SQLCAT's Guide to: Relational Engine

Microsoft SQLCAT Team

Guide & Reference

Microsoft

SQLCAT's Guide to: Relational Engine

Microsoft SQLCAT Team

Summary: This ebook is a collection of the most popular technical notes, tools and blogs authored by the SQLCAT team and posted to their blog over the course of several years. It covers SQL technology from 2005 to 2012.

Category: Guide & Reference
Applies to: SQL Server 2005 to 2012
Source: SQLCAT Blog
E-book publication date: September 2013

Contents

Section 1: Administration

DBCC Checks and Terabyte-Scale Databases

1. Overview:

Historically, the SQL Server product team and the support organization have recommended that all maintenance plans for Microsoft® SQL Server® include running DBCC CHECKDB on a regular basis to detect and correct potential corruption in the database. In the case of mission-critical applications such as trading systems, medical records, and banking operations, many customers have interpreted this recommendation as requiring even a *daily* DBCC check to ensure that no errors are present before performing a backup operation.

However, since the SQL Server 6.5 days, in which a daily DBCC CHECKDB may have been a best practice, there have been several trends - both challenges and opportunities -- that make it worth reconsidering the way that DBCC should fit into SQL Server database maintenance plans, especially for large databases.

Challenges:

- Many mission-critical databases have grown to massive size. It is not unusual to find databases in the 5-20 terabyte range on SQL Server, in which a DBCC operation requires many hours to complete. This is usually incompatible with a daily maintenance window, if feasible at all on a 24x7 system.
- Every new version of SQL Server has expanded the range of logical checks performed by DBCC so that it has become more time-consuming to execute. For example, SQL Server 2005 introduced data purity checks to validate that data present in columns adheres to valid ranges for the data type. SQL Server 2005 also introduced checks that verify the integrity of indexed views.

Opportunities:

- Enterprise-class storage subsystems and the widespread use of RAID, especially in mission-critical settings, have made the storage tier much less prone to physical corruption.
- SQL Server has introduced new mechanisms to detect the most frequent forms of physical corruption independent of DBCC. For example, torn page detection was introduced in SQL Server 7.0, checksum verification of physical pages was introduced in SQL Server 2005, and checksum protection of **tempdb** pages was introduced in SQL Server 2008.
- We continue to invest in technologies to automatically detect and correct sources of corruption - such as the introduction in SQL Server 2005 of an in-memory page scan to find checksum violations due to ECC failures that might occur during manipulation of read-only pages.

As more customers are deploying databases at the scale of terabytes or tens of terabytes, frequent DBCC checks are becoming less practical. But the many advances in both SQL Server and enterprise-class hardware offerings are enabling customers to back-away from the "Daily DBCC" best practices of the SQL Server 6.5 era, while still maintaining high confidence in the integrity of their data.

2. Suggestions for customers

DBCC CHECKDB remains an important tool for detecting and correcting logical consistency problems and physical corruption in the database. However, for large-scale databases utilizing a high quality SAN or storage subsystem, the specific recommendations this technical note presents can reduce the frequency of DBCC and certainly relax the prior standard of running such checks on daily basis.

SQL Server 2005 introduced an important new mechanism to proactively detect physical corruption in database pages: a database option that adds a checksum to each page as it is written to the I/O system and validates the checksum as it is read. **This database option is called PAGE_VERIFY = CHECKSUM, and utilizing it by default is a powerful alternative to scheduling frequent runs of DBCC CHECKDB.** For more information, see ALTER DATABASE SET Options (Transact-SQL) in SQL Server Books Online: http://msdn.microsoft.com/en-us/library/bb522682.aspx.

A. Recommendations for new databases (created in SQL Server 2005 or SQL Server 2008):

We encourage customers who have created their databases in SQL Server 2005 or SQL Server 2008 to rely on a strategy of using this PAGE_VERIFY = CHECKSUM feature and reduce the scheduled use of DBCC to a minimum. Below are some guidelines on how to accomplish this.

1) Use PAGE_VERIFY = CHECKSUM on all new databases created in SQL Server 2005 or later (this is the default setting for new databases).

2) Perform daily incremental or differential backup.

By performing an incremental/differential BACKUP WITH CHECKSUM operation each day (or even a full backup if feasible), you are guaranteed to read each page that has been added or modified since the prior backup. Because the reads performed during backup will validate the page checksums, any corruption that has occurred due to I/O errors

will be detected, and can be corrected by using the current backup set to restore corrupt pages. (If the database is using the bulk-logged or full recovery model, page-level restore can be used to minimize time for recovery.)

The steps above will ensure that most physical page corruption due to I/O errors will be detected on a daily basis and can be promptly corrected. Because I/O errors have historically been the largest source of database corruption, PAGE_VERIFY = CHECKSUM can significantly reduce the frequency of DBCC operations needed to maintain a high level of confidence in the integrity of the physical database.

There are two other sources of corruption that can still arise even if PAGE_VERIFY = CHECKSUM is used, although they are rare - and they are problems that only DBCC CHECKDB can discover proactively:

a) 'Scribbler' induced errors - where in-memory pages are corrupted either by third-party code running inside the SQL Server process or by drivers or other software with sufficient privileges executing in Windows® kernel mode.

SQL Server allows a variety of customer or third-party code to access the SQL Server address space, including extended stored procedures (XPs), unsafe SQL CLR assemblies, EKM providers, and OLE DB drivers for linked servers. Scribbler errors can arise as a result of bugs in these extensions, or from software such as malware protection tools or I/O drivers running in the kernel.

b) Potential SQL Server bugs that create logical errors.

We are not aware of any bugs in the current version of SQL Server that lead to logical errors in database objects, but the logical checks in DBCC can locate problems due to legacy errors or unknown errors in current versions of the product. Logical errors can usually be repaired by rebuilding indexes or re-establishing foreign key constraints - they are not typically fixed by restoring from a backup.

In addition, CHECKSUM cannot detect I/O problems if a page header itself is corrupt, but DBCC CHECKDB can.

These and other additional sources of errors, external to SQL Server, are reasons why customers should not eliminate DBCC CHECKDB entirely from maintenance plans. But the fact that the majority of errors located by DBCC CHECKDB result from I/O channel problems that are effectively caught using CHECKSUM allows customers to reduce the frequency of complete DBCC CHECKDB executions.

B. Recommendations for migrated *databases:*

If you are using a database that was created *prior* to SQL Server 2005, or if you are using a database that was created with a more recent version but without using PAGE_VERIFY = CHECKSUM, you need to be aware that checksums are added only as pages are added or modified. An initial run of DBCC CHECKDB using the PHYSICAL_ONLY option, after enabling the PAGE_VERIFY = CHECKSUM option, will check the integrity of all legacy pages but will not add a checksum.

Is there any risk that *legacy* pages from a migrated database can become corrupt and will not be detected by the checksum mechanism? It turns out that there are rare circumstances in which data on storage media can become corrupted in the absence of I/O ("bit rot") and not be detected by parity or RAID redundancy mechanisms, especially on low-end storage systems. If this occurs on a page *without* a SQL Server checksum, it can lead to database corruption that can only be detected by regular PHYSICAL_ONLY checks of DBCC CHECKDB.

So, in order to eliminate frequent DBCC CHECKDB runs for a database that was *not* originally created with PAGE_VEIRFY = CHECKSUM, we recommend either:

- Rebuilding all clustered indexes and recreating any heaps. This will ensure that all data and index pages are written once with CHECKSUM enabled.
- Or, less practically, creating a new database using PAGE_VEIRFY = CHECKSUM and migrating all objects to the new database.

The alternative, in a *legacy* database for which not all pages have a CHECKSUM, is to run DBCC CHECKDB with the PHYSICAL_ONLY option regularly prior to taking any *full* backup. If a problem is detected on a non-checksum page, the prior backup set can be used to restore a correct image of that page.

3. Periodic DBCC Strategies

Even if *daily* DBCC checks need no longer be part of an enterprise-class maintenance strategy, *periodic* DBCC checks remain an important tool. Frequency of DBCC checks should take into account whether scribbler errors are possible due to third party extensions running in-process, the kinds of additional privileged software running on the server, and the reliability and sophistication of the storage tier.

There are a variety of strategies to make DBCC compatible with shorter maintenance cycles and high availability production environments. Among those that are popular for large-scale, mission-critical deployments are:

- Utilizing a SAN-based snapshot, mounted to another instance of SQL Server that runs DBCC independently of the production system
- Performing DBCC on a per-filegroup basis, on a rotating schedule
- Using Resource Governor to adjust the degree of parallelism (MAXDOP) of the session running DBCC operations, either to run highly parallelized during a short maintenance window, or single-threaded in the background during non-critical production hours
- Using DBCC to perform the faster physical page validation checks (PHYSICAL_ONLY) more frequently than the logical checks. Such tests will bypass time-consuming validation of foreign key references and nonclustered indexes, but will ensure that all allocated pages are readable and will detect the majority of problems that can arise from problems in the I/O channel.

Some of these, and other strategies for executing DBCC operations, can be found in the recent blogs by Paul Randal and Bob Dorr, at:

http://www.sqlskills.com/blogs/paul/post/CHECKDB-From-Every-Angle-Consistency-Checking-Options-for-a-VLDB.aspx

http://blogs.msdn.com/psssql/archive/2009/02/20/sql-server-is-checkdb-a-necessity.aspx

4. Conclusion

While the reliability of SQL Server has improved dramatically over the past decade, database integrity still depends on the reliability of the storage tier and can be influenced by high-privileged, third-party code running in Windows kernel mode or permitted inside the SQL Server address space. SQL Server has invested in significant technologies to automatically detect and correct problems with data observed on-disk and in-memory, and this has reduced the need for frequent complete database integrity checks using DBCC CHECKDB. Periodic scheduled DBCC operations remain a best practice, but not at the near-daily frequency recommended in the past for mission-critical deployments. The recommendations here for using CHECKSUM features, along with periodic DBCC CHECKDB strategies, can help achieve a balance between a reasonable maintenance cycle and high confidence in database integrity.

Scheduling Sub-Minute Log Shipping in SQL Server 2008

Overview

Log shipping allows you to automatically take transaction log backups on a primary server, send them to one or more secondary servers, and then restore the transaction log backups on each secondary server. Many Microsoft SQL Server customers have asked for the ability to schedule the log shipping jobs with less than 1 minute frequency. In SQL Server 2005, SQL Server Management Studio user interface allowed the frequency of the scheduled jobs to be 1 minute or more, which meant that the minimum latency of log shipping was as long as 3 minutes (1 minute each for the backup, copy, and restore jobs). Many customers have asked for this latency to be less than 1 minute.

In this paper we introduce the new sub-minute log shipping capability in SQL Server 2008, and we discuss some considerations you need to be aware of in scheduling frequent log shipping jobs.

Introducing Sub-Minute Log Shipping in SQL Server 2008

SQL Server 2008 enables log shipping jobs to be scheduled with frequency in seconds. In SQL Server 2008, SQL Server Management Studio and the stored procedures sp_add_jobschedule and sp_add_schedule allow frequency settings in seconds, minutes, and hours. The minimum frequency is 10 seconds.

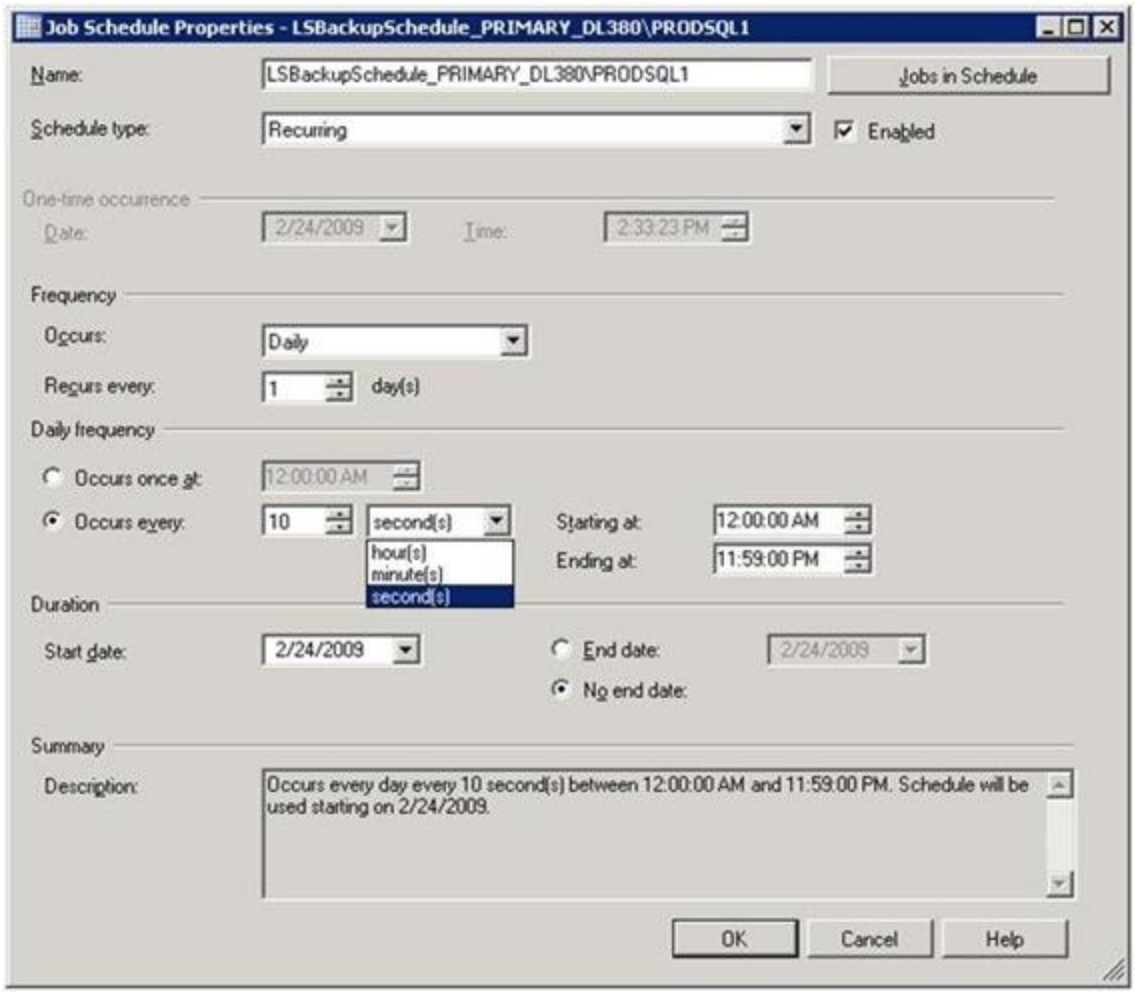


Figure 1: SQL Server 2008 Log Shipping user interface enables scheduling the jobs in hour, minute, or second frequency

