetApp Storage

Robert McPhail, NetApp
July 2008| TR-3693

Updated by John S. Parker | January 2011

MICROSOFT SQL SERVER RELATIONAL ENGINE: STORAGE FUNDAMENTALS FOR NETAPP STORAGE

This document will discuss the key fundamentals of how to lay out Microsoft® SQL Server® on NetApp® storage systems. This document will introduce the components of SQL Server and the proper layout for files, logical unit numbers (LUNs), volumes, and aggregates. This technical report will also discuss the uses of file groups and partitions as they relate to SQL Server storage.

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

Microsoft SQL Server is a powerful and cost-effective database platform that can be used to meet a variety of enterprise and departmental needs.

The combination of NetApp storage solutions and Microsoft SQL Server enables the creation of enterprise-level database storage designs that can meet today's most demanding application requirements.

In order to utilize both technologies optimally, it is vital to understand the Microsoft SQL Server relational engine storage architecture.

2 INTENDED AUDIENCE

This technical report is intended for database and storage professionals, who design, test, deploy, and manage SQL Server databases. It is assumed that the reader has working knowledge of the following:

- Microsoft Windows® Server
- Microsoft SQL Server 2005
- Microsoft SQL Server 2008
- Microsoft SQL Server 2008 R2
- NetApp SnapDrive® for Windows
- NetApp SnapManager® for SQL Server
- NetApp Data ONTAP®

3 SCOPE

This paper focuses on the storage-related areas of the SQL Server relational database engine. The following topics are included:

- SQL Server instances
- How SQL Server stores a database on disk
- File groups
- Partitions
- Storage design examples

This technical report does not provide in-depth guidance for database sizing, performance tuning or backup and restore, but does provide general best practices recommendations.

Note: This paper focuses on the *SQL Server relational engine* only.

4 INTRODUCTION

A good database storage design effectively supports the business requirements defined for the database.

The storage design should accommodate the current requirements as well as 12 to 18 months of growth.

This assures that the initial deployment will be successful and that the environment can smoothly grow over time as the business grows.

This technical report discusses the SQL Server relational database storage features that can be used to help achieve that goal. When determining which features to implement in your designs, remember to keep the design as simple as possible while utilizing the appropriate features.

5 SQL SERVER DATABASE STORAGE INTRODUCTION

There are two types of databases in SQL Server: system and user. There is no difference in the physical structure between these two database types. Each type at a minimum has data (*.mdf, *.ndf) and transaction log (*.ldf) files.

Figure 1 depicts an SQL Server database storage stack starting with the database and ending with the aggregate. The highlighted portions show the database and the physical files that composed them.

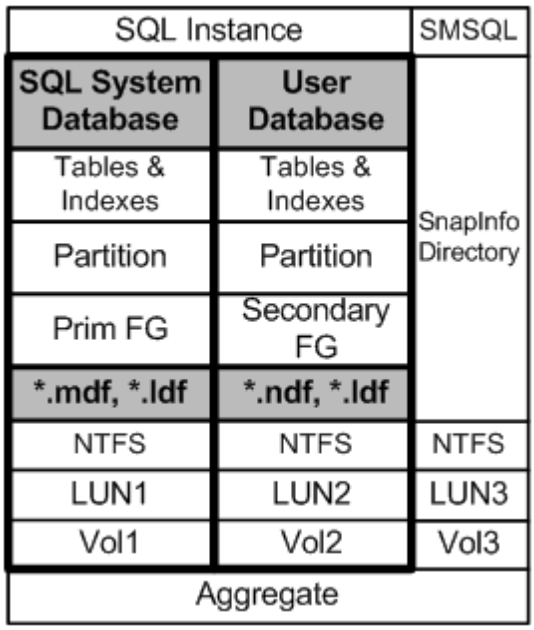

Figure 1) SQL Server database storage stack.

System databases are created each time an instance of SQL Server is installed or by enabling functions. For example, the distribution database is created when SQL Server replication is enabled. Multiple instances of SQL Server can be installed on the same Windows server. Once the SQL Server instance is installed and running, user databases can be created within each instance.

5.1 SQL SERVER DIRECTORY STRUCTURE

Figure 2 shows the directory structure of two SQL Server instances residing on a single Windows host. The optional SQL Server OLAP services and SQL Server Reporting Services are also installed. The directory structure will change depending on which services are installed. Note that each SQL Server instance includes the MSSQL directory tree, which contains the respective system databases for that instance.

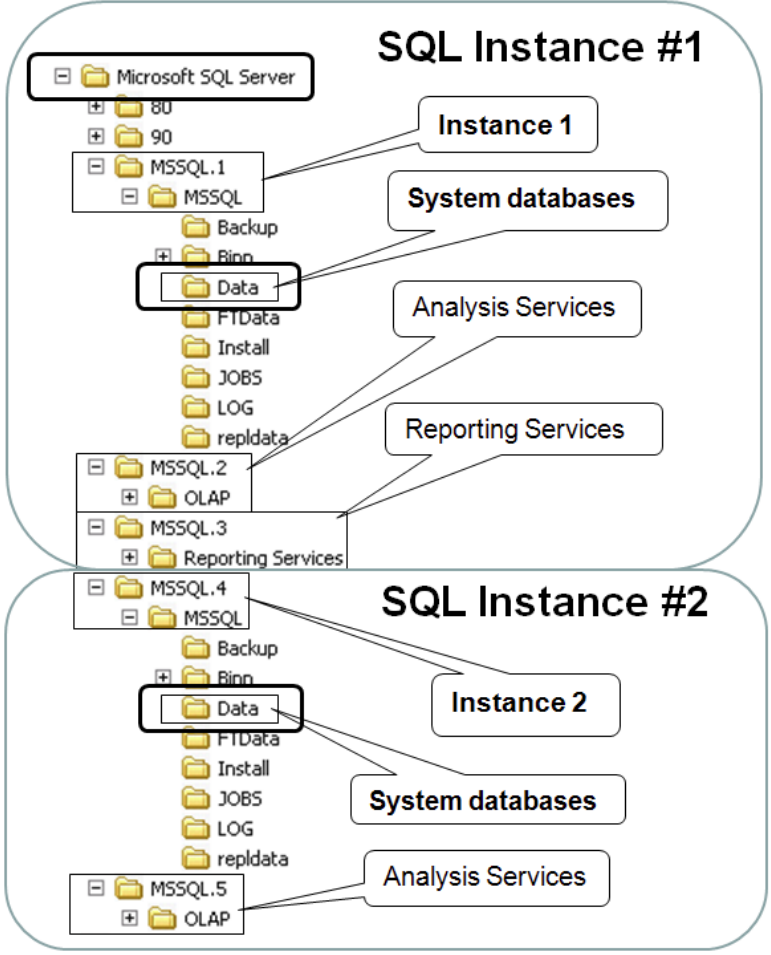

Figure 2) SQL Server directory structure.

6 SQL SERVER SYSTEM DATABASES

By default, the SQL Server system databases for each instance are stored in the MSSQL\Data subdirectory, seen in Figure 2. This is also the default directory for user databases, unless an alternate storage location is specified.

In Figure 3, the contents of the MSSQL\Data subdirectory are shown, highlighting the files composing the system databases. Each SQL Server database (system or user) will have at least one data file (*.mdf) and one transaction log file (*.ldf).

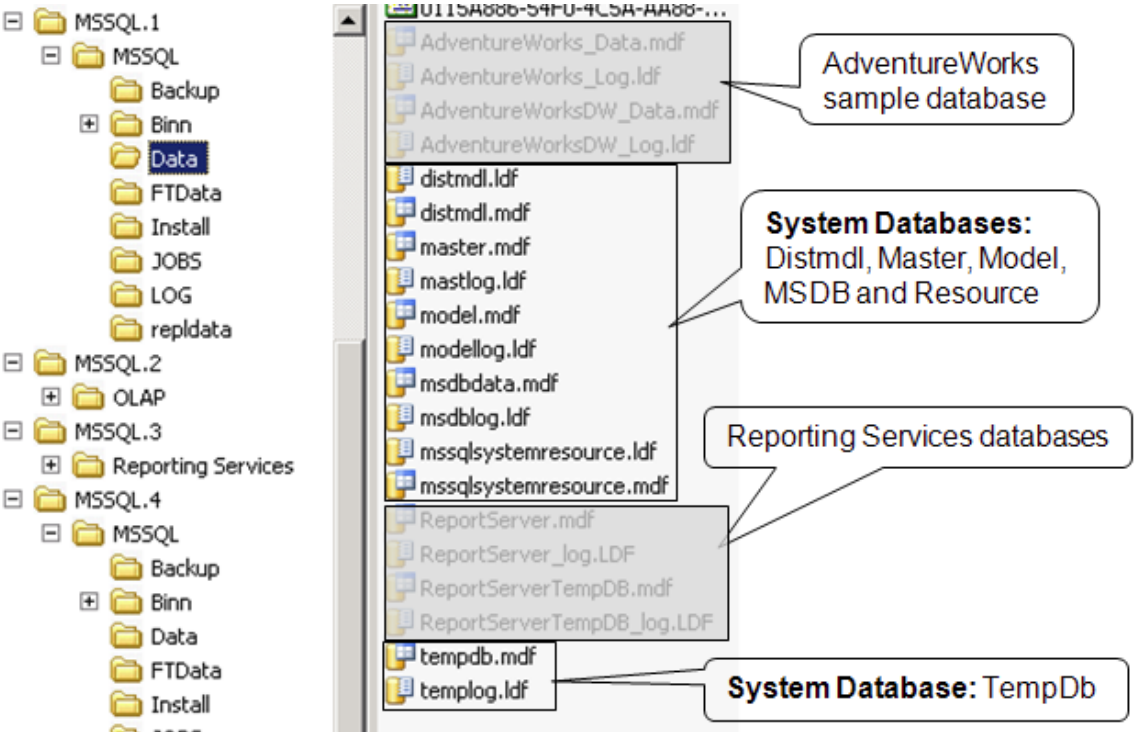

Figure 3) The MSSQL\Data subdirectory contents.

Note: AdventureWorks is a sample user database provided for training and testing. In addition, Report Server is not a system database, but is installed if Reporting Services are installed.

Following is a brief description of each system database:

- Master database: Contains all the system-level information for the SQL Server instance.
- Model database: The template used when creating new user databases determines attributes such as how large the database will be when the size is not explicitly stated.
- MSDB database: Used by the SQL Server agent for managing jobs and alerts and for other features such as the service broker and database mail.
- System resource database: A read-only database of system objects that logically appear in the **sys** schema of each database, but physically exist in this database.
- Tempdb database: Used to hold intermediate results and temporary objects. This database is recreated each time SQL Server restarts.