Considerations

There are some considerations you should be aware of when you set up too frequent log shipping jobs:

- The next execution of the job will not start until the previous execution has completed. Let's assume you have set the frequency interval of the log backup job to 10 seconds, but one execution of the log backup takes 12 seconds to complete. The next backup job will start at the next scheduled time, which is 20 seconds after the start of the previous backup job. One execution of the job is skipped in this case.
- Every time a log backup is completed, a message similar to the following is shown in the SQL Server ERRORLOG:

2009-02-09 15:25:56.94 Backup Log was backed up. Database: Test_LS, creation date(time): 2009/02/09(14:27:24), first LSN: 19:145:1, last LSN: 19:145:1, number of dump devices: 1, device information: (FILE=1, TYPE=DISK: {'\\PRIMARY_DL380\LSBackup\Test_LS_20090209232551.trn'}). This is an informational message only. No user action is required.

If you take a log backup every 10 or 15 seconds, the SQL Server ERRORLOG flooded with such messages.

If you don't want these messages flooding the SQL Server ERRORLOG, you can enable trace flag 3226. This trace flag doesn't alter the behavior of backup jobs; it just suppresses the backup completion messages, preventing them from getting into the SQL Server ERRORLOG. Note that this trace flag suppresses all backup messages – database backup as well as transaction log backup.

- Information about each backup is also recorded in the **msdb** database (the msdb.dbo.backupset, msdb.dbo.backupmediaset, msdb.dbo.backupmediafamily, msdb.dbo.backupfile, and msdb.dbo.backupfilegroup system tables). If you back up too frequently, you can expect these tables to grow faster than usual. You should periodically check the size of these tables and delete or archive the old information as necessary. To delete the old backup history, use the stored procedure sp_delete_backuphistory.
- The backup compression feature in SQL Server 2008 provides significant space and time savings. Backup compression results in smaller backups, and it helps improve the performance of all the operations performed by log shipping by providing the following:
    - Faster backup of the transaction log on the primary server.
    - Faster copying of the transaction log backup file to the secondary over network.
    - Faster restore of the log backup on the secondary.

However, the benefits of backup compression come with the cost of higher CPU utilization. If your log backup jobs use compression and are scheduled too frequently, you may notice frequent spikes in CPU utilization on the primary. Restoring from a compressed backup uses more CPU, and you could see frequent spikes in CPU utilization on the secondary as well. For more information about backup compression, see Tuning the Performance of Backup Compression in SQL Server 2008 and Tuning Backup Compression Part 2.

Conclusion

SQL Server 2008 provides the ability to schedule log shipping jobs as frequently as 10 seconds, which results in reduced latency of log shipping. Reduced log shipping latency can result in reduced data loss in case of loss of primary.

Tuning Backup Compression Part 2

Overview

This is the second part of the article Tuning the Performance of Backup Compression in SQL Server 2008 In the first part we described the benefits of backup compression, a methodology on how to tune backup compression for best performance, and shared some best practices. In this second part, we describe some more considerations in tuning backup compression, and how backup compression interacts with other important features in Microsoft SQL Server 2008. Specifically, we will discuss the following:

- Tuning BUFFERCOUNT and MAXTRANSFERSIZE
- Memory used by backup compression
- Backup compression and log shipping
- Backup compression and data compression
- Backup compression and transparent data encryption

Understanding the tuning techniques and the interoperability of backup compression with other features discussed in this article can help you get the best out of the backup compression feature.

Tuning BUFFERCOUNT

As described in Part 1 of the article, default BUFFERCOUNT is determined by SQL Server based on the number of database volumes and backup devices. If the database files are spread across several disk volumes and/or there are a large number of backup devices, the default BUFFERCOUNT value may provide optimal backup performance, and you may not need to tune BUFFERCOUNT further. As discussed in Part 1, you can use the trace flags 3605 and 3213 to find out the default BUFFERCOUNT value used in your backup. However, if the database files are spread across too few disk volumes and/or there are a small number of backup devices, the default BUFFERCOUNT value may not provide optimal backup performance. Tuning BUFFERCOUNT explicitly may improve backup performance.

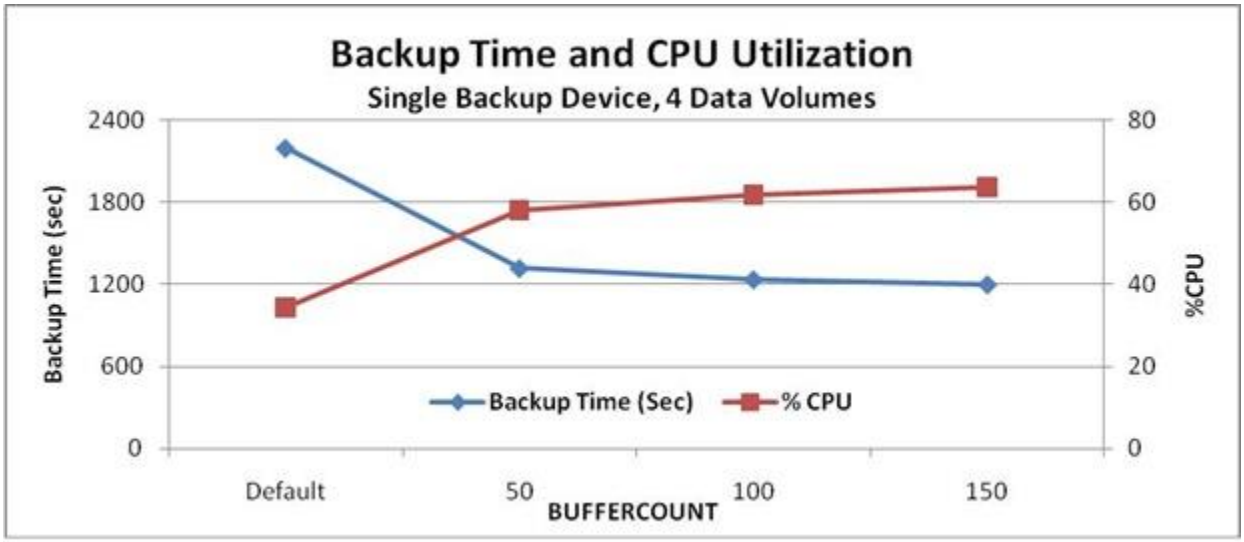


Figure 1: Backup time and CPU utilization with varying BUFFERCOUNT

As illustrated in Figure 1, increasing BUFFERCOUNT results in reduced backup time, at the cost of higher CPU utilization. Your results may vary depending upon your database size, storage layout, and server capacity; however, you will notice that the impact of increasing BUFFERCOUNT on backup performance tends to reduce as BUFFERCOUNT is increased beyond a certain value. In our test, the curve tends to flatten out as we increase BUFFERCOUNT beyond 50.

BUFFERCOUNT value impacts the amount of memory used for backup (discussed later in the section "Memory Utilization by Backup"). Keep this in mind if you explicitly specify the BUFFERCOUNT value.

Tuning MAXTRANSFERSIZE

MAXTRANSFERSIZE refers to the size of the I/O operation issued to read data from the database files. The default value of MAXTRANSFERSIZE is 1 MB. Performance of sequential I/O operations generally benefit from larger block sizes, which is the reason the value is set to 1 MB by default. One drawback of larger I/O sizes is the potential impact on performance of the smaller I/Os being issued concurrently by an OLTP workload. Because I/O queue structures are shared, intermixing large I/O sizes with smaller concurrent I/O requests results in increased latency for both. In today's shared storage network environments, there is potential for these operations to also impact other hosts sharing the same physical devices. Tuning this parameter is likely unnecessary in hardware configurations using dedicated storage, and it may be necessary only if it is determined that the backup operations impact concurrent workloads. As recommended in Part 1 of the article, tuning MAXTRANSFERSIZE should be considered as a secondary tuning option, and it should only to be utilized when it is determined to be beneficial through testing. In the majority of deployments, the default value will be acceptable.

Figure 2 illustrates observations of backup performance using various transfer sizes.

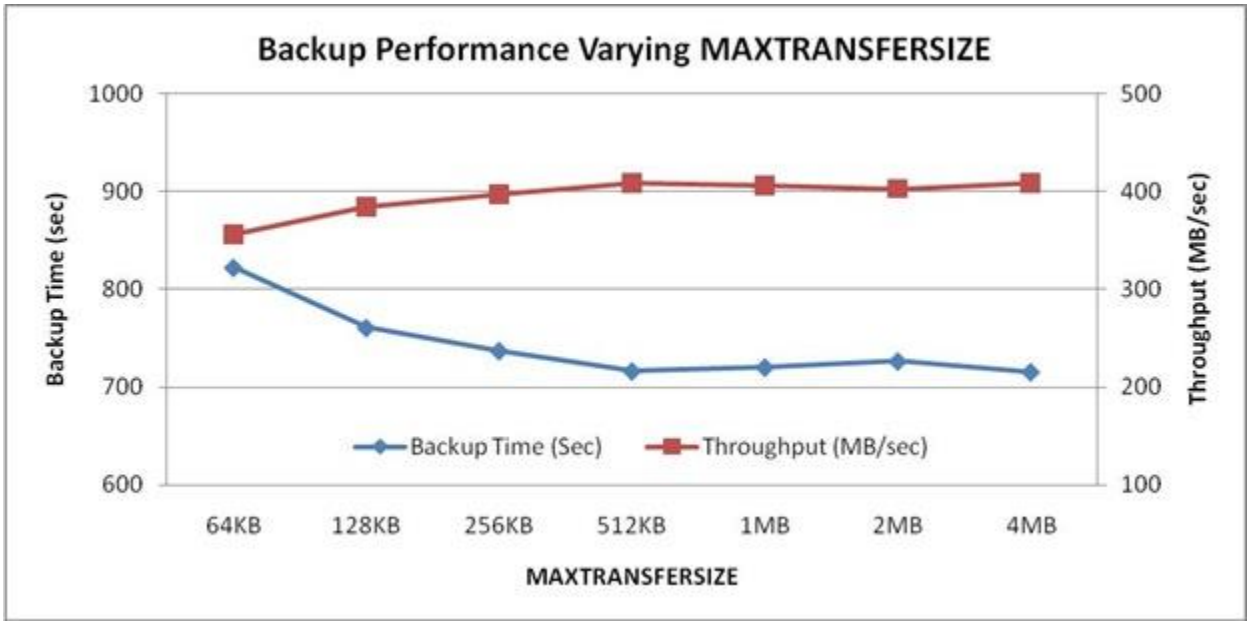


Figure 2: Backup time and throughput with varying MAXTRANSFERSIZE (BUFFERCOUNT = 50, 1 Backup Device)

As illustrated in Figure 2, smaller MAXTRANSFERSIZE (64 KB) results in lower backup throughput and hence longer backup time, and as you increase MAXTRANSFERSIZE, you observe reduced backup time and higher throughput. However, the impact of increasing MAXTRANSFERSIZE on backup performance tends to reduce as MAXTRANSFERSIZE is increased beyond a certain value. In our tests, we observed optimal backup performance when MAXTRANSFERSIZE was between 128 KB and 512 KB. Your results may vary based on your I/O configuration (throughput and latency of your I/O subsystem).

Similar to the BUFFERCOUNT setting, the value chosen for MAXTRANSFERSIZE will also impact the amount of memory used for backup operation (discussed later).

Memory Utilization by Backup

As discussed in Part 1 of the article, memory used by backup buffers comes from virtual address space outside the buffer pool. On 32-bit systems, there is a fixed amount of virtual address space set aside outside of the buffer pool (default of 384 MB); increasing BUFFERCOUNT and MAXTRANSFERSIZE options to high values may fail because the memory for backup operations is calculated and set aside at the beginning of the operation.