When NetApp SnapManager® for SQL Server Configuration Manager is run, any database, system, or user can be migrated to NetApp storage by moving the physical database files. This provides an easy way to migrate databases onto NetApp storage systems.

6.1 A CLOSER LOOK AT TEMPDB SYSTEM DATABASE

When it comes to storage and the SQL Server system databases, tempdb is the system database with which to be most familiar, because depending on factors such as query logic and table sizes, tempdb can grow very large and be subject to intense I/O processing. This can lead to poor database performance and could actually cause tempdb to run out of space.

There is only one tempdb in an SQL Server instance, but it is a global resource; used by all the other databases within the instance, including the other system databases, so it is important to understand how each database will utilize it. Figure 4 illustrates how tempdb is a shared resource.

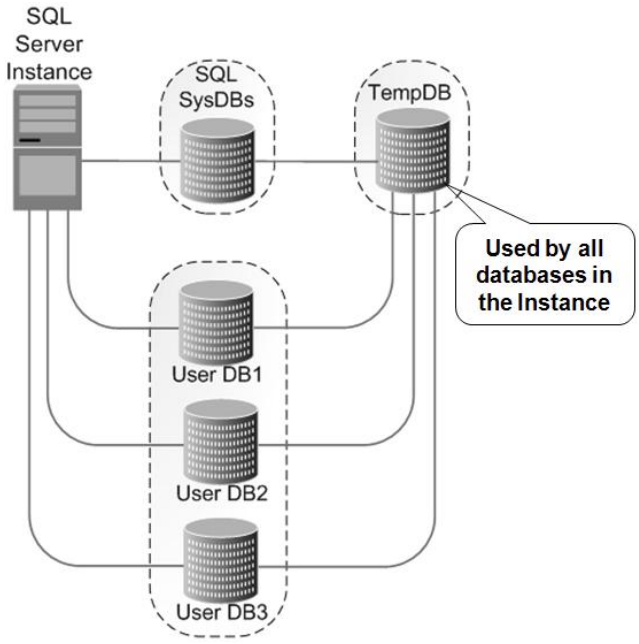

Figure 4) The tempdb system database.

Tempdb, as its name implies, is used to hold temporary information. Also, it is recreated each time SQL Server is restarted. Following are details on how tempdb is used:

- Holds temporary, explicitly created user objects such as stored procedures, cursors, temporary tables, and table variables
- Stores SQL Server internal objects such as work tables for intermediate results for spools or sorting and running database verifications
- Row versions in databases that use Snapshot™ isolation transactions or read-committed using row versioning isolation
- Row versions that are generated by data modification transactions for features such as online index operations, multiple active result sets (MARS), and AFTER triggers

TEMPDB UTILIZATION AND WORKLOAD TYPES

Online transaction processing (OLTP) is a common workload type. OLTP typically does not make heavy use of tempdb, but makes heavy use of the transaction log. Reporting and decision support system (DSS) type workloads, also common, often are heavy users of tempdb, with relatively light use of the transaction log file. Both workload types can coexist on a single storage system with multiple Windows hosts and SQL Server instances.

TEMPDB AND BACKUPS

Tempdb should not be included in a backup since the data it contains is temporary. Place tempdb on a LUN that is in a storage system volume where Snapshot copies will not be created; otherwise, large amounts of valuable Snapshot space could be consumed.

TEMPDB SPACE MANAGEMENT

To prevent tempdb from running out of space, the SQL Server `FILEGROWTH` attribute can be used. The `FILEGROWTH` attribute can be specified for any SQL Server database file. It handles out-of-space conditions by automatically growing the file in the LUN. Note that the LUN must have available space for the file to grow.

`FILEGROWTH` can have a negative impact on storage performance because, when data and log files are created or grown, the new file space is initialized by filling it with zeros. The file growth increment should be set to a reasonable size to avoid either (1) the file constantly growing because the increment is too small or (2) taking too long to grow because the increment is too large. Microsoft's rule of thumb is to set the increment at 10% for tempdb databases larger than 200MB.

Monitor tempdb file size, disk IOPS, and disk latency. Adjust tempdb's size (remember that it is recreated each time SQL Server restarts) accordingly to minimize the number of times `FILEGROWTH` triggers. Use `Alter Database` to change the size. When altered, that will become the new size each time tempdb is recreated. The goal is to eventually set the initial tempdb size so that `FILEGROWTH` rarely, if at all, triggers. Note that this goal might seem counterintuitive because it defeats `FILEGROWTH`. Consider file growth as a failsafe. If tempdb runs out of space, the entire SQL Server instance might stop functioning.

TEMPDB AND DATABASE VERIFICATION

DBCC CHECKDB and related statements typically must read each allocated page from disk into memory so that it can be checked. Running DBCC CHECKDB when there is already a lot of activity on the system impairs DBCC performance for two reasons. First, less memory is available, and the SQL Server database engine is forced to spool some of DBCC CHECKDB's internal data to the tempdb database. Second, DBCC CHECKDB tries to optimize the way that it reads data from the disk. If an intensive workload is also using the same disk, optimization will be greatly reduced, resulting in slower execution.

Because the tempdb database resides on disk, the bottleneck from I/O operations as data is written to and from disk impairs performance. Regardless of system activity, running DBCC CHECKDB against large databases (relative to the size of available memory) causes spooling to the tempdb database.

TEMPDB PERFORMANCE TUNING

Tempdb can use more than one data file. By using more than one data file, I/O can be distributed across multiple files. For a SQL Server instance in which tempdb is heavily used, a general guideline is to create one data file per logical CPU on the host system. This is only a recommendation, and the actual number of data files can be more or less than the number of logical CPUs in the system.

7 SQL SERVER DATABASES

SQL Server by default uses 8KB pages to store tables and indexes, this size can also vary based upon the NTFS block size configuration for the host that where the SQL instance resides. This can be altered to 4K, 8K, 16K, 32K, 64K, 128K and 256K by changing the NTFS block size. The Microsoft best practice is to have the NTFS block size set to 64K. This setting benefits the database when executing large queries against wide tables. The Microsoft best practice for page size is to set the windows allocation block size to 64K. The data (tables and indexes) are organized in the data files (*.mdf, *.ndf), and performs I/O at the page level. The pages are organized within these files in one of two possible ways:

- Heaps: A table without a clustered index. Table rows are not stored in any particular sort order.
- Clustered: A table with a clustered index. Table rows are sorted in order of the clustered index.

Following are some of the key characteristics for each file type:

- Primary data files (*.mdf):
  - Every database has one primary file.
  - Resides in the primary file group.
  - Contains database startup information and points to the other files that compose the database.
  - Can contain user-defined objects, such as tables and indexes.

- Optional secondary data files (*.ndf):
  - A database can have none, one, or many of this file type.
  - Can reside in the primary file group or in optional secondary file groups.
  - Contains user-defined objects such as tables and indexes.

- Transaction log files (*.ldf):
  - A database can have one or many of this file type.
  - Contains the database transaction log records.

7.1 TABLES

Tables contain rows and columns of data. A database can have one-to-many tables. SQL Server tables are stored in one or more database partitions, which will be discussed later in this technical report. *Be sure to not confuse database partitions with LUN partitions, as they are completely different from one another*.

7.2 INDEXES

Indexes can be created for tables; they help queries run much more quickly. Tables without indexes are scanned when searching. Indexes can be clustered or nonclustered, and the difference between the two is significant.

A clustered index takes the table column to be indexed and sorts all the data rows in the table to the order of this column. The index is integrated with the data, and, because of this, clustered indexes always reside in the same data file as the table they support. A table can have either no or one clustered index.

A nonclustered index is stored outside the table in a b-tree format. The actual table rows are not sorted. Nonclustered indexes can reside in the same data file as the table they support, or they can be stored in a separate file. A table can have no, one, or many nonclustered indexes.

PAGE SPLITS, FILL FACTOR, AND PAD_INDEX

When an index page is full and a new row needs to be added, the index page splits. Split processing creates a new page and then moves ~50% of the rows from the original page to the new page. This creates room inside the original page so new rows can once again be added. With clustered indexes, this split processing occurs against the actual data pages.

Page split processing can negatively affect storage performance and cause file fragmentation. To reduce page splits, a fill factor can be used. The fill factor allocates extra space at the leaf level in each page. This provides extra room in the page to accommodate new rows without causing a page split. Pad_index does the same as fill factor, but applies the fill factor to the intermediate-level index pages.

Example: Create an index with a fill factor of 80 and pad the intermediate-level index pages also.

```
USE MyDB;
GO
CREATE NONCLUSTERED INDEX IX_MyIndex
ON MyDB.MyTable(MyCol1)
WITH (FILLFACTOR = 80,
PAD_INDEX = ON,
DROP_EXISTING = ON);
GO
```

FILL FACTOR AND PAD_INDEX CONSIDERATIONS

- Increases disk space usage by the additional free space percentage
- Can decrease database read performance since more pages must be retrieved to access the real data
- Can be applied to new or existing indexes
- Only honored when the index is created or rebuilt
- Fill factor affects the leaf pages only
- Default fill factor is 0
- Pad index affects the intermediate pages and uses additional disk space
- Pad_index only has an effect if fill factor is greater than 0

7.3 TRANSACTION LOG (*.LDF) FILES

Each SQL Server database has a transaction log, which is a write-ahead log file. I/O to the log file is always synchronous. Database modifications are first sequentially written in the transaction log, after which the application can commit the transaction to the database or abort the transaction, in which case modifications to the database will not take place and the log records will be discarded.

Each database in SQL Server uses a recovery model. The recovery model determines the level of logging that takes place for that database. It is a dynamic database attribute, so it can be changed on the fly using the `Alter Database` command. There are three different recovery models:

Simple:

- Use this model when you don't want to take log backups. Only choose this model when a degree of data loss is acceptable, since you can only recover to the point in time when the last backup was taken.
- No log backups can be taken. SQL Server automatically manages the transaction logs, truncating them with each checkpoint.

Bulk-logged:

- If used, this model is used in conjunction with the full recovery model to increase performance and reduce logging.
- Bulk operations such as `SELECT INTO` and index operations such as `CREATE INDEX` are minimally logged, but all other transactions are fully logged, the same as with the full recovery model.

- Use `ALTER DATABASE` to dynamically change the recovery model prior to running bulk operations.
- This model requires log backups. If they are not taken, the log file(s) will fill up.
- Does not support point-in-time recovery, so recovery must be up to the last backup.

Full:

- This is the normal model used for production databases.
- Requires log backups. If they are not taken, the log file(s) will fill up.
- This model enables up-to-the-minute restores as well as restores to a specific point in time.
- The transaction log is used for the following:
    - Recovery of individual transactions
    - Roll back of all incomplete transactions
    - Recovery to the point of failure
    - To support transactional replication and standby server solutions

USE CASE

During large data loads, the fill factor can be set high to reduce the number of splits during the load. Additionally, if the database recovery model is normally full, it can be set to bulk-logged to greatly reduce the amount logging. Only use the bulk-logged recovery model if you are willing to restart the load from the beginning in the event it fails before completion. Performing these two steps can often significantly speed up the load process. After the load is successful, the recovery model can be set back to full, and the table fill factor reduced to a more normal operating value. The new fill factor will take effect the next time the table is reorganized, which is a good step to perform after a data load.

TRANSACTION LOG UTILIZATION AND WORKLOAD TYPES

The amount of I/O load on the transaction log depends on the rate of change against the database. For example, OLTP is a common workload type and typically makes a lot of data changes (add, change, delete), which causes heavy use of the transaction log. In contrast, reporting and DSS type workloads, also common workloads, do mostly reads against the database, so are relatively light users of the transaction log. Realize that both workload types can concurrently exist, possibly running against the same database.

MULTIPLE TRANSACTION LOG FILES

Like tempdb, a database transaction log can have more than one physical file. Unlike tempdb, log file access is sequential to only one file at a time. The reason an additional physical file might be added is to increase log file space. For example, if the LUN where the log file resides is full, a second LUN can be created to add extra space, but this would do nothing for performance, and the I/O is still to one file at a time, filling the first file, then moving on to the second file. With the ability to easily expand LUNs with NetApp storage, there is no identifiable reason to use more than one physical log file.

PLANNING AND DESIGN CONSIDERATIONS

- When the full recovery model is used, the transaction log LUN must be large enough to accommodate all log records generated between log backups.
- The longer the time between log backups, the larger the transaction log will grow.
- The larger the transaction log, the longer it will take to perform backups and recoveries.
- When using SnapManager for SQL Server, the SnapInfo partition/directory must be large enough to hold the maximum number of log backups that will be retained. This is determined when identifying the RTO and RPO for the database.
- If the transaction log is sensitive to I/O delays, it must be backed by enough physical spindles to support the transactional I/O demands.

MONITOR/MANAGE

- Assure that the SnapInfo partition maintains enough space to hold all log backups. SMSQL backups can be configured to maintain the number of backups desired.
- Schedule log backups to keep the log file from filling up.
- Monitor `FILEGROWTH`. Adjust the file size as needed to prevent `FILEGROWTH` from triggering.

8 FILE GROUPS

Up to this point, the SQL Server directory structure and database files, including logs, tables, and indexes, have been discussed. This section will discuss file groups.

8.1 FILE GROUP OVERVIEW

File groups are a core component in the SQL Server database storage architecture. File groups are a logical grouping of physical files (*.mdf, *.ndf) and are used to organize database files in a way that optimizes storage space and performance. When a file group contains more than one file, all the files within the group should be the same size.

The SQL Server database storage stack in Figure 5 highlights three file groups and the files they each contain. Note: A file cannot be defined in more than one file group.

<table>
<tr><td colspan="6">SQL Server Database</td></tr>
<tr><td colspan="5">SQL Server Tables and Indexes</td><td rowspan="3">Tran Log</td></tr>
<tr><td colspan="5">SQL Server Partition</td></tr>
<tr><td>Primary File group</td><td colspan="2">FG1</td><td colspan="2">FG2</td></tr>
<tr><td>*.mdf</td><td>*.ndf</td><td>*.ndf</td><td>*.ndf</td><td>*.ndf</td><td>*.ldf</td></tr>
<tr><td>Partition 1</td><td>P2</td><td>p3</td><td colspan="2">P4</td><td>P5</td></tr>
<tr><td>LUN1</td><td>L2</td><td>L3</td><td colspan="2">L4</td><td>L5</td></tr>
<tr><td colspan="3">Vol 1</td><td colspan="2">Vol 2</td><td>Vol 3</td></tr>
<tr><td colspan="5">AGGR 1</td><td>AGGR 2</td></tr>
</table>

Figure 5) Database file groups and files.

On multiprocessor servers, file groups help take advantage of parallel I/O processing by distributing LUNs across the processors. The Microsoft recommendation on multiprocessor servers is to have at least one LUN per logical processor. For more information, see `affinity I/O mask` in the SQL Server books online.

Characteristics of the preceding design include (excluding the logs):

- Primary file group:
  - Contains the *.mdf file where the system databases reside.
  - Resides on its own LUN partition, LUN, and storage system volume.
  - Shares the same aggregate as the other storage system volumes.
- File group 1 (FG1):
  - Contains two secondary (*.ndf) files.
  - Each file has its own partition and LUN.
  - Shares a volume with the primary file group.
  - Shares the same aggregate with all the other volumes.
- File group 2 (FG2):
  - Contains two secondary (*.ndf) files.
  - Both files share the same partition and LUN.
  - Shares the same aggregate with all the other volumes.

File groups facilitate placing database files on different storage systems, and the level of granularity enables placing a single table, index, or log on a dedicated LUN, volume, or even aggregate. Figure 6

is the same storage design shown in Figure 5; however, it uses two NetApp storage systems, with the data files placed on one storage system, and the log files placed on the other.

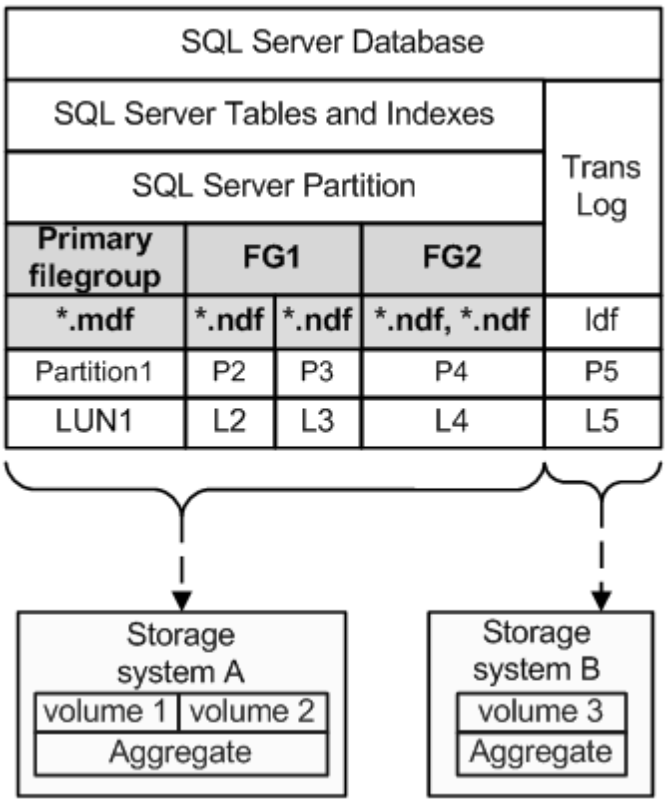

Figure 6) Database file groups and files separate storage controllers.

8.2 HOW SQL SERVER PERFORMS I/O WITH MULTIPLE FILES IN A FILE GROUP

SQL Server uses a proportional fill strategy to write data across multiple files within a file group. When a file group has multiple files, and all the files in the group are the same size, I/O to all files within the group is uniform. When a new file is added to an existing file group that already contains populated tables, the I/O is not initially uniform. Instead, SQL Server will write more data to the new file in proportion to the other files, until space in the new file is consumed to the same amount of space already consumed in the other files within the group, after which I/O will again become uniform across all the files in the file group.

8.3 FILE GROUP TYPES

There are two types of file groups: primary and user defined. Following are some of the characteristics of each:

- Primary
  - There is one per database, created when the database is created.
  - The SQL Server system tables reside here, in the *.mdf file.
  - Contains any other files not allocated to another file group.
- User defined
  - Each database can have no, one, or many of these.
  - Can contain one or many data files (*.ndf).
  - Contains user database tables and indexes.

8.4 THE DEFAULT FILE GROUP

One file group in each database can be designated as the default file group. By doing so, when a table or index is created for that database and a file group is not specified, it will still be defined in the intended set of physical files. If a default file group is not specified, the primary file group is the default file group.

If the initial default file group (primary) is never changed, and you never explicitly specify a different file group name, then when you create new databases, tables, and so on, they will be created in the files in the primary file group. The net effect is that the SQL Server system databases and the user-defined databases will reside on the same physical files. It is a general SQL Server best practice to change the default file group so this will not occur. This is done for performance and availability reasons.

The `ALTER DATABASE` command can be used to specify the default file group. In the following example, a default file group is specified for database "MyDB".

Change the default file group:

```
ALTER DATABASE MyDB
MODIFY FILEGROUP MyDB_FG1 DEFAULT
GO
```

Note: When SnapManager for SQL Server is used to migrate the SQL Server system databases to NetApp LUNs, the primary file group will point to the NetApp LUNs.

8.5 READ-ONLY FILE GROUPS

User-defined file groups can be marked read only so the data within the file group cannot be changed. A common use of read-only file groups is for storing online, historical data. Provided the storage requirements are maintained, further storage savings could be obtained by placing the file group(s) on less expensive disks.

8.6 FILE GROUPS AND PIECEMEAL RESTORES

It is good to be aware of piecemeal restores for two reasons:

- NetApp storage systems enable SQL Server DBAs to avoid this complex procedure.
- Starting in SQL Server 2005, a database can be brought online after restoring just the primary file group, without restoring any of the other file groups.

With Microsoft SQL Server 2005 and later versions, databases with multiple file groups can be restored and recovered in stages through a process known as piecemeal restore. Piecemeal restore involves first restoring the primary file group, after which the database can be brought online, while one or more secondary file groups are restored.

A sample use case would be to store mission-critical data in a select subset of file groups. These could be managed differently; for example, they could reside on more expensive and highly protected storage. Less critical data could be stored in other secondary file groups, perhaps on less expensive storage. In a recovery scenario, the mission-critical file groups would be restored first so the database and related critical applications using that database could be brought online. Then the less critical file groups could be restored in the background.

When using NetApp Snapshot technology, piecemeal restores are usually not needed because a restore from a Snapshot copy is so fast that the entire database can be brought back very quickly.

8.7 CREATING AND MANAGING FILE GROUPS

Following are examples of common file group–related management tasks.