On 64-bit systems, the virtual address space for any process can be up to 8 TB which is far beyond the physical memory supported on current 64-bit version of Windows. As a result, memory for allocations outside the buffer pool does not have to be set aside at the time SQL Server is started and is potentially unlimited. Setting an appropriate value for 'max server memory' using SP_CONFIGURE is recommended on 64-bit deployments of SQL Server to ensure that enough physical memory will be available to support allocations outside the buffer pool for backup operations.

Trace flags 3605 and 3213 can be used to report the number of buffers used for backup operations to the SQL Server ERRORLOG. The following example shows the information that is reported in the ERRORLOG when these trace flags are enabled.

```
LogDate                 ProcessInfo Text

2009-01-16 12:42:13.700 spid53      Backup/Restore buffer configuration
parameters

2009-01-16 12:42:13.700 spid53      Memory limit: 32761MB

2009-01-16 12:42:13.700 spid53      BufferCount: 100

2009-01-16 12:42:13.700 spid53      MaxTransferSize: 2048 KB

2009-01-16 12:42:13.700 spid53      Min MaxTransferSize: 64 KB

2009-01-16 12:42:13.700 spid53      Total buffer space: 200 MB

2009-01-16 12:42:13.700 spid53      Tabular data device count: 1

2009-01-16 12:42:13.700 spid53      Fulltext data device count: 0

2009-01-16 12:42:13.700 spid53      Filestream device count: 0
```

```
2009-01-16 12:42:13.700 spid53      TXF device count: 0

2009-01-16 12:42:13.700 spid53      Filesystem i/o alignment: 512

2009-01-16 12:42:13.700 spid53      Media Buffer count: 100

2009-01-16 12:42:13.700 spid53      Media Buffer size: 2048KB

2009-01-16 12:42:13.700 spid53      Encode Buffer count: 100

2009-01-16 12:45:41.820 Backup      Database backed up. Database: TESTPART,
creation date(time): 2008/11/06(12:31:03), pages dumped: 1295126, first LSN:
158:223:37, last LSN: 158:239:1, number of dump devices: 1, device
information: (FILE=1, TYPE=DISK: {'R:\Backup\TESTPART_Compressed.bak'}). This
is an informational message only. No user action is required.
```

For an uncompressed backup, the total memory used by the backup buffers can be computed as BUFFERCOUNT multiplied by MAXTRANSFERSIZE. Compressed backup needs three sets of buffers – one set of buffers are used to read from the database files, the second set of buffers are used to compress the data, and the third set of buffers are used to write to the backup media. Therefore, a compressed backup will utilize three times as much memory as the uncompressed backup.

You can observe the memory used by the backup task by monitoring the “Process:Private Bytes” counter for the “sqlservr” process in the Reliability and Performance Monitor (also known as Perfmon). Figure 3 displays this Perfmon counter for a compressed backup, with no other workload. As illustrated in Figure 3, you will see an increase in the Private Bytes counter during the backup task. This increase is equal to (BUFFERCOUNT * MAXTRANSFERSIZE) for uncompressed backups, and equal to (3 * BUFFERCOUNT * MAXTRANSFERSIZE) for compressed backups.

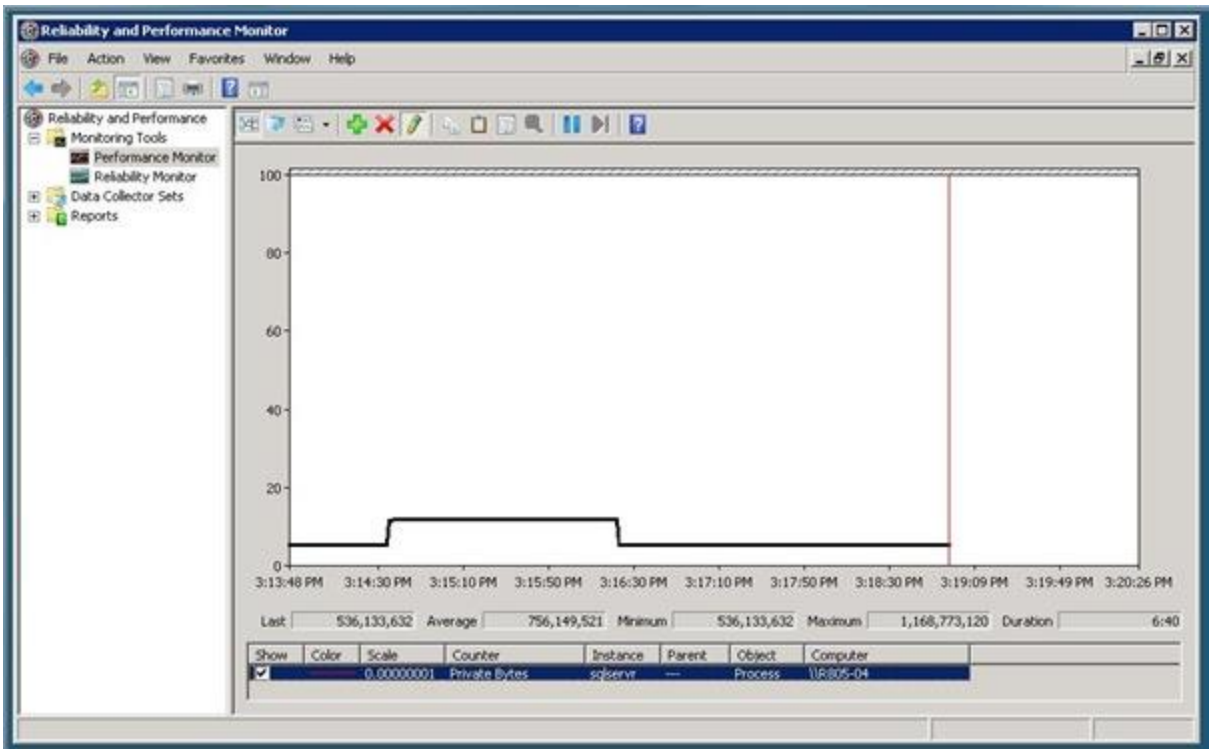


Figure 3: Memory used by a backup compression operation, as measured by Perfmon

Another observation from Figure 3 is that this memory is allocated at the beginning of the backup operation, and it is released when the backup is complete.

Backup Compression and Log Shipping

Performance of log shipping will also benefit from compressed backups. Log shipping sends transaction log backups from a primary server to a secondary server by copying log backup files to a network share to be applied to the secondary server. When backup compression is used, transaction log backups are compressed. The reduction in file size for log backups improves performance of all the operations performed by log shipping:

1. Backup the transaction log on the primary server.
2. Copy the transaction log backup file to the secondary server over the network.
3. Restore the log backup on the secondary server.

The transaction log backups during log shipping can be compressed in one of the following two ways:

- If you are using SQL Server Management Studio to setup log shipping, set the backup compression setting by selecting **Compress backup** from the **Set backup compression** list, as shown in Figure 4. If you have set backup compression as the default server level setting (sp_configure option backup compression default), you can pick the "Use default server setting" option as well.
- If you are using the stored procedures to set up log shipping, set the parameter @backup_compression of stored procedure sp_add_log_shipping_primary_database to 1. If you have set backup compression as the default server level setting, you can set this parameter to 2.

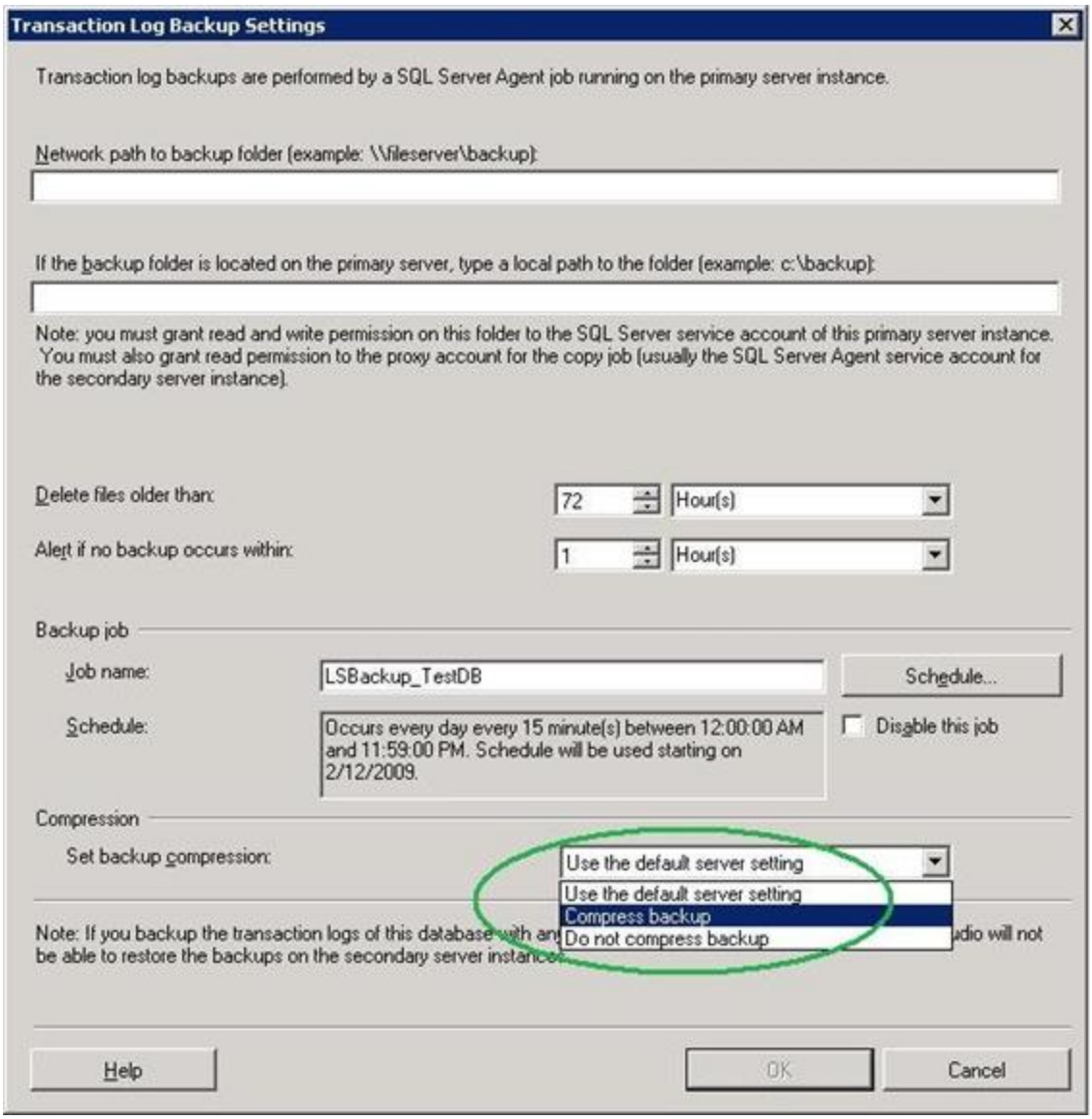


Figure 4: Compressing transaction log backups for log shipping

Figure 5 provides a data point from a customer deployment. The compressed log backup is significantly smaller and faster.

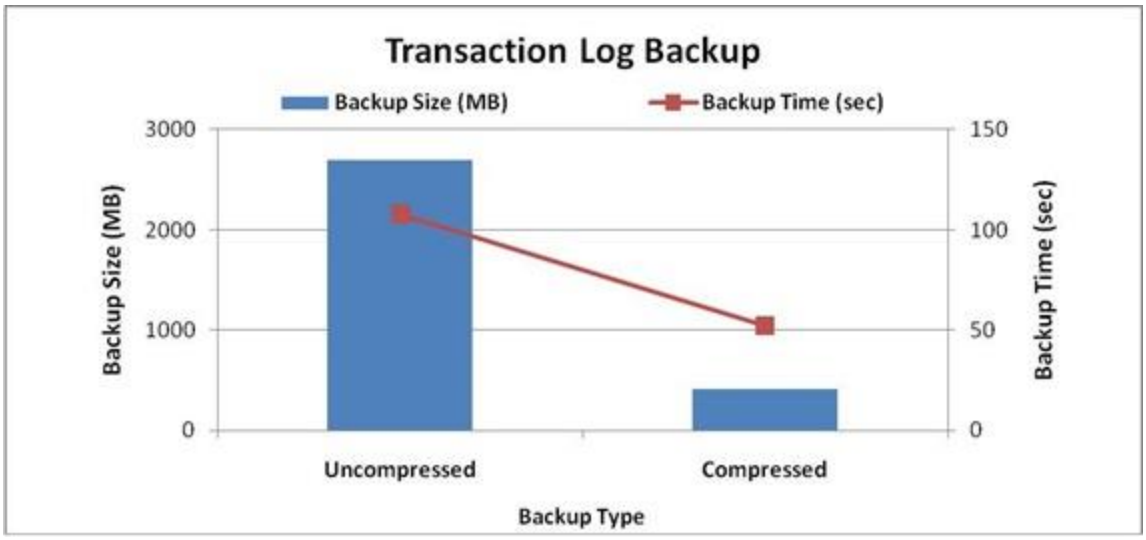


Figure 5: Compression of transaction log backup for log shipping (default values for BUFFERCOUNT and MAXTRANSFERSIZE, 1 backup device)

Backup Compression and Data Compression

Data compression is a feature of SQL Server 2008 that can save disk space by compressing data pages within the database. A commonly asked question is "Are there additional benefits realized by the use of compressed backup operations when data compression is used (both compression ratio and performance of the backup operation)?"