Add a new file group to an existing database:

```
USE master
GO
ALTER DATABASE MyDB
```

```
ADD FILEGROUP MyDB_FG2
GO
```

Add a file to the file group just created in the preceding example:

```
USE master
GO
ALTER DATABASE MyDB
ADD FILE
( NAME = MyDBdat2,
FILENAME = 'g:\MyDB_data2\MyDBdat2.ndf',
SIZE = 25GB,
MAXSIZE = 100GB,
FILEGROWTH = 15GB )
TO FILEGROUP MyDB_FG2
```

Resize an existing file in a file group:

```
USE master
GO
ALTER DATABASE MyDB
MODIFY FILE
(NAME = MyDBdat2,
SIZE = 200GB)
GO
```

Remove the file just added in the preceding example:

```
USE master
GO
ALTER DATABASE MyDB
REMOVE FILE MyDBdat2
GO
```

Change the default file group:

```
ALTER DATABASE MyDB
MODIFY FILEGROUP MyDB_FG1 DEFAULT
GO
```

Make the primary file group the default file group:

```
USE master
GO
ALTER DATABASE MyDB
MODIFY FILEGROUP [PRIMARY] DEFAULT
GO
```

Delete a file group from a database (all files must be removed first):

```
USE master
GO
ALTER DATABASE MyDB REMOVE FILEGROUP MyDB_FG1
GO
```

8.8 PLACING INDEXES ON DEDICATED STORAGE

By default, all indexes reside in the same data file as the table they support. Alternatively, nonclustered indexes can be placed in their own files. This allows nonclustered indexes to be placed on their own LUN or even volume, which might be desirable for performance purposes. Following is an example of how to place an index in its own file in a file group.

In this example a new file group is added to the “MyDB” database. This file group will contain a new file with an extension of .idx. That extension name will help identify it as a file that contains indexes. The index will then be created in that file.

To place an index in its own file, perform the following steps:

1. To use a dedicated LUN, present it to the host and create a partition. NetApp recommends using SnapDrive for this step because it automatically handles creating the LUN and mapping it to the host, as well as creating a properly aligned partition and formatting it. SnapDrive supports volume mount points as well as drive letters.
2. Create a new file group called MyDB_EX_fg1:

```
USE master
GO
ALTER DATABASE MyDB
ADD FILEGROUP MyDB_IX_fg1
GO
```

3. Add the new file to the file group just created:

```
ALTER DATABASE MyDB
ADD FILE
( NAME = MyDB_IX_fg1,
FILENAME = 'g:\MyDB_ix_fg1\MyDB_IX1.idx',
SIZE = 5MB,
MAXSIZE = 10MB,
FILEGROWTH = 1MB )
TO FILEGROUP MyDB_IX_fg1
```

4. Create a new index on the ErrorLog table and place it in the new file group:

```
CREATE NONCLUSTERED INDEX IX_username
ON dbo.ErrorLog (UserName)
ON MyDB_IX_fg1;
Go
```

The new index now resides in its own file, rather than in same file in which the table resides.

9 AN INTRODUCTION TO SQL SERVER TABLE AND INDEX PARTITIONING

This section examines SQL Server table partitions, how they can be used to enhance applications, and the storage implications.

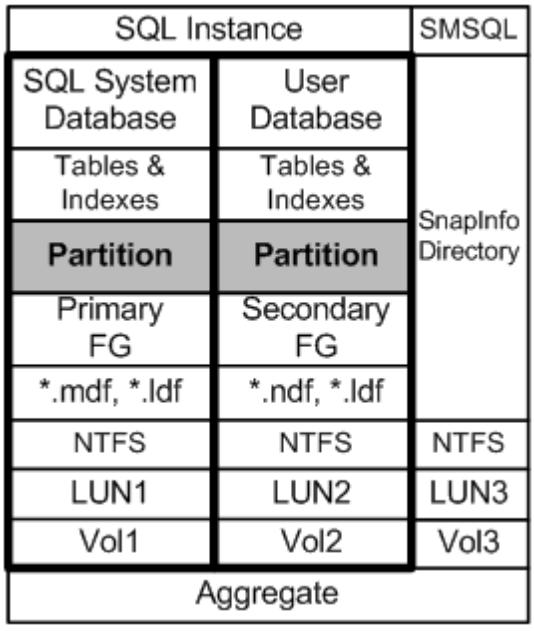

Figure 7) Table and index partitions in the SQL Server database storage stack.

Starting in SQL Server 2005, partitions form the base physical organizational unit for tables and indexes; every table and index page is stored in a partition. A table or index with a single partition, as

illustrated in Figure 7, is equivalent to the organizational structure of tables and indexes in earlier versions of SQL Server.

Partitions allow the data rows stored in one table to be split and stored in multiple smaller tables called partitions. After a table or index is partitioned, you still use the same name to reference it. For example, the “Employee” table is still referenced by “Employee,” whether it has one or multiple partitions.

There are two types of table partitioning: vertical and horizontal. Vertical partitioning splits a table by columns, so that different columns reside in different partitions. Horizontal partitioning splits a table by rows, so that different rows reside in different partitions. This paper discusses horizontal partitioning only.

Note: Partitioned tables and indexes are only available in the Enterprise and Developer editions of SQL Server 2005, 2008, and 2008 R2, and the Data Center edition of SQL Server 2008 R2.

9.1 WHY USE PARTITIONS?

Table partitioning has two significant qualities:

- Partitions offer significant performance improvements in retrieving data from disks.
- Partitions add a level of intelligence to how data rows are distributed across the file groups that compose a table, by automatically inserting rows into the appropriate partition, based on predefined data ranges for each partition.

Consider the following two designs. They show the same table: one without partitions and one with partitions. The table has a clustered index and three nonclustered indexes.

Example 1

The first example, Figure 8, shows the default: one partition.

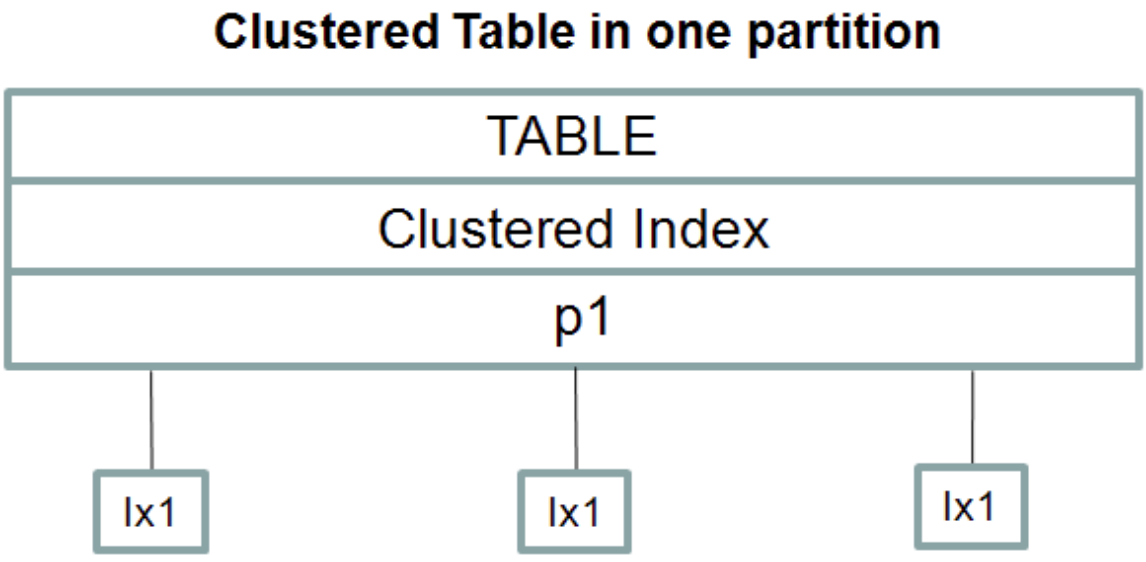

Figure 8) A table and its indexes residing in a single partition.

Example 2

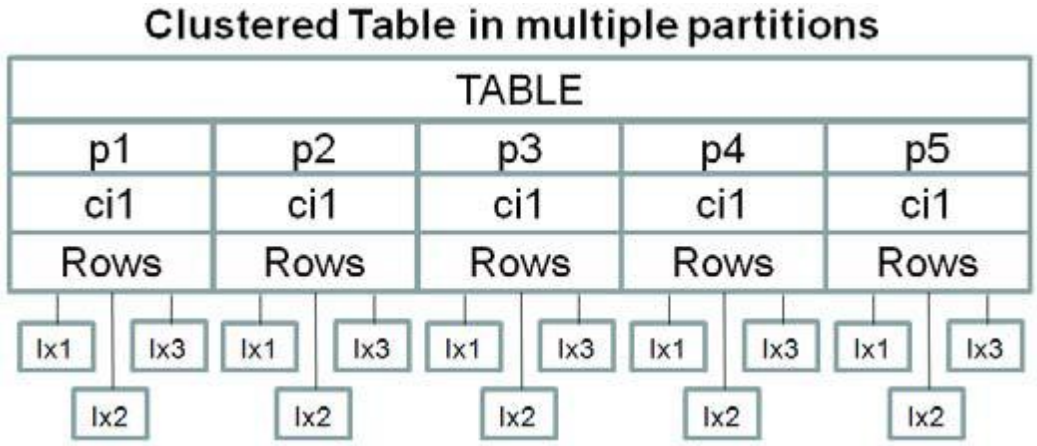

Figure 9) A table and its indexes residing in five different partitions.

The second example, in Figure 9, shows the same table, but using multiple partitions. In this example, each partition would hold ~one-fifth the data as compared to example 1. For example, with a one-million-row table, each partition would contain 200,000 rows.

INFORMATION LIFECYCLE MANAGEMENT AND SLIDING WINDOWS

One use for partitions is with information lifecycle management and sliding windows solutions, both of which have very similar processing. Because partitions can be aligned with file groups, and because file groups can be aligned with LUNs, they enable business applications to drive data migration at the LUN level through a process of splitting, switching, and merging partitions.

PARTITIONS AND PERFORMANCE

As the following information is reviewed, keep in mind that each partition can reside on its own LUN. Many operations occur at the partition level. Query performance can be improved because only the partitions containing the data ranges are searched.

Referring to the preceding example 1 and example 2, without partitions, table scans, index reorgs, new index creation, and other table-level operations must operate against all one million rows. With partitioned tables, those same operations process just one-fifth of the rows, so they operate much more quickly.

Loading data from an OLTP to an online analytical processing (OLAP) system is much faster, taking seconds rather than minutes or hours in some cases, since just the partitions containing the required source data are read, rather than the entire table.