In this test, all tables and indexes were compressed in the database, and then the performance of backup compression was measured. Separate tests were run with no compression, ROW compression applied to all tables/indexes, and PAGE compression applied to all tables/indexes. Figure 6 compares the size of the compressed backup, backup compression ratio, and backup time for databases that contain NONE, ROW, and PAGE compressed tables and indexes.

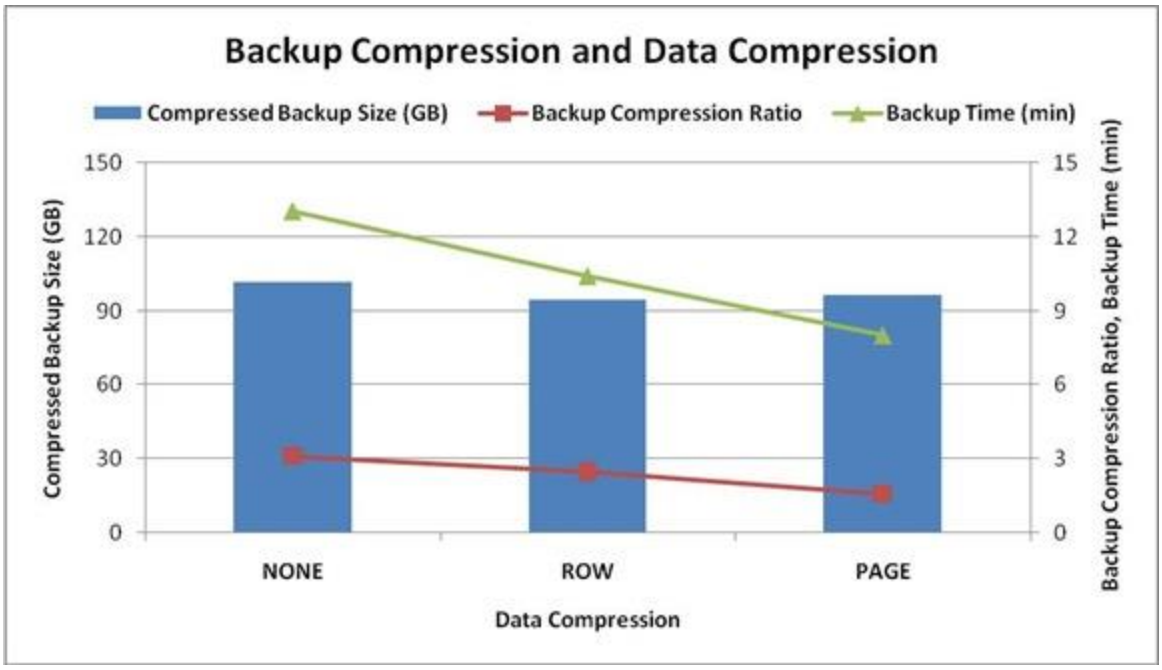


Figure 6: Backup compression with data compression (BUFFERCOUNT = 50, MAXTRANSFERSIZE = default, 4 backup devices)

Some observations from the results of Figure 6:

- Backup compression can result in additional disk space savings even on databases that contain ROW or PAGE compressed tables or indexes. The size of the compressed backup and the backup compression ratio depend upon the characteristics of the data in the database, and they can vary from the results shown in these examples.
- Backup operations on databases that use ROW or PAGE compression will likely result in shorter backup times, because the smaller database size translates into less I/O.
- CPU consumption during the backup operation for databases that use ROW or PAGE compression may be higher as a result of less I/O, which results in more CPU time for compression operations.

Backup Compression and Transparent Data Encryption

Transparent data encryption (TDE) is another very useful feature in SQL Server 2008. TDE provides encryption of data in a database at the storage level without requiring any application changes. A common question related to this is "How does backup compression perform against an encrypted database?"

In the example below, backup compression was performed against a database with TDE enabled. Figure 7 compares the size of the backup, CPU consumption, and backup time for compressed and uncompressed backups on the TDE-enabled database.

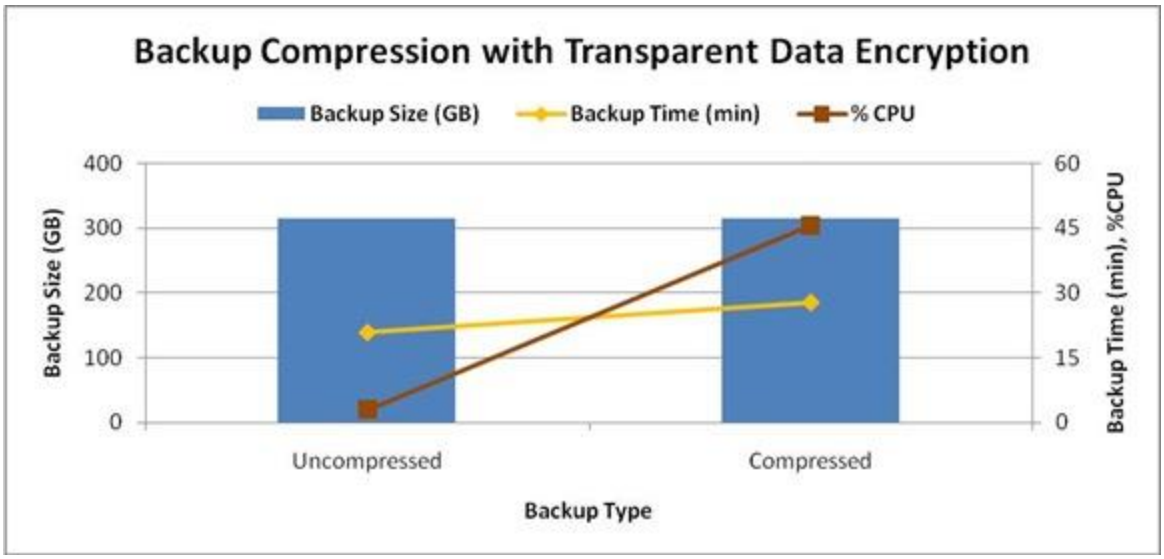


Figure 7: Backup compression with TDE (BUFFERCOUNT = default, MAXTRANSFERSIZE = default, 1 backup device)

We made the following observations when we performed compressed backup against TDE-enabled databases:

- On a TDE-enabled database, backup compression doesn't help reduce the size of the backup. The backup compression ratio is nearly 1.0, independent of the data in the database. This is due to the fact that encrypted data does not lend itself well to backup compression.
- CPU utilization for the compressed backup is higher than the uncompressed backup, even though the backup size is not much different. This is because CPU resources are wasted in the compressed backup operation, because it attempts to compress the data, even though the data is not very compressible.
- On a TDE-enabled database, it takes longer to perform a compressed backup than it takes to perform an uncompressed backup. This is due to the fact that I/O operations are not reduced, because the data does not compress well. However, there is time spent attempting to compress the incoming data.

For these reasons, we do not recommend the use of backup compression on a TDE-enabled database.

Conclusion

Backup compression is one of the most popular features in SQL Server 2008 Enterprise. Most SQL Server deployments will benefit from this feature; it can reduce both the time taken to perform the backup operation and the disk space required to store database backups. Understanding the tuning techniques and considerations described in Part 1 of the article as well as the interoperability of backup compression with other features discussed in this article can help you get the best out of the backup compression feature.

Appendix A: Test Hardware and Software

All tests (except those in Figure 4 and Figure 6) were performed on the following hardware and software environment.

Server

HP DL380G5 with:

- 2 socket quad core
- Intel Xeon CPU E5345 @2.33 GHz
- 16 GB RAM

Storage

EMC Symmetrix DMX-4 2500

- Data volumes
    - 4 data volumes from a disk group with
        - 32 disk drives, 300 GB each @15K RPM
        - RAID 1+0
- Backup volume
    - 1 backup volume from a disk group (separate from data volumes) with
        - 32 disk drives, 300 GB each @15K RPM
        - RAID 1+0
- ~2 HBAs (4 Gb Fiber Channel)

Software

- The 64-bit edition of Windows Server 2008 Enterprise
- The 64-bit edition of SQL Server 2008 Enterprise

Restart SQL Audit Policy and Job

As noted within the Reaching Compliance: SQL Server 2008 Compliance Guide (you can also check the sqlauditcentral codeplex project), an easier way to view and manage all of the audit logs within your SQL Server environment is to place all of the audit logs in one central location. As per the guide, you can then use a SSIS package to import in all of these logs files into a separate SQL database where you can then generate reports to view all of the audits within your entire SQl Server environment.

The problem that we recently discovered is that if SQLAudit loses connectivity to the folder it places the audit files, provided that you did not tell SQL Server to shutdown if it cannot write an audit:

- The audit's is_state_enabled column in sys.server_audits will remain 1, meaning true, but the audit status in sys.dm_server_audit_status will be "RUNTIME_FAILED" and no events will be written to the audit log.
- Even when connectivity to the folder has returned, the audit will remain in the "RUNTIME_FAILED" state - meaning it still tries to write to the log but will always fail as it is using an old and now invalid handle, or reference, to the audit log from before the connectivity loss. Currently the only way to get the audit to create a new valid handle for the audit log is to stop and restart the audit – which will create a new audit file.

There is a bug assigned to this issue and will be resolved in the future. But for us whom are working with SQL Audit right now, to work around this problem, please go to the sqlauditNetworkConnectivity Codeplex project where you can download the full Centralized Audit Framework project. Within this project is the **Restart SQL Audit Policy and Job** folder. This folder contains three pieces of source code:

- **Server Audit Status (Started).xml** - Import this on-schedule policy into your server's Policy-Based Management as it will determine if the audit is enabled and able to write to the file system.
- **Create Audit Job.sql** - This is a SQL Server Audit job that will execute the noted policy; you will need to schedule this yourself
- **Create Audit Alert.sql** - This is a SQL Server Audit job that will create an audit alert.

Together these three source components will (whenever manually executed or scheduled) determine if all of the audits on your server are able to write to the folder. If they are not, they will send out an alert as well as stop and restart the audit job re-initializing it so that way the audits will start writing again.

SQL DMVStats Toolkit

http://www.codeplex.com/sqldmvstats

A SQL Server 2005 Dynamic Management View Performance Data Warehouse

Introduction

Microsoft SQL Server 2005 provides Dynamic Management Views (DMVs) to expose valuable information that you can use for performance analysis. DMVstats 1.0 is an application that can collect, analyze and report on SQL Server 2005 DMV performance data. DMVstats does not support Microsoft SQL Server 2000 and earlier versions.

Main Components

The three main components of DMVstats are:
• DMV data collection
• DMV data warehouse repository
• Analysis and reporting.
Data collection is managed by SQL Agent jobs. The DMVstats data warehouse is called DMVstatsDB. Analysis and reporting is provided by means of Reporting Services reports.

For more details, refer to the file attachment DMVStats.doc.

Section 2: Database Design

Character data types versus number data types: are there any performance benefits?

Introduction

Working on a recent project, I observed that some developers choosing between character and number data types favored character data types. However, in my experience, this is not always the best choice. In the example I discuss here, number data types turned out to the better option. This paper describes a recent case in which we redesigned the data warehouse of a telecommunications company. As part of the design process, we ran a series of tests to compare performance of the two data types.

One of the tables had to store phone numbers in international format: + 1 234 5678910, where +1 was the country code and 234 was the area code. In this case, using the character data type simplifies development - you can use either of the following formats for storing the country code: the '+' or the leading '00'.But for comforts in our life we always have to pay. Is this also true in this case? There are considerations other than ease of development. Wouldn't storage size, compressibility, and improved query performance outweigh the evident development simplicity?

What do we need to store really?

Before I discuss the tests we used to determine the benefits of using either numeric or character data, I would like to talk about a best practice that helps solve many problems in designing databases, and in deciding which data type to use.

This best practice is simple: talk to the business, and find out what they *really* need. In our project there were many possible strategies to store phone numbers efficiently. For example, we could split the number into separate columns for country code, city code, and number. Each strategy had pros and cons.

However, we were able to identify the best and simplest decision after business users explained that they always use one of two methods. They either look up the entire phone number in the format '1 234 5678910', without any leading zeros and without any non-numeric characters, or they search a set of numbers with leading digits, like all phone numbers starting with '123456%'.

Note: If end users want to be able to choose either format ('001 234 5678910' or '+1 234 5678910'), you can leave it up to the application to extend the display of the number with the leading '00' or '+'.

After we agreed on the phone number format, we performed tests to compare character-based and numeric formats. We chose to test varchar for the character-based storage and bigint for the numeric. We could also have chosen to use the int data type, but in many cases international phone numbers exceed the limits of digits that an int can store. Note that prior to compression considerations, a bigint uses 8 bytes, and a varchar uses as many bytes as there are digits. For this company, the number of digits in a call data record (CDR) usually averages 70-80% of the data in a row. Row sizes are generally between 200,000 and 600,000 bytes.

Test preparation

Fact and dimension table

The customer scenario uses CDRs, which is the representation of the information that is stored by telecom hardware for every call made. A CDR contains call details such as duration and number dialed.

The CDR information ususally looks like this.

RecordT	IMEI	IMSI	MSISDN	ChargingDateTime	Chargeab
0	35504600727335	250997401539330	79032649242	2010-03-29 05:02:44.000	1641
2	35451401937889	250991201206706	79099442522	2010-03-29 05:05:19.000	1740

In the data warehouse, CDRs can be stored in a star schema; a fact table can store numbers, durations, and so on, and dimesion tables can store attributes of the phone, subscriber, and so on.For testing we used the following schema: two fact tables for comparison: one with a bigint column (FactInt) and another with a varchar column representing the phone number (FactChar), as well as a few dimention tables.

1. Fact table with bigint column:

```
CREATE TABLE [dbo].[FactInt](

      [RecordTypeOrPartial] [tinyint] NULL,

[IMEI] [bigint] NULL,                              -- Phone HW ID

[IMSI] [bigint] NULL,

[ChargingDateTime] [datetime] NOT NULL,

[ChargeableDuration] [smallint] NULL,
```

```
[CellID] [varbinary](2) NULL,

… some other columns

[OriginalCalledNumber] [bigint] NULL,           -- dialed number

[OriginalCalledNumberType] [varbinary](20) NULL,

[SSRequest] [tinyint] NULL,

[MSISDN] [bigint] NOT NULL,                     -- subscriber phone number

[MSCID] [bigint] NULL,

[FileID] [int] NOT NULL)GO
```

2. Fact table with varchar column:

```
CREATE TABLE [dbo].[FactChar](

[RecordTypeOrPartial] [tinyint] NULL,

[IMEI] [varchar](16) NULL,

[IMSI] [varchar](16) NULL,

[ChargingDateTime] [datetime] NOT NULL,

[ChargeableDuration] [smallint] NULL,

[CellID] [varbinary](2) NULL,

…some other columns [OriginalCalledNumber] [varchar](40) NULL,  -- dialed
number

[OriginalCalledNumberType] [varbinary](20) NULL,

[SSRequest] [tinyint] NULL,

      [MSISDN] [varchar](20) NOT NULL,

[MSCID] [varchar](20) ] NULL,

[FileID] [int] NOT NULL)
```

Note: We could have used varchar (20) to store phone numbers [OriginalCalledNumber], which would in our comparison correspond to the number of digits bigint can hold. We used varchar (40) because this is what the customer actually used, and in this paper I prefer to be closer to a real case sceanrio. 3. Dimension table with attributes on where the subscriber number is registered (phone number is stored as varchar (20)):

```
CREATE TABLE [dbo].[DimDefinitionAChar](

[MSISDN] [varchar](20) NOT NULL,

[ARegionID] [varchar](20) NULL,

[AFilialID] [varchar](20) NULL,

[ACountryID] [varchar](20) NULL,

CONSTRAINT [PK_MSISDN_Char] PRIMARY KEY CLUSTERED

(

      [MSISDN] ASC

)
```

The tables FactChar and FactInt were partitioned with six partitions for each 10,000,000 rows. The generated dimension tables have 1,000 to 10,000 rows.

Queries

After we built the tables, we defined a set of queries to run. We identified three queries, representing the most frequent or the most long-running queries in the customer's workload, to see whether the data type choice influenced performance. The first query looked for a single phone number within a specified time frame, the second looked for a range of numbers, and the third used a LIKE operator to find the number or numbers.

We used the LIKE operator to look up phone numbers with some leading digits, where the phone number is stored as a bigint. For example, users may need to check how many prepaid phones, sold by a specified shop, made calls (and therefore were activated). Those phones may have sequential numbers where only the last digit or two differ. The LIKE operator determines character string matches; however, if any one of the arguments is not of character string data type, the SQL Server Database Engine converts it to the character string data type, if possible.

Here are the queries:

1. Query type 1: Single phone number lookup in a specific time frame:

```
SELECT [IMEI]

,[ChargingDateTime]      -- time the call was made
```

```
,[ChargeableDuration]      -- call duration

,[OriginalCalledNumber]  -- dialed number

,a.[MSISDN]                -- subscriber phone number

,a.[AFilialID]

,a.[ACountryID]

,m.[MSCID]

,m.[MSCRegionID]

,m.[MCSCountryID]

,[FileID]

FROM .[dbo].[FactInt]

INNER JOIN DimDefinitionA as a ON a.[MSISDN]= FactInt.[MSISDN]

INNER JOIN DimTableMSC as m ON m.[MSCID]= FactInt.[MSCID]

WHERE [ChargingDateTime] >'2010-01-01 00:00' and [ChargingDateTime]<  '2010-
01-02 00:00'

AND FactInt.MSISDN = 1234567899995324  -- subscriber phone number
```

2. Query type 2: Big report query where the range of phone numbers is looked up:

```
SELECT [IMEI] ,[ChargingDateTime] ,[ChargeableDuration]
,[OriginalCalledNumber] ,Fact.[MSISDN] ,a.[AFilialID]

,a.[ACountryID]

            ,Fact.[MSCID] ,m.[MSCRegionID] ,m.[MCSCountryID]

,[FileID]

FROM [dbo].[FactInt]

INNER JOIN DimDefinitionA as A ON A.[MSISDN]= FactInt.[MSISDN]

INNER JOIN DimDefinitionPosition as P ON P.[CellID]= FactInt.[CellID]
```

```
INNER JOIN DimTableMSC as M ON M.[MSCID]= FactInt.[MSCID]WHERE
[ChargingDateTime] >'2010-01-01 00:00'and [ChargingDateTime]<  '2010-01-02
00:00'AND FactInt.MSISDN > 1234567899995324 and FactInt.MSISDN <
12345678999995924AND FactInt.[MSCID] > 790300000705 and FactInt.[MSCID] <
790300000800
```

3. Query type 3: A lookup in which the LIKE operator is used:

```
SELECT [IMEI] ,[ChargingDateTime] ,[ChargeableDuration]

,[OriginalCalledNumber]

,FactInt.[MSISDN]

,a.[AFilialID]

,a.[ACountryID]

,FactInt.[MSCID]

,m.[MSCRegionID]

,m.[MCSCountryID]

,[FileID]

FROM [dbo].[FactInt]

INNER JOIN DimDefinitionA as A ON A.[MSISDN]= FactInt.[MSISDN] INNER JOIN
DimDefinitionPosition as P ON P.[CellID]= FactInt.[CellID]

INNER JOIN DimTableMSC as M ON M.[MSCID]= FactInt.[MSCID]

WHERE [ChargingDateTime] >'2010-01-01 00:00' and [ChargingDateTime]<  '2010-
01-02 00:00'

AND FactInt.MSISDN Like '12345678999953%'

AND FactInt.[MSCID] Like '7903000007%'
```

Because we needed to know the real query performance at the beginning of each query execution, we compared queries with and without cache cleanup.D

```
DBCC FREESYSTEMCACHE ('ALL');

DBCC DROPCLEANBUFFERS;
```

```
DBCC FREEPROCCACHE Go
```

Test

After all preparation was finished, we performed the following tests for each type of query:

1) **Test 1.** Query fact tables and dimension table, compression not enabled , cache is cleaned up (cold)

2) **Test 2.** Query fact tables and dimension table, compression not enabled, cache is warm

3) **Test 3.** Query fact tables and dimension table, fact table compressed, cache is cold

4) **Test 4.** Query fact tables and dimension table, fact table compressed, cache is warm Here are the numbers the tests runs generated.

Example of results from Tests 1 and 2

Cold Cache		Warm Cache	
Query 1 (FactChar, ms)	Query1 (FactInt, ms)	Query 1 (FactChar, ms)	Query1 (FactInt, ms)
17631	6737	1144	418
21684	6548	1142	364
22179	6476	1158	351
17136	6646	1296	347
18198	6503	1135	383

Returned: 994 rows

Cold Cache		Warm Cache	
Query 2 (FactChar, ms)	Query 2 (FactInt, ms)	Query 2 (FactInt, ms)	Query 2 (FactInt, ms)
86341	26510	6754	1343
86086	26394	5697	1431
85866	26560	5802	1456
84778	26676	5514	1551

85595	26279	5638	1259

Returned: 56179 rows

Cold Cache		Warm Cache	
Query 3 (FactChar, ms)	Query 3 (FactInt, ms)	Query 3 (FactChar, ms)	Query 2 (FactInt, ms)
98691	27850	1177	1063
98692	26984	1124	1087
99105	27127	1640	1017
98474	26989	1300	1011

Returned: 9991 rows

You can see that:

- Queries against the bigint table with a cold cache ran between about 54% and about 67% faster than the queries against tables where data was stored as a character data type.- These queries also showed significant performance gain for tables with bigint in comparison to varchar data type when run against a warm cache. Queries against the FactInt table ran about 24% to about 70 % faster than queries against the FactChar table.

Next, we enabled compression on the tables to see its effect on query performance.

Example of results from Tests 3 and 4

Cold Cache		Warm Cache	
Query 1 (FactChar, ms)	Query 1 (FactInt, ms)	Query 1 (FactChar, ms)	Query 1 (FactInt, ms)
13538	5453	1121	423
13206	5164	1171	420
13153	5155	1185	470
13568	5598	1233	487
13561	5668	1179	397

Cold Cache		Warm Cache	
Query 2 (FactChar, ms)	Query 2 (FactInt, ms)	Query 2 (FactChar, ms)	Query 2 (FactInt, ms)
20058	13195	2232	1548
20318	13327	2228	1766
19802	13546	1967	1657

19835	13276	1904	1714
19789	13326	2081	1617

Cold Cache		Warm Cache	
Query 3 (FactChar, ms)	Query 3 (FactInt, ms)	Query 3 (FactChar, ms)	Query 3 (FactInt, ms)
24922	13924	1271	1138
33755	13692	1287	1262
32091	13883	1448	1071
31190	13879	1286	1137
31083	13659	1389	1138

The results indicated that after compression was enabled on the tables, performance of the queries on the tables FactInt and FactChar was almost identical.

Compression did improve the performance for both queries, compared to the same test against the noncompressed data. What was interesting was that average execution time dropped slightly. However, the queries still took a relatively long time to execute. The only real difference appeared when the cache was cold: queries against the FactInt table ran about 32% to 58% faster than queries against the FactChar table.

Finally, we observed a significant space savings when bigint was used. The following table compares the amount of space you need to store data using both a character type data format and a numeric data format.

Name	**Rows**	**Compressed?**	**Data**	**index_size**
FactChar	60000000	no	9634,281 MB	2192,016 MB
FactChar	60000000	yes	4735,781 MB	2175,133 MB
FactInt	60000000	no	7324,219 MB	1597,125 MB
FactInt	60000000	yes	4077,320 MB	1585,586 MB

Conclusion

When you design complex database solutions, you may be tempted to choose character-based data types and save development time and efforts. However, this may not be the best choice for your

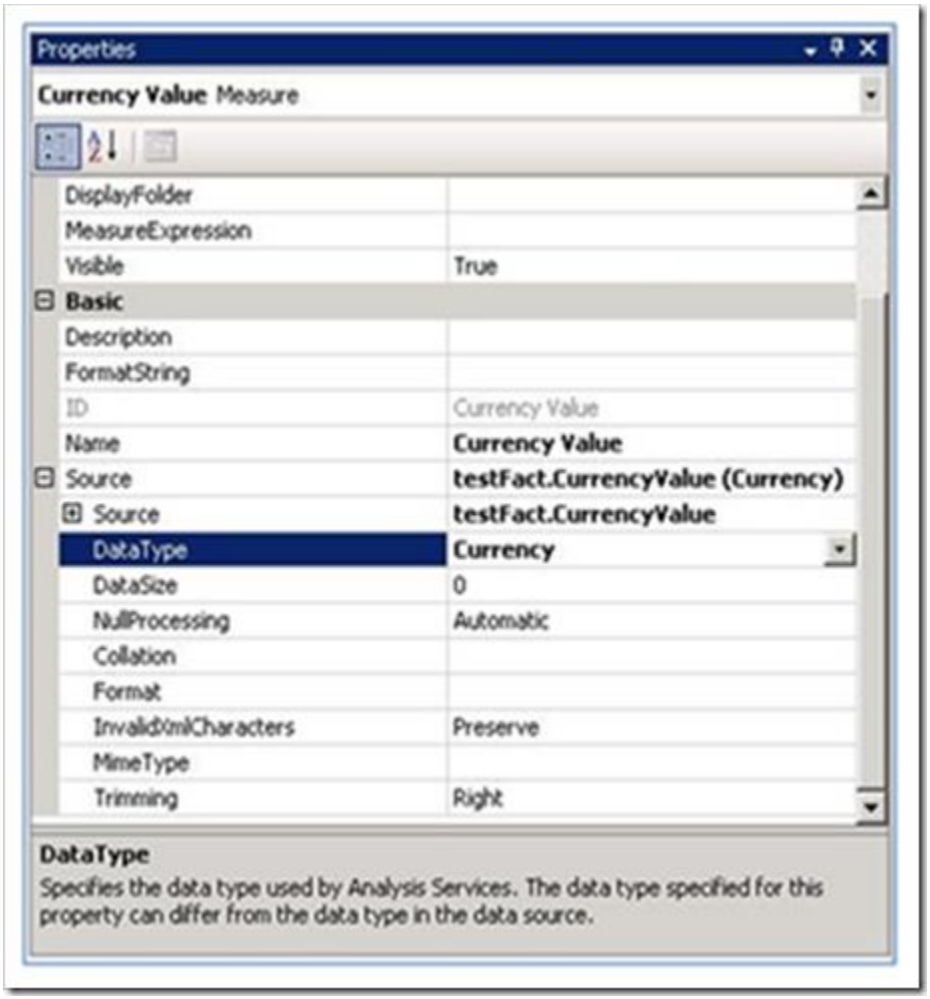
scenario. Talking to the business may help you identify the best data type and format for your customer, based on the way the data is queried and the effects of different settings on performance. As our tests indicate, apparent simplicity may cost you query performance and storage space. Choosing numeric data types over character-based data types may offer performance gains, which could improve overall system throughput.