When adding new partitions to existing tables, the data can initially be loaded into an empty, offline table, minimizing the impact on the production systems. After the load completes, the new table can be "switched" into the online table, which is a very fast metadata operation.

Table partitions can eliminate the need for partitioned views. Partitioned views are less efficient and require more administrative overhead.

9.2 RANGE LEFT OR RANGE RIGHT

This aspect of partitioning can relate directly to the number of LUNs that will be required for the table, depending on the partitioning strategy that has been deployed. Partitions have boundaries, or ranges that determine which table rows belong in which partition. Boundaries are defined using the `CREATE PARTITION FUNCTION` command. Boundaries are enforced by way of a partitioning key, which is just a column in the table, and optionally, table constraints. Figure 10 shows a table with three partitions (p1, p2, and p3) with ranges of 1–10, 11–20, and 21–30.

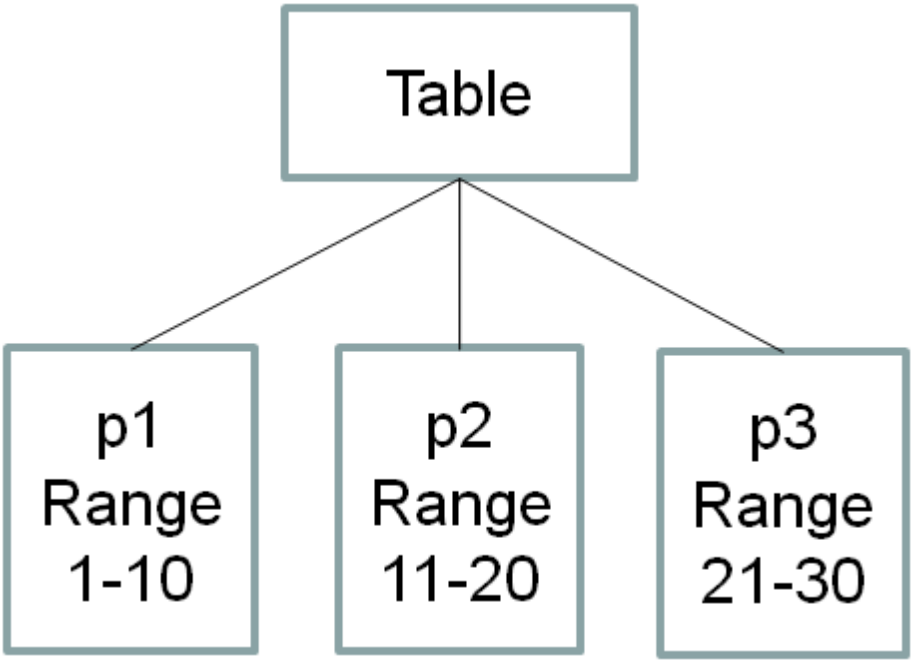

Figure 10) A table with three partitions.

As each row is written to the table, the partitioning key is evaluated, and the row is written to the appropriate partition. The question is: Where do rows that are outside the range of all the partitions get written? These rows must be accommodated. The answer is that one extra partition actually exists. The extra partition does not have a boundary value. Depending on the nature of the data, the extra partition will be placed either to the left or to the right of all the other partitions. Placement is determined by using range left or range right. An example of using range right is included in the command examples that follow.

9.3 CREATING PARTITIONS

After the partition design is complete, which includes identifying the partitioning key and determining how many partitions, file groups, and LUNs will be needed, you can create the partitions. Following is an example to demonstrate the commands used to create partitions. The LUNs are already mounted and ready to use.

These steps also are the basic flow when building partitions for a sliding-window scenario.

1. Create the file groups. Commands for doing this were demonstrated earlier. This example uses four file groups, with one LUN in each file group.

2. Create the partition function. This specifies the partitioning key and range values for each partition.

In the following example, four partitions are created.

```
Use MyDB
Go
CREATE PARTITION FUNCTION pf1_MyRange (int)
AS RANGE RIGHT FOR VALUES (1, 10, 20);
Go
```

The data will be distributed across the partitions as follows:
p1: range values <= 1

p2: range values > 1 and <= 10

p3: range values > 10 and <= 20

p4: range values > 20

3. Create the partition scheme. This will map each partition created earlier to a file group.

```
Use MyDB
Go
CREATE PARTITION SCHEME ps1_MyRange
AS PARTITION pf1_MyRange
TO (MyDB_fg1, MyDB_fg2, MyDB_fg3, MyDB_fg4);
Go
```

This creates the following mapping:
p1 to MyDB_fg1

p2 to MyDB_fg2

p3 to MyDB_fg3

p4 to MyDB_fg4

4. Create the partitioned table and optional index(es).

```
Use MyDB
Go
CREATE TABLE pt_Table (col1 int, col2 int, col3 int))
ON ps1_MyRange (col1);
GO
CREATE NONCLUSTERED INDEX MyDB_IX1
ON MyDB.pt_Table (col1)
ON ps1_MyRange (col1);
GO
```

5. Load the table. This is a standard table load.
6. Other remaining steps might include creating clustered indexes and foreign key references.

9.4 MANAGING PARTITIONS

Microsoft SQL Server provides a complete set of functions for managing partitions. The business requirements determine the exact partition management processes. Figure 11 shows a high-level example of the steps that occur for a specific scenario.

In this scenario, 12 monthly partitions are being maintained. It is time to add the next month of data. The oldest partition needs to be swapped out, and the new partition needs to be added in order to maintain just 12 partitions.

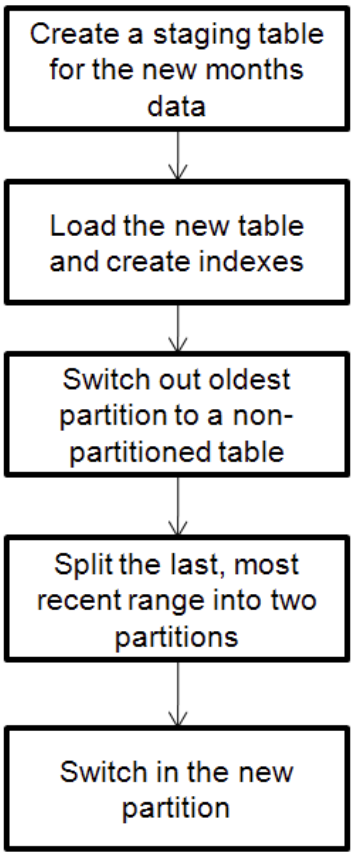

Figure 11) High-level step flow for adding a new partition and removing the oldest partition.

PARTITIONING SUMMARY

Partitioning introduces additional levels of complexity, including in the storage management area, and improperly designed partitions can actually impair query performance. But for certain business needs, the compelling features offered by partitioning outweigh these complexities, which in any case are reduced with SQL Server 2008 and SQL Server 2008 R2 by the included partitioning wizards.

For complete details on SQL Server partitions, refer to the SQL Server Books Online.

10 GENERAL STORAGE RECOMMENDATIONS

This section provides general recommendations for creating database storage designs on NetApp storage systems. Note that working with storage and databases is a subjective process, and sometimes best practices have exceptions.

The following information is based on systems using the following technologies:

- Microsoft Windows Server
- NetApp SnapDrive for Windows
- NetApp SnapManager for SQL Server
- NetApp storage system (FAS or V-Series)

10.1 KEEP IT SIMPLE

Keep the design as simple as possible while taking advantage of the appropriate technology features. Always verify that the design supports and meets all the business requirements.

10.2 RAID TYPE

Use RAID-DP®. When many spindles are available, put 16 spindles in each RAID-DP group.

10.3 USE ONE AGGREGATE

Aggregates are the lowest level in the storage stack. They are containers of physical disks out of which volumes are carved.

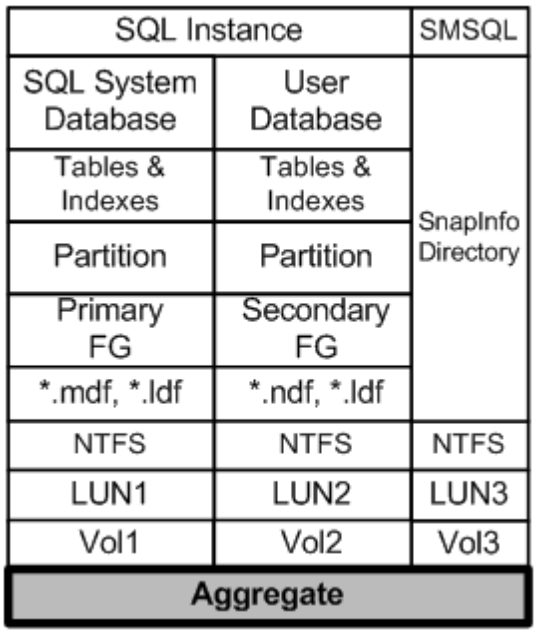

Figure 12) Aggregates highlighted in the SQL Server database storage stack.

Figure 12 shows one aggregate containing three volumes. Two volumes are being used by the databases, and the other volume is being used by SnapInfo.

NetApp recommends using one large aggregate for all SQL Server databases, even though aggregates can be dedicated to specific databases. There are two reasons for this:

- One aggregate makes the I/O abilities of all spindles available to all files.
- One aggregate enables the most efficient use of disk space.

NetApp has performed various tests using both approaches. Data and log separation as well as DSS and OLTP workload types were tested on both shared and dedicated aggregates. The conclusion is that one large aggregate yields significant performance benefits and is easier for administrators to manage. Also, as physical disks become larger and larger, efficient space management using multiple aggregates becomes even more challenging. For example, significant disk space would be wasted in an aggregate containing high-capacity drives dedicated just to a high-utilization log file.

The prevailing reason for using more than one aggregate is high availability: for example, one for data and another for logs. With more than one aggregate, if one of the aggregates failed (while highly unlikely), the odds are increased that there will be less data loss because the other aggregate is still available.

For environments requiring extremely high availability, NetApp Sync Mirror® can be used to create and maintain a local mirror of the complete aggregate.

When creating and sizing aggregates, take into consideration:

- The total size of all the databases using the aggregate
- The number of database Snapshot copies that will be retained
- The number of log backups that will be retained
- The rate of data change and duration of Snapshot retention

- The I/O load generated by all users accessing all the databases that will share the same aggregate
- The projected storage space growth
- The projected user count growth
- Plans for adding new databases to the aggregate
- Any other nondatabase files that might use the same aggregate
- When creating aggregates, let the NetApp storage system select which physical disks will be in each aggregate

10.4 STORAGE SYSTEM VOLUMES

NetApp FlexVol® volumes are created and reside inside aggregates. Many volumes can be created in a single aggregate, and each volume can be expanded or shrunk. Figure 13 shows an aggregate containing four volumes.