The Many Benefits of Money…Data Type!

Background

Our initial reason for looking at the **money** data type can be found within the Precision Considerations for Analysis Services Users white paper. In this white paper, we provide extensive examples of the types of precision issues when your SQL relational data source and your Microsoft® SQL Server® Analysis Services cube have different non-matching data types (e.g., if you query one way you get the value 304253.3251, but run the query in another way and you get the value 304253.325100001).

To avoid these types of problems, you need to ensure that your SQL relational data source and Analysis Services measure groups have matching data types. By default, when you create an Analysis Services measure on a **money** data type, Microsoft Visual Studio® Business Intelligence Development Studio will set the data type reference to **double**. To avoid precision loss and have faster performance, you should change the data type to **currency** within the Source Properties as noted in the screen shot below.



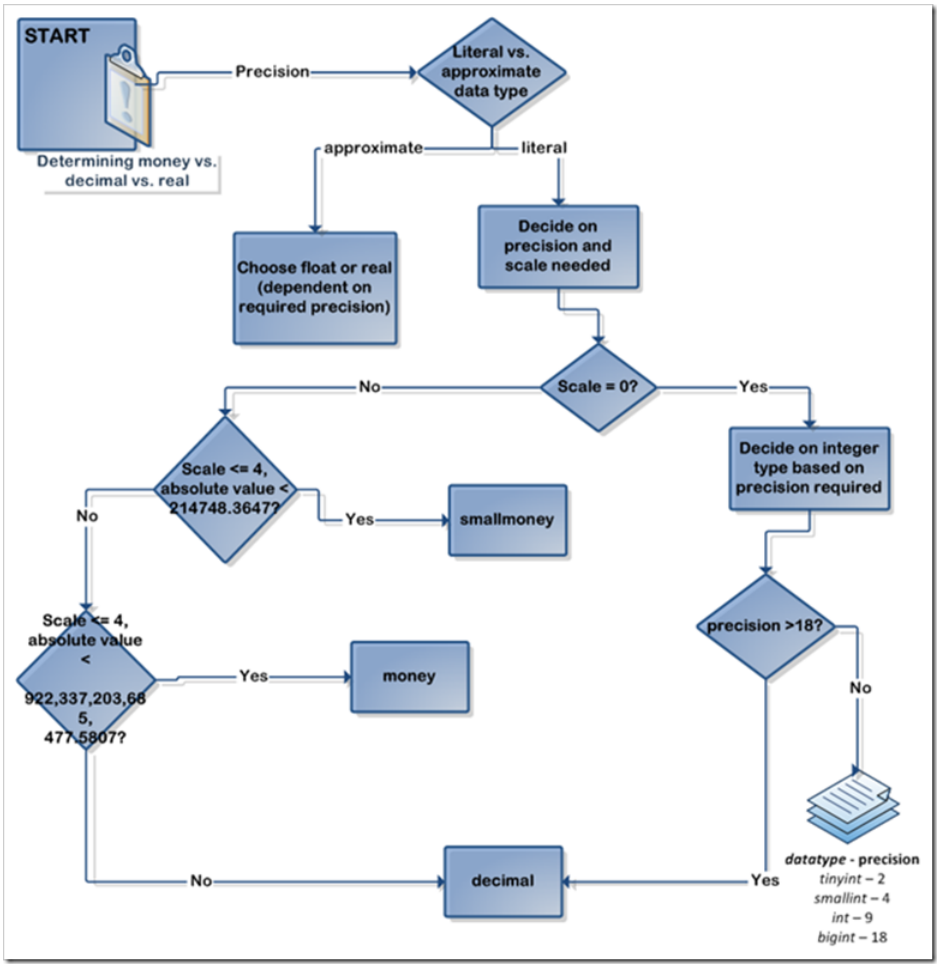
Show Me The Money! …Data Type for Faster Processing Performance

Working on customer implementations, we found some interesting performance numbers concerning the **money** data type. For example, when Analysis Services was set to the **currency** data type (from **double**) to match the SQL Server **money** data type, there was a 13% improvement in processing speed (rows/sec). To get faster performance within SQL Server Integration Services (SSIS) to load 1.18 TB in under thirty minutes, as noted in SSIS 2008 - world record ETL performance, it was observed that changing the four **decimal(9,2)** columns with a size of 5 bytes in the TPC-H LINEITEM table to **money** (8 bytes) improved bulk inserting speed by 20%. Note that within SSIS, the equivalent of the **money** data type is DT_CY, which currently does not support fast parse. Hence, getting **money** out of text files may incur additional cost.

*Note that these tests were performed on 64-bit systems. Relative performance may be different in the 32-bit editions of SQL Server because of differences in the way it performs 64-bit **integer** (or **money**) operations.*

Money vs. Decimal vs. Float Decision Flowchart

Below is a high-level decision flowchart to help you decide which data type you should use. Note that this is a generalization that may not be applicable to all situations. For a more in-depth understanding, you can always refer to Donald Knuth's The Art of Computer Programming – Volume 1.



As well, remember that different data types have different client API mappings. Some more in-depth references to this include SQL Server Data Types and ADO.NET and A Money type for the CLR.

Money (Data Type) Internals

The reason for the performance improvement is because of SQL Server's Tabular Data Stream (TDS) protocol, which has the key design principle to transfer data in compact binary form and as close as possible to the internal storage format of SQL Server. Empirically, this was observed during the SSIS 2008 - world record ETL performance test using Kernrate; the protocol dropped significantly when the data type was switched to money from decimal. This makes the transfer of data as efficient as possible. A complex data type needs additional parsing and CPU cycles to handle than a fixed-width type.

Let's compare the different data types that are typically used with money (data types).

Breakdown	money	decimal	float
Simple/complex data type: Simple data types align more directly to native processor types. Complex data types require CPU to review type metadata and to perform branching.	Simple	Complex	Simple
Fixed/variable length writers: Because a variable-length data type may incur a memcpy when moving, causing additional CPU overhead, use a fixed 8-byte or 4-byte integer assignment if possible.	Fixed	Variable	Fixed
Storage format: Incurs less overhead if the data type is composed of native literals (e.g., **int**, **uint**, **long**, **ulong**) instead of approximate data types (e.g., **float**).	8-byte integer	Scaled integer (one sign byte plus one to four **ulong** depending on precision)	8-byte integer
Comments: This row lists other issues of concern.	None	TDS wire format is always packed, so extra overhead is required to pack and unpack this data type.	Approximate data are types more expensive to compare/convert than native literals; there may be precision issues on conversion.

The key here is that the **money** data type is a simple fixed-length integer-based value type with a fixed decimal point. Composed of an 8-byte signed integer (note that **small money** is a single 4-byte integer) with the 4-byte CPU alignment, it is more efficient to process than its **decimal** and **floating point** counterparts. The other side of the coin is that floating points (but not **decimal**) can be more quickly calculated in the floating point unit of your CPU than **money**. However, bear in mind the precision issues of **float** as noted above.

Saving (*Space for*) Your Money!

In the context of SQL Server data compression, the **money** and **small money** data types tend to compress well when the absolute value is low (e.g., values between -0.0128 and 0.0127 compress to 1 byte, while values between -3.2768 and 3.2767 compress to 2 bytes). It is the absolute value that matters for compression, not the number of significant digits; both 1,000,000 and 1,234,567.8901 will take 5 bytes compressed. On the other hand, **decimal** will compress better when there are fewer significant digits. For example, both 1,000,000 and .0001 will compress to 2 bytes, but 1,234,567.8901 will take several more bytes because it has more significant digits.

$ummary

There will be many scenarios where you preferred option will still be to use data types such as **decimal** and **float**. But before skipping over this detail, take a look at your data and see if you can change your schema to the **money** data type. After all, a 13% improvement in Analysis Services processing speed and 20% improvement in SSIS processing isn't chump change.

…and that's our $0.02.

How many files should a database have? - Part 1: OLAP workloads

The subject - how many files a database should have - is a question that comes up often. The answer is of course: it depends. But, what does it depend on?

Background information

If a filegroup in SQL Server contains more than one file, SQL Server will “stripe” allocations across the files by using a proportional fill algorithm. If the files in the group have the same size (which we recommend), the allocation is essentially a round-robin. The “stripe size” of this round-robin is by default one extent - 64KB. Hence, the first allocated extent goes to the first file in a filegroup, the second extent to the second and so on. This striping mechanism can be quite useful, because you can spread your I/O over several LUNs by allocating a data file to each. You should strive to have all files in a filegroup be of equal size. Using Files and Filegroups on TechNet rovides more background on filegroup and allocation.

Each file in the database has its own PFS, GAM and SGAM pages. These special “administration pages” track the free space and allocation in the in the file. Every new allocation in the file will have to access a PFS page and in some cases also the GAM/SGAM pages. (For more background on this see "Inside SQL Server 2005: The Storage Engine” by Kalen Delaney).

In this Tech Note, we look at files from an OLAP / Data Warehouse workload perspective. Because OLAP and OLTP workloads differ greatly, different recommendation for file allocation apply.

Too few files in a filegroup

If a filegroup receives a lot of insert activity, the pressure on the PFS and GAM/SGAM page access becomes significant. At some point, this becomes a bottleneck, effectively slowing down the insert throughput. In a data warehouse load, the contention is typically on PFS pages.

If PFS contention is present in a workload, it will show up as waiting for **PAGELATCH_UP** in **sys.dm_os_wait_stats**. You can use **sys.dm_os_waiting_tasks** to see which pages you are waiting for. You will see something like the following:

	session_id	exec_context_id	wait_type	resource_description
1	85	0	PAGELATCH_UP	6:1:2895504
2	122	0	PAGELATCH_UP	6:1:2895504
3	99	0	PAGELATCH_UP	6:1:2895504

The format of the **resource_description** column is: DBID:FILEID:PAGEID. You can use the **resource_description** to look up the page in **sys.dm_os_buffer_descriptors** to see if your wait is on a PFS page.

If you discover that you have many waits for PFS pages, you probably need to add more files to the affected filegroup. Because each file has its own administration pages – the presence of more files reduce contention on PFS pages.

Too many files in a filegroup

Increasing the number of files is useful if you use the files to "stripe" across LUNs or if you run into the PFS bottleneck as described above.

However, there is a disadvantage in having ***too many*** files. Remember that SQL Server will stripe the extents over the files in stripe sizes of 64KB. Assume that you have great deal of insert activity on a single filegroup that contains many files. Since SQL Server distributes the extents across the files, your average I/O request will typically have a size of 64KB. If you instead had fewer files in the filegroup, SQL Server could "bundle" the allocations and hence, drive larger block sizes. Most I/O systems deliver a better throughput if you can drive large block sizes.

Testing with TPC-H LINEITEM data shows the following pattern when loading the database using minimally logged operations:

# Files in filegroup	Avg I/O block size (KB)
1	256
2	196
4	132
8	64
16	64
32	64

The above numbers are for a single bulk stream to the file.

SQL Server is quite good at bundling I/O operations together in large blocks – a process known as scatter/gather. In our test, we tried to increase the concurrency of the bulk load to utilize this functionality. With 64 bulk streams, we were able to drive block sizes up to 196 KB, even with 32 files in a single filegroup. But still, with 1 file in each of multiple filegroups, we were getting a faster, 256 KB block sized I/O.

So, while adding more files can benefit performance by eliminating PFS contention, it can make the I/O pattern less efficient. You can measure the size of the block requests using the perfmon counter: **Logical Disk: Avg Disk Bytes / Write** to gauge how efficient your block size is.

Having many *filegroups* in a database adds an administrative burden: you now have to balance the space usage in the database between the filegroups. Therefore, you probably do not want to go overboard in optimizing your I/O block sizes by adding filegroups with few files and allocating your tables across them. In our TPC-H load test we only saw a 5% increase in disk throughput from optimizing block sizes and the benefit was only realized above several hundred MB / Sec insert speeds.

Another factor to consider, from an administrative perspective, is database startup recovery. File recovery after server restart or after a **SET ONLINE** operation on the database are done sequentially. If you have many files (hundreds) in your database, this recovery process can take a long time because each file is opened sequentially.

The –E startup flag

The SQL Server startup flag –E forces SQL Server to allocate 4 extents at a time to each file, essentially quadrupling the stripe size. In heavy insert scenarios, this drives larger block sizes to the disk. Also, your pages allocation will be more sequential with the same data file, allowing better sequential I/O for range and table scan operations (which are common in OLAP workloads).