<table>
<tr><td colspan="3">SQL Instance</td><td>SMSQL</td></tr>
<tr><td>System DBs</td><td>SQLDB1</td><td>SQLDB2</td><td rowspan="5">SnapInfo Directory</td></tr>
<tr><td>Tables & Indexes</td><td>Tables & Indexes</td><td>Tables & Indexes</td></tr>
<tr><td>Table Partition</td><td>Table Partition</td><td>Table Partition</td></tr>
<tr><td>File Group(s)</td><td>File Group(s)</td><td>File Group(s)</td></tr>
<tr><td>*.mdf, *.ldf</td><td>*.mdf, *.ndf, *.ldf</td><td>*.mdf, *.ndf, *.ldf</td></tr>
<tr><td>NTFS</td><td>NTFS</td><td>NTFS</td><td>NTFS</td></tr>
<tr><td>LUN1</td><td>LUN2</td><td>LUN3</td><td>LUN4</td></tr>
<tr><td>Vol1</td><td>Vol2</td><td>Vol3</td><td>Vol4</td></tr>
<tr><td colspan="4">AGGR</td></tr>
</table>

Figure 13) One aggregate with four volumes.

SNAPSHOT COPIES

SnapDrive creates Snapshot copies at the volume level. SnapManager for SQL Server also creates Snapshot copies at the volume level. With volume-level Snapshot copies, all the data in the volume is included in the Snapshot copy, even when only some of the data in the volume is pertinent to the specific database being backed up. Up to 255 Snapshot copies can be created per volume.

SNAPMIRROR

Like Snapshot copies, SnapMirror also operates at the volume level, as well as at the qtree level. All the data in a source volume will be mirrored to the target volume. Using one volume per database provides the most granular control over SnapMirror frequency, as well as provides the most efficient use of bandwidth between the SnapMirror source and the destination volumes. When addressing HA

implementations, it would be more common to have one volume for logs and a separate volume for all the data LUNs.

VOLUME DESIGN BEST PRACTICE CONSIDERATIONS

Before a database volume design can be created, the backup and recovery requirements must be defined. They provide the “specs” needed for the volume design process. Following are best practice considerations to apply during the volume design process:

- Place the SQL Server system databases on a dedicated volume to assure separation from the user databases.
- Place tempdb on a dedicated volume to assure it is not part of a Snapshot copy.
- When using SnapManager for SQL Server, place the SnapInfo directory on a dedicated volume.
- It is common to take transaction log backups more frequently than database backups, so place the transaction log and data files on separate volumes so independent backup schedules can be created for each.
- If each database has a unique backup requirement, do one of the following:
  - Create separate FlexVol volumes for each database and transaction log.
  - Place databases with similar backup and recovery requirements on the same FlexVol volume. This can be a good option in cases of many small databases in a SQL Server instance.
- Storage administration overhead can increase as more volumes are used.
- Do not share a volume with more than one Windows server. The exception is when using Microsoft Cluster Services (Windows Server 2003) or Windows Failover Cluster (Windows Server 2008 and 2008 R2).

For complete details, refer to the SQL Server Books Online and the SnapManager for SQL Server IAG.

10.5 WINDOWS VOLUME MOUNTPOINTS

NetApp storage solutions and Microsoft SQL Server 2005, 2008, and 2008 R2 support mount points. Mount points are directories on a volume that can be used to “mount” a different volume. Mounted volumes can be accessed by referencing the path of the mount point. Mount points eliminate the Windows 26-drive-letter limit and offer greater application transparency when moving data between LUNs, moving LUNs between hosts, and unmounting and mounting LUNs on the same host. This is because the underlying volumes can be moved around without changing the mount point path name.

VOLUME MOUNTPOINT NAMES

When using volume mount points, a general recommendation is to give the volume label the same name as the mount point name. The reason is illustrated in the following example. Consider a database with four volumes. Two volumes use drive letters, and two use mount points.

- Drive M: for the logs
- Drive S: for SnapInfo
- The data files use two mount points: “mydb_data1” and “mydb_data2.” These are mounted in drive M:, in the directory named “mount.”

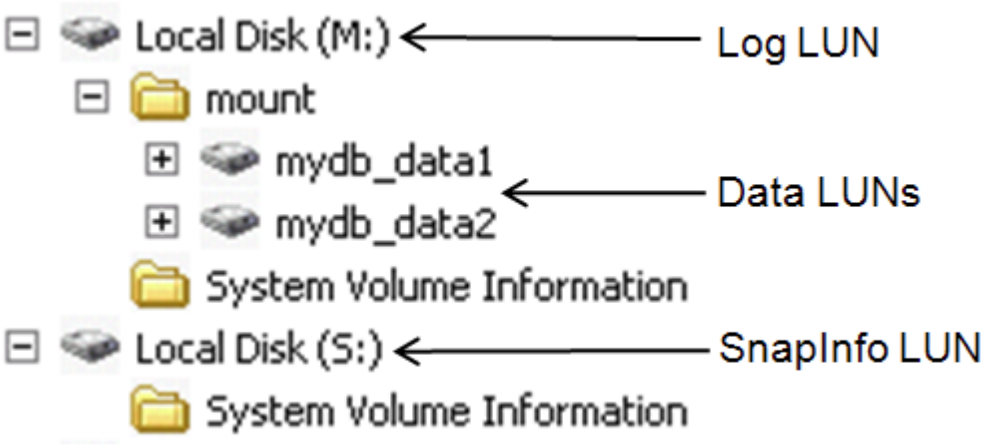

Figure 14) Screen shot of database "mydb" data, log, and SnapInfo file paths with no labels.

Figure 15 shows the volumes in the Windows Disk Management Console.

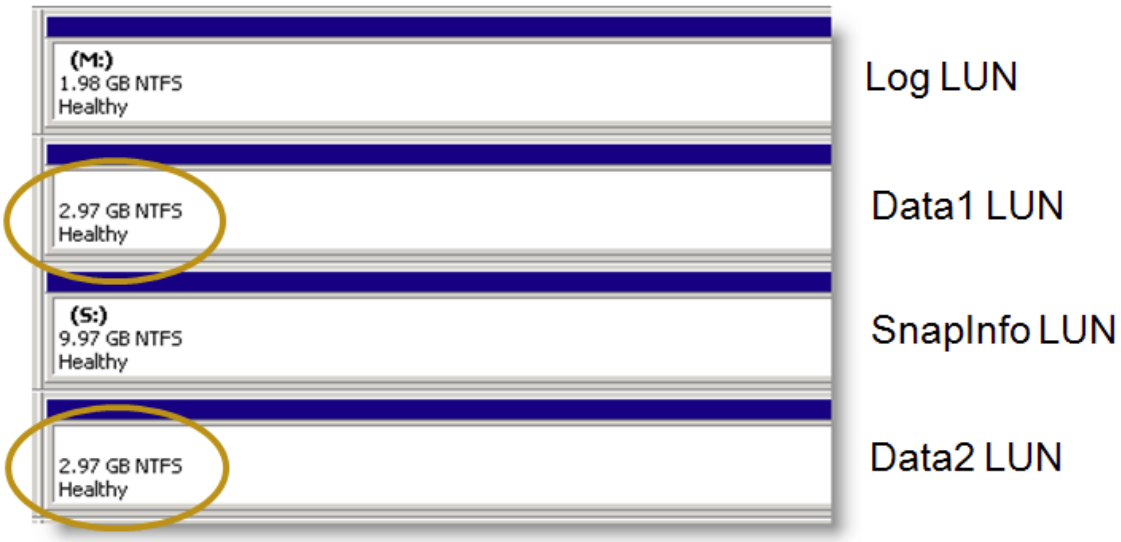

Figure 15) Screen shot of database "mydb" data, log, and SnapInfo LUNs with no labels.

Notice it is not possible to identify how each volume is used. This can be resolved by using volume labels, as shown next.

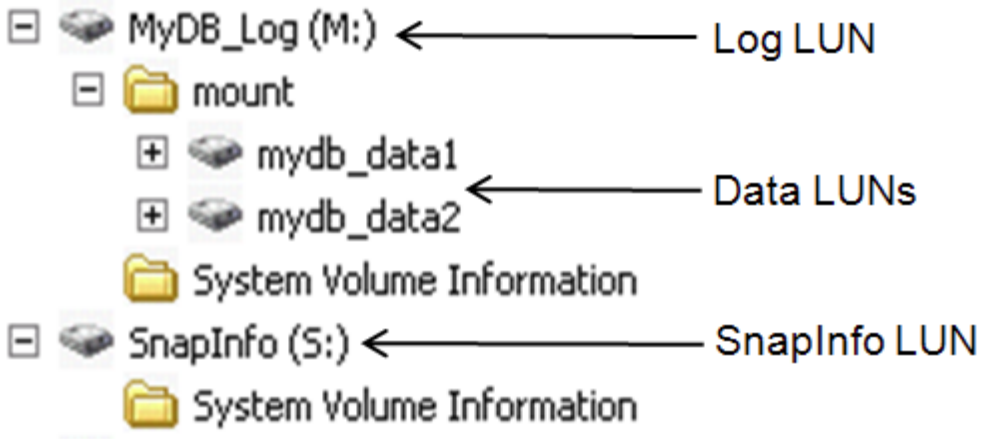

Figure 16) Screen shot of database "mydb" data, log, and SnapInfo file paths with labels.

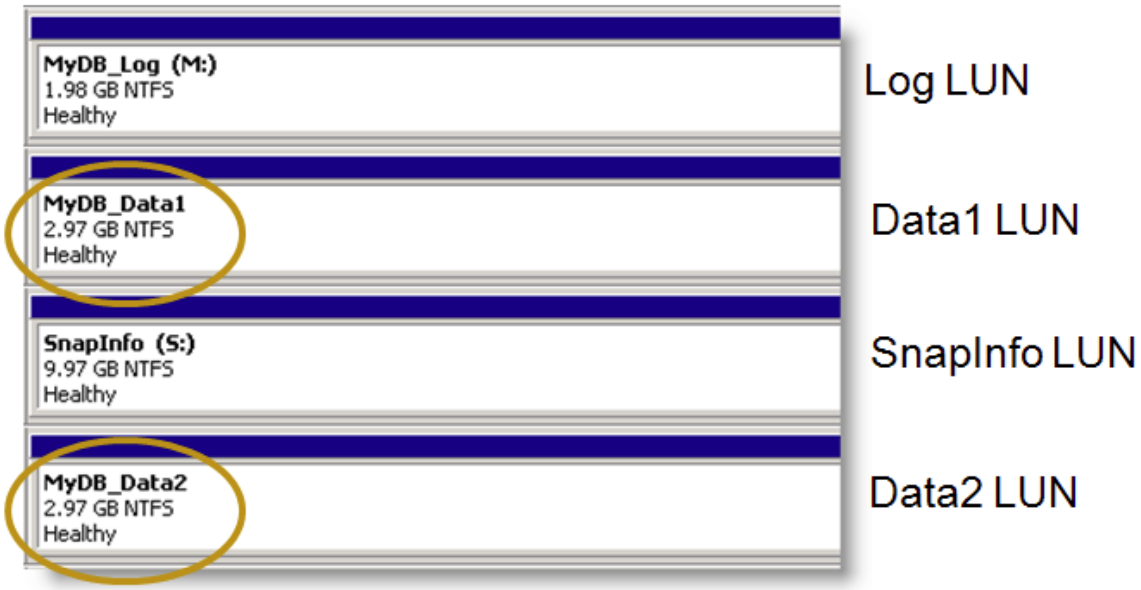

Figure 17) Screen shot of database "mydb" data, log, and SnapInfo LUNs with labels.

With the labels added, as shown in Figures 16 and 17, you can easily identify the purpose of each volume. This is particularly useful when volumes are dismounted, then mounted on a different host or back to the same host, because the mount points have to be reestablished.

Changing the volume name can be done by right-clicking the volume, in Windows Explorer, for example, selecting properties, and typing in the new name. This is just a label, and changing the name does not affect the volume mount point path name used by applications.

MOUNTPOINTS AND SNAPMANAGER FOR SQL SERVER

NetApp SnapManager for SQL Server supports the use of mount points. For more information, refer to the SnapManager for SQL Server Installation and Administration Guide.

11 DATABASE STORAGE DESIGNS

This section provides examples of SQL Server designs for NetApp storage and takes into consideration environments that use SnapManager for SQL Server.

11.1 DESIGN EXAMPLE 1: BASIC

This is a simple storage design.

- It does not use SQL Server partitions beyond the default configuration.
- There is one aggregate for the SQL Server instance.
- It uses a dedicated vol/LUN for the SQL Server system databases, including tempdb.
- It uses a dedicated vol/LUN for each user database.
- It uses a dedicated vol/LUN for SMSQL's SnapInfo directory.

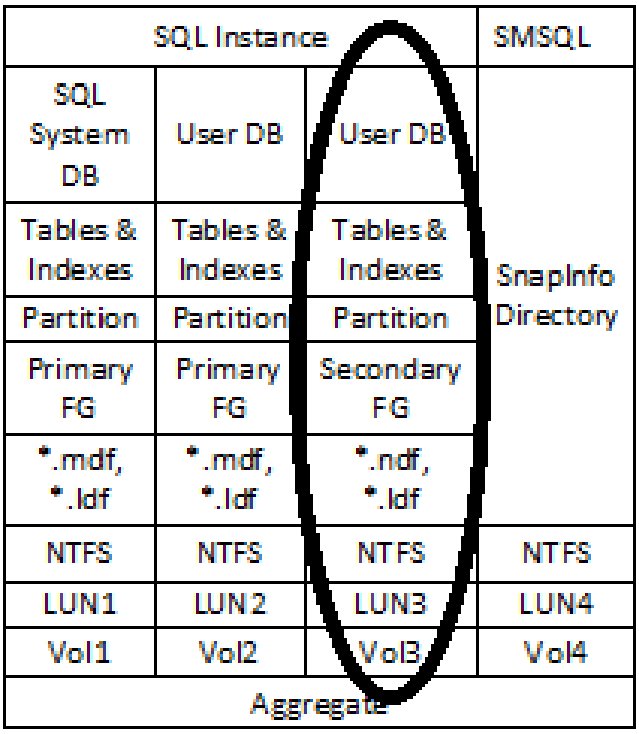

Figure 18) Basic SQL Server database design for NetApp storage system.

This configuration can be scaled for multiple user-defined databases per instance by replicating the circled area. It can also be scaled for multiple SQL Server instances on the same server by replicating the circled section.

The same aggregate can be used as the user databases and the SQL Server instances are scaled out. In fact, the same aggregate can be used for all the SQL Server hosts connected to the storage system.

For SnapManager for SQL Server, there can be one SnapInfo directory for all the databases, which is implied in this design, or one per database. The SnapManager for SQL Server IAG provides the information needed to determine what is appropriate for your specific design.

11.2 DESIGN EXAMPLE 2: SEPARATE TEMPDB

This design example is identical to design example 1, except tempdb has been placed on its own volume. Isolating tempdb onto its own volume makes it possible to keep it out of Snapshot copies. It also provides more granular control of which disks it resides on and how many LUNs it is composed of. Spreading tempdb across multiple LUNs can help improve its ability to handle higher I/O requirements.

After the storage is configured, tempdb can be moved to its own LUNs using NetApp SnapManager for SQL Server or Microsoft SQL Server Management Studio.

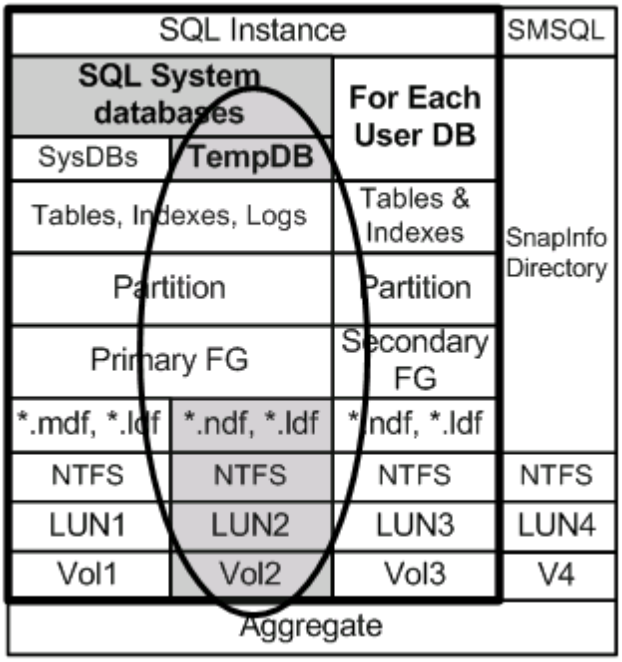

Figure 19) SQL Server database design for NetApp storage systems: tempdb on dedicated volume.

11.3 DESIGN EXAMPLE 3: SEPARATE TRANSACTION LOG

This design builds on the previous two. In this design, the user database log has been separated onto its own volume. This provides more granular control of which disks it resides on and how many LUNs it is composed of. Spreading the log across multiple LUNs can help improve its ability to handle higher I/O requirements.

This also allows the log file to be managed independently of the data file(s). Remember that Snapshot copies occur at the volume level. Using SnapManager for SQL Server, one schedule can create database Snapshot backups every hour, and a different schedule can create log backups every 10 minutes.

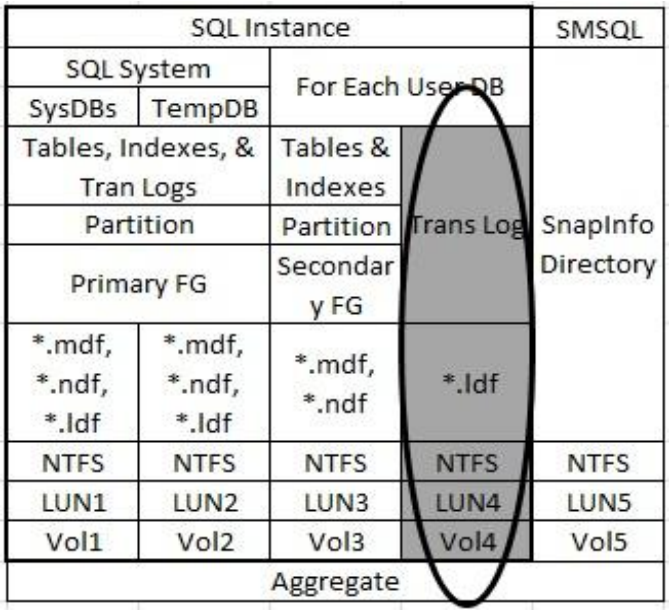

Figure 20) SQL Server database design for NetApp storage systems: transaction log on dedicated volume.

When using SnapManager for SQL Server, a variation of this design can be used. Rather than having the transaction logs on a completely dedicated LUN, they can instead be placed in the root of the SnapInfo LUN. This design would then change from having five volumes to having four volumes. This is a good design because, taking the separate SnapManager for SQL Server schedules described earlier, SnapInfo would get backed up each time the logs are backed up.

11.4 DESIGN EXAMPLE 4: MULTIPLE FILE GROUPS

The user databases in this example use six different file groups. The SQL Server instance has one aggregate, four volumes, and nine LUNs.

This shares the following design characteristics of example 3:

- Tempdb is on its own LUN.
- The user database transaction log file is separate from the data files.

The differences in this design are as follows:

- The user database transaction log shares the same LUN as the SnapInfo directory.
- The user database has three tables.

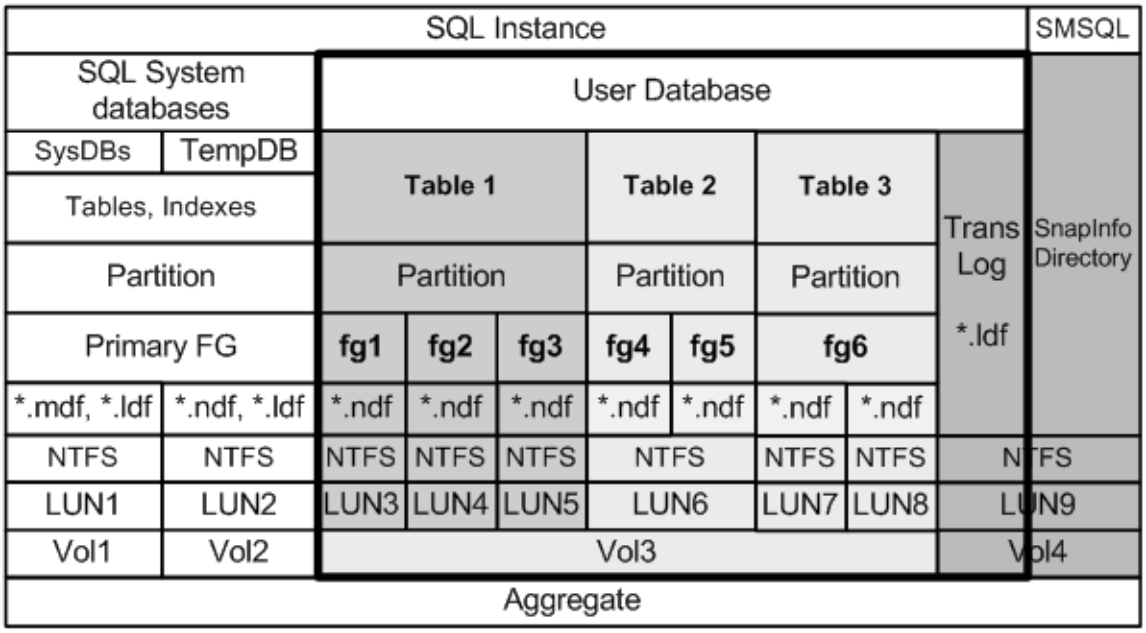

Figure 21) SQL Server database design for NetApp storage systems with multiple file groups.