This startup flag provides most, but not all, of the above mentioned benefits to I/O system – without the overhead of managing multiple filegroups. Be aware that this flag is supported only in 64-bit environments. You can find information about the –E startup flag in File allocation extension in SQL Server 2000 (64-bit) and SQL Server 2005 (KB329526).

So, how many database files should I have in my OLAP system?

To allocate the optimal number of files you must understand your database workload. The amount of insert activity is the determining factor. You also must balance the following factors:

- PFS contention
- need for SQL based striping
- I/O pattern – block sizes
- File recovery times

PFS contention and SQL-based striping drives you towards allocating more files. Optimizing I/O requests and file recovery leans towards fewer files. If your work load is very insert heavily, you generally want more files, but in a controlled manner. We have seen benefits of having up to half the amount of files as you have cores – and even more in the case of t**empdb**. If your data load is more read intensive, having fewer files may benefit you, since PFS contention is not a problem in this case and your I/O will arrive in larger bundles.

In extreme cases, when inserting hundreds of MB / Sec, you can benefit from partitioning your table into filegroups, each with low number of files to bundle I/O requests together and create

larger block sizes with sequential disk access. But, by doing this you assign performance priority over ease of administration.

So the answer to the question, “How many files should my data warehouse have?” is: as few as possible, without running into PFS contention and without sacrificing striping ability.

SQL Server Partition Management Tool

SQL Server Partition Management Tool is available at http://www.codeplex.com/SQLPartitionMgmt with source code.

This tool provides a set of commands (at the Command Line or via Powershell) to create a staging table on-demand (including all appropriate indexes and constraints) based on a specific partitioned table and particular partition of interest. By calling this executable, with parameters, from maintenance scripts or SSIS packages, DBAs can avoid having to 'hard code' table and index definition scripts for staging tables. The tool eliminates the challenge of keeping such scripts in synch with changes to partition tables' columns or indexes. It also provides a fast, single-command shortcut for the operation of quickly deleting all data from a partition.

This tool supports SQL Server 2008, but is also fully compatible with SQL Server 2005. An earlier verison of the tool (for SQL Server 2005) is also available on codeplex under the same project.

The latest version supports new features in SQL Server 2008 such as filtered indexes, new data types, and partition-aligned indexed views.

Section 3: Fast-track

Lessons Learned and Findings from a Large Fast-Track POC

Executive Summary

To aid Contoso's goal in establishing an enterprise data warehouse (EDW) with the objective of creating a single version of the truth, Microsoft has participated in a proof of concept (POC) to demonstrate the performance, scalability, and value of the Microsoft SQL Server application platform. The Microsoft team used our Fast Track Data Warehouse solution based on HP hardware and SQL Server 2008 R2 RDBMS. This involved conversion work from the current Oracle production system to the test SQL Server system. In summary, we built a Fast Track configuration that scaled according to the demands put on it by the requirements. We were able to deliver excellent performance numbers for both pre and post optimization serial runs. Additionally the performance numbers for volumized data scaled according to the growth in size of data. We did additional testing on different approaches for loading the data, that is, BULK INSERT vs. the **bcp** utility. We also loaded data into databases with tables using nonclustered and clustered primary keys and were able to show the workload performance numbers for both. The compression scenario delivered impressive compression ratios and indicated areas where it could be used for query performance optimization. The calculated compression ratio before compression was a near estimate of the actual table compression size. The customer's data compressed better than the estimate predicted. The backup scenario proved that the speed and backup compression will be key wins for manageability of the data warehouse. When testing updates in place, we optimized the performance numbers by setting the fill factor appropriately to avoid page splits. Early feedback indicated that this was much faster than Oracle and other competing vendors, and an order of magnitude faster than the current production system. Finally we were able to conclude that the performance of the Fast Track configuration was better than current production by many orders of magnitude across all the testing options. Given the opportunity to deliver the enterprise data warehouse, the Microsoft Fast Track Data Warehouse can deliver workload performance far surpassing expectations with the latest G7 hardware. We believe we can deliver the best performance through our expertise in optimization and migrating code from different database platforms.

Introduction

Contoso, one of the biggest banks in Malaysia, called for a POC among Oracle (Teradata), IBM (DB2 on zSeries), Greenplum, and Microsoft. They developed a very elaborate and complete POC scope that can be broken into five tracks: 1. Track A is a simulation of the current environment. To create Track A, load two months of information from an Oracle database. Migrate objects and data to SQL Server but do not do any optimizations. That way, get a baseline of a direct migration by running serially specific queries and stored procedures (six fixed queries, four stored procedures, and 20 or so ad hoc queries)2. Track B. Apply optimizations to database and queries to Track A. Run the six queries and four stored procedures in parallel while updating the customer master table. Also, switch in a new partition and ensure that there are no issues with currently running queries or dirty data.3. Track C. Start with Track B, and then multiply all the dimension tables by three and the fact details table to seven years (that is, an additional 82 months). Record load times. 4. Track D. Modify queries from Track B to get the new bigger data set.

Hardware Configuration

Configuration: HP DL785 G5 with 8 socket quad core: 32 AMD Opteron 8376 HE, 2300 MHz and 256 GB ram. (NOTE: Even though this system is not in the list of the published HP Fast Track configurations, it followed the rules of fast track to achieve a well-balanced system between cores and LUNs.)I/O: Five HP Smart Array P411 controllers and 10 D2700 enclosures direct attached (DAS). Each enclosure had 25 disks x 146 GB each 10k rpm SAS 6G dual port. We used four controllers for FT with 32 RAID 1 LUNs of 2 disks each. The fifth controller, which was used for backup and other storage, had three RAID 5 LUNs of 8 disks each. Operating system: Windows Server 2008 R2 Enterprise with hotfixes KB 2155311, 977977, 976700, and 982383. These are all I/O related fixes for Windows Server2008 R2 and multi core systems.DBMS: SQL Server 2008 R2 Enterprise plus Cumulative Update package 3. We used –E and -T1118 trace flags. Trace flag 1118 forces uniform extent allocations instead of mixed page allocations. It is commonly used to assist in **tempdb** scalability by avoiding SGAM and other allocation contention points. We allocated a maximum of 250 GB RAM to SQL Server. **Database configuration** we used one filegroup for the Fast Track database with 32 data files of equal size, one in each LUN because we had 32 cores. We used the same configuration for **tempdb**. For the transaction log, we had four extra RAID 1 LUNS of two disks, one in each enclosure.

Track A – Nonclustered Indexes as Primary Keys

Notes

- This test ensured adherence to the loading of data in waves. The PK indexes are nonclustered, in keeping with Contoso's wishes.• Primary key duplicates are ignored by the process and not inserted into the tables. There are other bad-data errors. These are logged in the error log files.• The data errors were left in the load files and stripped on load to show the worst-case scenario.Here is an example of the command we used to load data in from the command batch.

```
bcp TrackA.dbo.[<Table>] in <Table>.txt -c -F2  -r \n -m 999999 -t"|"  -b 100000  -U %USERNAME%  -S %SERRVERNAME% -P %PASSWORD% -e <Table>.txt_Errors.log -h "TABLOCK"
```

Loaded Row Count

Table Name	Actual Loaded Row Count
ACCT_HT	5,592,356
ACCT_XREF	22,898,855
CARD_CREDIT_LINE_HT	1,217,097
CERT_DEPOSIT_HT	1,146,640
CODE_HT	39,796
COLLAT_ACCT_REL_HT	2,744,101
COLLAT_CONS_HT	71
COLLAT_CUST_REL_HT	3,955,228
COLLAT_FIN_HT	847,391
COLLAT_GUAR_HT	491,094
COLLAT_MACH_HT	7,059

COLLAT_MVEH_HT	1,246,607
COLLAT_OASST_HT	3,660
COLLAT_OVEH_HT	174
COLLAT_PROP_HT	205,294
COLLATERAL_HT	2,789,530
COMMERCIAL_LOAN_HT	225,983
CONSUMER_INST_LN_HT	1,254,016
CREDIT_PROVISIONS_HT	1,645,043
CUST_ACCT_REL_HT	7,642,494
CUST_CUST_REL_HT	1,073,973
CUST_HT	7,165,192
CUST_NON_PERS_HT	392,467
CUST_PERS_HT	6,773,268
CUST_XREF	12,311,195
PRD_DIMENSION	4,017
PRODUCT	1,497
RETAIL_CHECKING_HT	165,041
RETAIL_CREDIT_LN_HT	8,048
RETAIL_SAVING_HT	1,583,574
TRAN_DETAIL	19,107,244

Comments

The total load time was 6,393 seconds. This was due to the fact that all the indexes were kept and the data was not pre-cleaned to remove corrupt rows and duplicate primary key data.

Track A - Clustered Indexes as Primary Keys – FAST LOAD

Notes

- The primary key indexes were changed to clustered indexes, and primary key data and bad data were removed from the files to ensure clean fast loads.• The nonclustered indexes were built on the data after it was loaded in a parallel batch.Here is an example of the command we used to load data in from the command batch.`bcp TrackA.dbo.[<Table>] in <Table>.txt -c -F2 -r \n -m 999999 -t"|" -b 100000 -U %USERNAME% -S %SERRVERNAME% -P %PASSWORD% -e <Table>.txt_Errors.log -h "TABLOCK"`

Comments

The total load time (to build the entire database) was 2,485 seconds. This time is made up of 2,166 seconds for the data load and 319 seconds for the creation of all the nonclustered indexes. This was much faster than the 6,393 seconds it took to load the data using nonclustered indexes as primary keys. The 2,166 seconds can be further reduced if the declarative referential integrity (DRI) constraints are removed. This was not tested because Contoso had a requirement to keep the DRI constraints in the database.

Track A - Best Practices

- Normally you should not have constraints on the DW tables. Foreign keys and logical integrity of the data to be loaded should be handled at the ETL layer to minimize data load performance issues. In

our case, we could not avoid the constraints due to the POC requirements. - SQL Server Integration Services can provide a better means of cleaning the data and taking care of slowly changing dimensions. - You need to determine what you want to focus on: load performance for historical data or query performance. Sometimes you cannot optimize for both at the same time. In our case, we optimized for query performance.- Consider loading data into partitioned tables for incremental loads.- Use the BCP utility to load data in and specify the TABLOCK and ORDER hints. Ensure that the order of the data is the same as it is in the clustered index.

Query and Stored Procedure - Execution Summary

Results Before and After Optimization

This table shows the numbers before and after optimization for the execution of procedures and queries. **All executions were performed serially.** That means the following:

1. **Track A** database was used for the before-optimization number, that is, where nonclustered indexes used for primary keys. The queries and stored procedures were migrated from Oracle and not optimized.

2. **Track B** database was used for the after-optimization number. The queries and stored procedures were optimized for the after-optimization numbers.

Item	Execution time in current Oracle production	Execution time (before optimization) Hr:Min:Sec	Execution time (after optimization) Hr:Min:Sec
Q1 – Coreplan	00:16:00	00:02:00	00:00:02
Q2 – FINS_CASHDEP	00:05:00	00:03:02	00:00:02
Q3 – FINS_FDPLACE	00:03:00	00:00:02	00:00:00 (1 ms)
Q4 – MUTIARA_CPS	34:07:00	00:02:14	00:01:18
Q5 – PIDM_APP1	> 48 hours	00:00:04	00:00:02
Q6 – PIDM_APP2	> 48 hours	00:00:17	00:00:14
Stored Proc 1 - CTSM_ME_PROD_PARTITION	00:23:00	00:01:06	00:00:35
Stored Proc 2 - MUTIARA_MAIN_JUL10	02:51:00	00:11:39	00:06:40
Stored Proc 3 - DCS_MERGED_IND_SP	03:20:00	Runs for over 3 hours - terminated	00:03:00 - 00:05:00 (3-5 mins)
Stored Proc 4 - Kpi_Run_On_Age	00:27:00	01:15:41	00:02:00 - 00:04:00

Optimizations Performed – High Level Summary

• Rewrite CASE and WHERE statements as joins.• Clean key join column to remove leading and trailing spaces.• Add nonclustered indexes.• Create statistics.• Build derived columns for substring joins.• Convert Oracle-like function calls from SQL Server Migration Assistant (SSMA) to native function calls, such as:

```
ssma_oracle.to_char_date(sysdatetime(), 'DD-MON-YYYY HH:MI:SS AM')to
CONVERT(varchar(32), sysdatetime(), 109)
```

Results for Seven Years Data Serial Query Execution

Description	Min:Sec
Q1 – Coreplan	00:30
Q2 – FINS_CASHDEP	39:36
Q3 – FINS_FDPLACE	0:02
Q4 – MUTIARA_CPS	10:1
Q5 – PIDM_APP1	0:05
Q6 – PIDM_APP2	10:50

The results seem proportional to the data volume and the fact that the queries were run in parallel. A data set expanded three times by account base and seven years by time will have an impact on the number of rows to be scanned as well.