Because all the tables share the same volume (Vol3), all the data will be captured in a single Snapshot copy. As before, the log file is separated from the data, but now the SnapManager for SQL Server SnapInfo directory will also be included in Vol4 Snapshot copies.

Let's look at each of the three tables:

Table 1

- This table uses three file groups.
- Each file group contains one file, and each of those files resides on its own LUN.

Table 2

- This example is included to demonstrate it can be done.
- This table uses two file groups.
- Each file group contains its own file, but all the files share the same LUN.

Table 3

- This table has only one file group.
- The file group contains two files.
- Each file resides on its own LUN.

With all three tables, the only realistic level of management is at the table level. Moving individual file groups, files, or LUNs would not make much sense.

11.5 DESIGN EXAMPLE 5: TABLE PARTITIONS

This example uses table partitions. Table 1 and table 2 are identical to table 1 and table 2 in example 4, except table 1 has been partitioned.

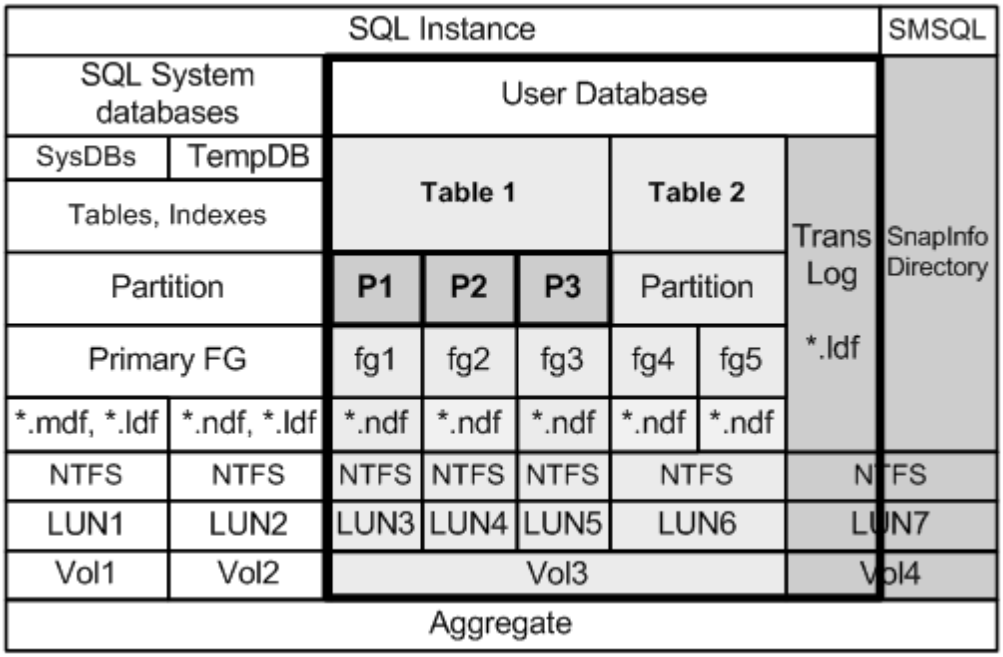

Figure 22) SQL Server database design for NetApp storage systems: table and index partitions.

Table 1

- In this design, the SQL Server engine is distributing the rows between the three partitions based on a range value.
- Each partition can be split off and used as an independent table without corrupting table 1.
- Because each partition is storage-aligned (each partition has a dedicated LUN), it can be migrated to different types of storage based on the business rules implied by the partition ranges without concern of corrupting the table. For example, P3 could be moved off to a SATA volume. It could be done with the database online.
- As discussed earlier in the partitioning section, each partition can have its own indexes, and nonclustered indexes could be located on their own LUNs.

12 SUMMARY

Microsoft SQL Server is an appropriate product for many business-critical applications. There are many different ways to design the underlying storage. This technical report has introduced how SQL Server stores its data on disk and has explored various options that can be used to design a storage solution on NetApp storage systems. To be successful, spend the time to identify and understand required service-level agreements. Then, utilize the SQL Server and NetApp storage features that have been discussed in this report to create a storage and data management design that meets the requirements.

Once a solution has been designed and implemented, run tests to establish a baseline. Make certain the environment performs as expected and save the results from the baseline tests for future reference. They can be very useful if the performance characteristics of the environment change. After deployment, monitor both SQL Server and the NetApp storage with a product such as Microsoft System Center Operations Manager using management packs from Microsoft for SQL Server and NetApp ApplianceWatch™ PRO to monitor NetApp storage systems.

NetApp has proven data protection and disaster recovery tools for Microsoft SQL Server. These include SnapManager for SQL Server for backup and restore and SnapMirror for disaster recovery. More information on these products can be found on www.netapp.com.

APPENDIX A: GLOSSARY

Aggregate: A manageable unit of RAID-protected storage consisting of one or two plexes that can contain one traditional volume or multiple FlexVol volumes.

Data ONTAP: Data ONTAP 7G is a highly optimized, scalable operating system that supports mixed NAS and SAN environments. It includes a patented file system, multiprotocol data access, and advanced storage virtualization capabilities. Data ONTAP 7G is NetApp's premier operating system for general-purpose enterprise computing environments. It is the default software platform that is shipped with all FAS and V-Series storage systems.

FlexVol volume: A FlexVol volume is a logical file system of user data, metadata, and Snapshot copies that is loosely coupled to its containing aggregate. All FlexVol volumes share the underlying aggregate's disk array, RAID group, and plex configurations. Multiple FlexVol volumes can be contained within the same aggregate, sharing its disks, RAID groups, and plexes. FlexVol volumes can be modified and sized independently of their containing aggregate.

Logical unit number (LUN): From the storage system, a LUN is a logical representation of a physical unit of storage. It is a collection of, or a part of, physical or virtual disks configured as a single disk. When you create a LUN, it is automatically striped across many physical disks. Data ONTAP manages LUNs at the block level, so it cannot interpret the file system or the data in a LUN. From the host, LUNs appear as local disks on the host that you can format and manage to store data.

Partitioned view: A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers, making the data appear as if from one table. Microsoft SQL Server 2005, 2008, and 2008 R2 distinguish between local and distributed partitioned views. In a local partitioned view, all participating tables and the view reside on the same instance of SQL Server. In a distributed partitioned view, at least one of the participating tables resides on a different (remote) server. In addition, SQL Server differentiates between partitioned views that are updatable and views that are read-only copies of the underlying tables. In SQL Server, the preferred method for partitioning data locally is through partitioned tables.

Proportional fill: SQL Server uses a proportional fill strategy across all the files within each file group and writes an amount of data proportional to the free space in the file. This enables the new file to be used immediately. In this way, all files generally become full at about the same time. However, transaction log files cannot be part of a file group; they are separate from one another. As the transaction log grows, the first log file fills, then the second, and so on, by using a fill-and-go strategy instead of a proportional fill strategy. Therefore, when a log file is added, it cannot be used by the transaction log until the other files have been filled.

Qtree: A qtree is a logically defined file system used to partition data within a volume or assign quotas to limit the amount of storage space a user is allowed to use. Every file or directory is always in a qtree, since the root of a volume is also a qtree. Other qtrees can be created as special directories in the root of a volume using the 'qtree' console command.

SnapDrive: SnapDrive for Windows software integrates with the Windows Volume Manager so that storage systems can serve as storage devices for application data in Windows Server environments. SnapDrive manages LUNs on a storage system, making this storage available as local disks on Windows hosts.

SnapMirror: This Data ONTAP feature enables you to periodically make Snapshot copies of data on one volume or qtree; replicate that data to a partner volume or qtree, usually on another storage system; and archive one or more iterations of that data as Snapshot copies. Replication on the partner volume or qtree makes sure of quick availability and restoration of data, from the point of the last Snapshot copy, should the storage system containing the original volume or qtree be disabled.

Snapshot copy: An online, read-only copy of an entire file system that protects against accidental deletions or modifications of files without duplicating file contents. Snapshot copies enable users to restore files and to back up data to tape while the NetApp storage system is in use.

WAFL - Write Anywhere File Layout. The WAFL® file system was designed for the NetApp storage system to optimize write performance. The storage system uses the WAFL blocks-based file system to manage file access and storage system performance. WAFL is a block-based file system that uses inodes to describe files. It uses 4-KB blocks with no fragments.

APPENDIX B: REFERENCES

This section lists useful resources that will assist you in planning and managing your SQL Server storage environment.

- NetApp Storage Systems
  www.netapp.com/us/products/storage-systems/
- Data ONTAP documentation
  http://now.netapp.com/NOW/knowledge/docs/ontap/ontap_index.shtml

Additional documentation available from the NetApp Support site:

- TR-3411: Database Layout with Data ONTAP 7G
- SnapDrive for Windows Installation and Administration Guide
- SnapManager for SQL Server Installation and Administration Guide
- Microsoft SQL Server Customer Advisory Team: resources for complex enterprise SQL Server implementations
  http://sqlcat.com/
- Microsoft SQL Server Storage Engine Blog
  http://blogs.msdn.com/sqlserverstorageengine/default.aspx
- MSDN: Product documentation, including SQL Server Books Online
  http://msdn.microsoft.com/en-us/library/bb545450.aspx
- SQL Server Best Practices
  http://technet.microsoft.com/en-us/sqlserver/bb671430.aspx
- SQL Server Hardware and Software Requirements
  http://technet.microsoft.com/en-us/library/ms143506.aspx