Description	Hr:Min:Sec
Stored Procedure 1 - CTSM_ME_PROD_PARTITION	0:12:24
Stored Procedure 2 - MUTIARA_MAIN_JUL10	0:09:24
Stored Procedure 3 - DCS_MERGED_IND_SP	1:00:35
Stored Procedure 4 - Kpi_Run_On_Age	0:09:25

The volumization of data seems proportional to the query run time in some instances. Heavy processing via function calls in search conditions and the SELECT clause of the queries within procedures lengthened the run time of procedures. NOTE: The Fins_Cashdep query has a time window of five days on Tran_Detail but the substring function applied on TXD_ACCT_NO column of TRAN_DETAIL will cause the table to be scanned. Generally, we have seen when that when functions such as CONVERT, SUBSTRING, CAST, LTRIM, and RTRIM are used, the plan becomes serial and performance can be affected negatively. For more information about queries and stored procedures execution plans and optimizations, see Appendix B.

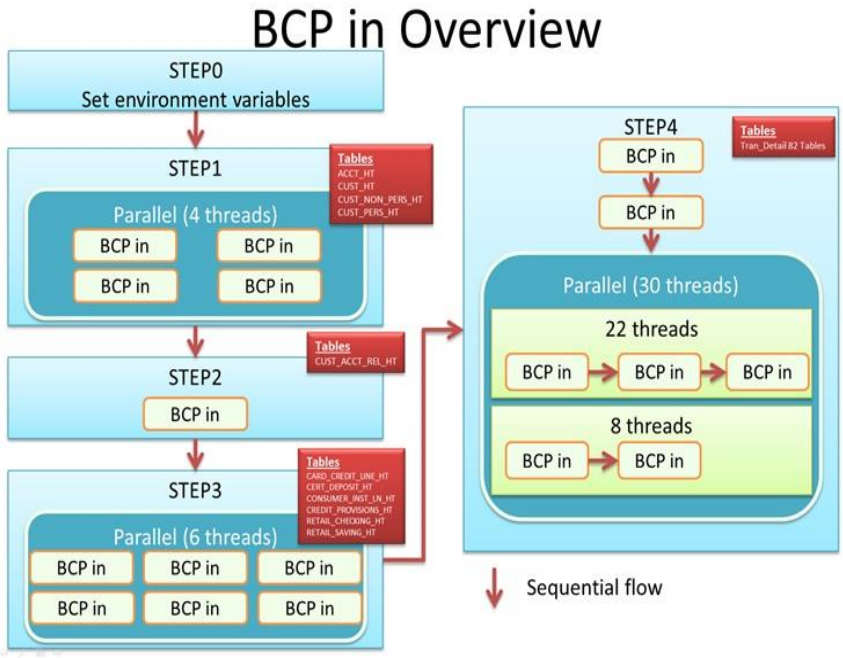
Volume Growth and Scalability

Notes

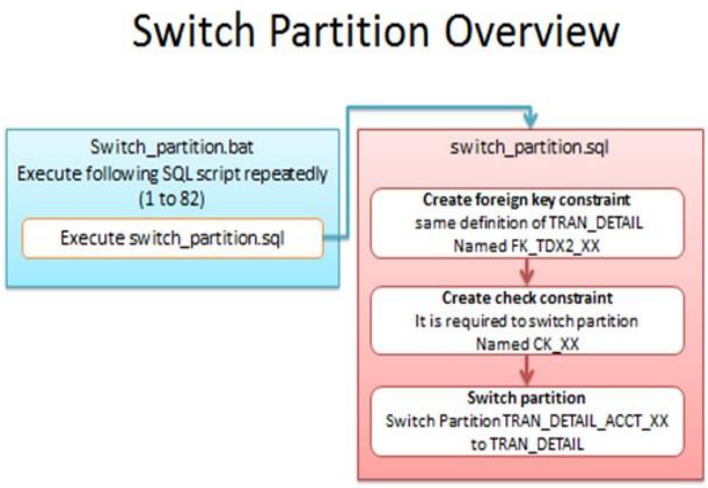
The purpose of Track C is to measure the load time for volumized data. The data volume was increased in keeping with volumization rules set by Contoso.

Load Process

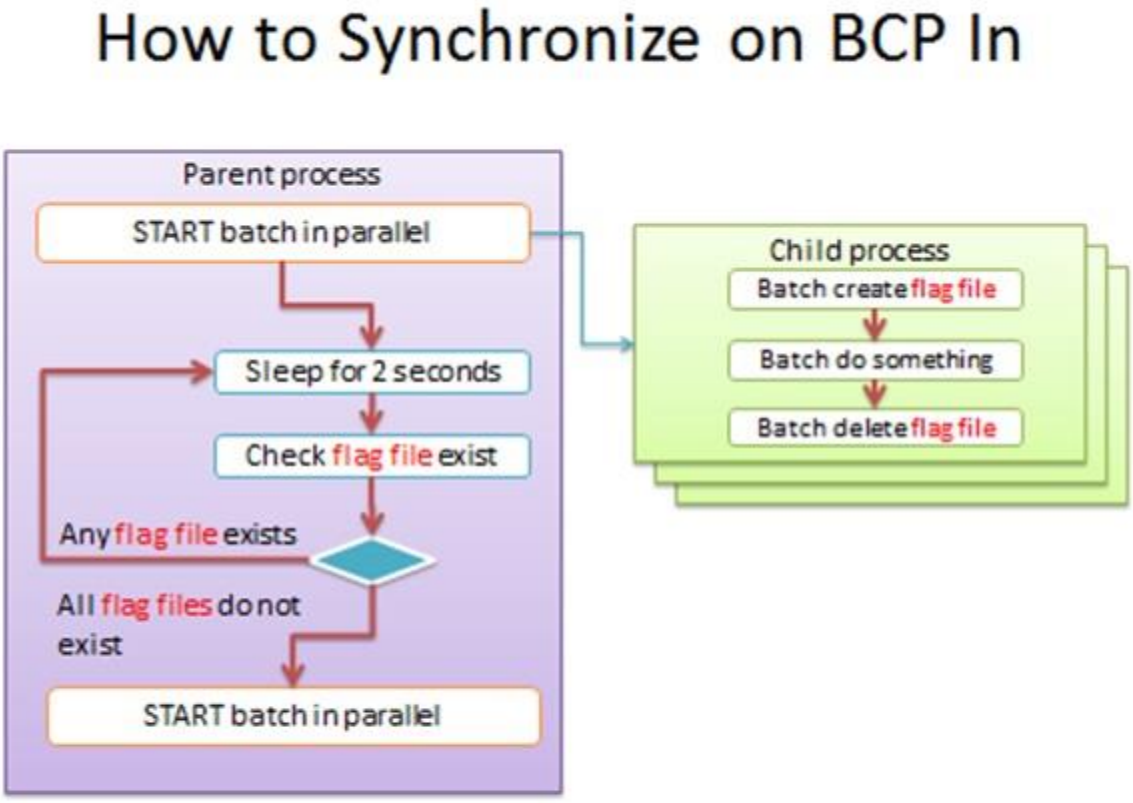
The diagram shows the load sequence of bulk copying of the files using the BCP



utility. After the BCP files are loaded, the data is



switched. The file load is synchronized using a flag-file method.



Results

STEP	Table name	TIME(ms)	TIME(min)
STEP1 Parallel	ACCT_HT.LOG	1,913,961	32
	CUST_HT.LOG	1,701,456	28
	CUST_NON_PERS_HT.LOG	45,365	1
	CUST_PERS_HT.LOG	772,750	13
Subtotal STEP1		1,913,961	32
STEP2	CUST_ACCT_REL_HT	2,807,191	47
Subtotal STEP2		2,807,191	47
STEP3	CARD_CREDIT_LINE_HT	800,581	13

Parallel	CERT_DEPOSIT_HT	248,494	4
	CONSUMER_INST_LN_HT	550,090	9
	CREDIT_PROVISIONS_HT	413,043	7
	RETAIL_CHECKING_HT	-	-
	RETAIL_SAVING_HT	329,147	5
Subtotal STEP3		800,581	13
STEP4-1 Serial	TRAN_DETAIL_ACCT_83	1,923,914	32
	TRAN_DETAIL_ACCT_84	1,446,005	24
Subtotal STEP4-1		3,369,919	56

Tran_Detail Import Performance

We did two passes of the import; the results are shown for both. Pass 1 uses BCP. Here is the command.`bcp %DB_NAME%.dbo.TRAN_DETAIL_ACCT_%NUM% in %3\TRAN_DETAIL_ACCT_%NUM%.txt -c -r \n -t"|" -S %SERVER% -U%USER% -P %PASS% -b 100000 -h "TABLOCK, order(TXD_CYC_DT ASC, TXD_CYC_FREQ ASC, TXD_A_HLD_ORG_CD ASC, TXD_ACCT_ID ASC, TXD_A_FIN_INST_NO ASC, TXD_APPL_SYS_ID ASC, TXD_ACCT_NO ASC, TXD_TRAN_DT ASC, TXD_TRAN_SEQ_NO ASC, TXD_TRAN_CD ASC)" -o %LOG_DIR%\STEP4_TRAN_DETAIL_ACCT_%NUM%.log`- This test loads the data into the table unsorted, and the server has to sort it.- The average import time per table was 1,945 seconds (that is, 32 minutes). The longest tables import was 5,006 seconds (that is, 83 minutes).Pass 2 uses BULK INSERT. Here is the command.`osql -S %SERRVERNAME% -d %DATABASE% -U %USERNAME% -P %PASSWORD% -Q "BULK INSERT %DB_NAME%.dbo.tran_detail_acct_%NUM% FROM '%3\TRAN_DETAIL_ACCT_%NUM%.txt' WITH (TABLOCK ,CODEPAGE = 'RAW',FIELDTERMINATOR ='|',ROWS_PER_BATCH = 100000, ORDER(TXD_CYC_DT ASC, TXD_CYC_FREQ ASC, TXD_A_HLD_ORG_CD ASC, TXD_ACCT_ID ASC, TXD_A_FIN_INST_NO ASC, TXD_APPL_SYS_ID ASC, TXD_ACCT_NO ASC, TXD_TRAN_DT ASC, TXD_TRAN_SEQ_NO ASC, TXD_TRAN_CD ASC))"`- The BULK INSERT time also includes the time it took to build the index. The import tables have indexes defined. The average import time per table was 1,620 seconds (that is, 27 minutes). The longest table's import was 2,591 seconds (that is, 43 minutes).- We used the TABLOCK and –h "ORDER()" parameters in the BCP command for optimization. This decreased the load time almost by half.

Loaded Row Count

Table Name	Loaded	Bulked Up Row Count
ACCT_HT	5,592,356	16,777,068
CARD_CREDIT_LINE_HT	1,217,097	3,405,424
CERT_DEPOSIT_HT	1,146,640	3,220,152
CONSUMER_INST_LN_HT	1,254,016	3,501,549
CREDIT_PROVISIONS_HT	1,645,043	4,495,969
CUST_ACCT_REL_HT	7,642,494	23,276,637
CUST_HT	7,165,192	21,495,576
CUST_NON_PERS_HT	392,467	1,177,401
CUST_PERS_HT	6,773,268	20,319,804
RETAIL_CHECKING_HT	165,041	490,279
RETAIL_SAVING_HT	1,583,574	4,341,972
TRAN_DETAIL	19,107,244	2,300,000,000

Comments

The best approach is to use BULK INSERT with the primary key in place.

Fast Track Backup Performance – Bonus Item

The performance we achieved was **2,107.8 MB per second.**

Command

```
BACKUP DATABASE [TrackC] TO  DISK =
N'C:\MountR5\R5A\Backups\TrackC_F1.bak',  DISK =
N'C:\MountR5\R5B\Backups\TrackC_F2.bak',  DISK =
N'C:\MountR5\R5C\Backups\TrackC_F3.bak' WITH NOFORMAT, NOINIT,  NAME =
N'TrackC-Full Database Backup', SKIP, NOREWIND, NOUNLOAD, COMPRESSION,  STATS
= 10GO
```

Results

We backed up a **1.3-terabyte** database in **10:44 minutes. The size of the backup file was 203 GB.**

Compression of Tables – Extra Item

Compression Summary

A number of tables were compressed using page compression. The tables show the actual and estimated compression ratio.

Track D - Compression of Big Tables					
	Actual Values			Estimated	
	Before Compression	After Compression	Actual Compression Ratio	Before Compression	Estimated Compression
ACCT_HT	18757	7262	**61%**	18757	9446
CUST_HT	17644	9241	**48%**	17644	10790
CUST_ACCT_REL_HT	9762	1928	**80%**	9762	5451
TRAN_DETAIL	2102900	277507	**87%**	2102900	699720

Items in red show actual compression ratio and sizes. This is higher than the estimated compression.

Note: For TRAN_DETAIL its actual size is 85 times the partition size, because there are 85 partitions.

Ad-hoc Query Performance

A set of ad-hoc queries ran in compressed and uncompressed state for the four tables.

Results

Description	CompressedHr:Min:Sec	Uncompressed Hr:Min:Sec
Query1	2:21:4	2:10:54
Query2	2:50:47	2:43:48
Query3	0:17:56	0:11:51
_ERR Fixed Sub query Track D Query4	0:02:02	0:02:18
Query5	0:00:33	0:01:20
Query6	0:02:47	0:02:54
Query7	0:11:54	0:12:41
Query8	0:12:9	2:51:17

CUST_HT, ACCT_HT, and CUST_ACCT_REL_HT were uncompressed, and TRAN_DETAIL was compressed.

Comments

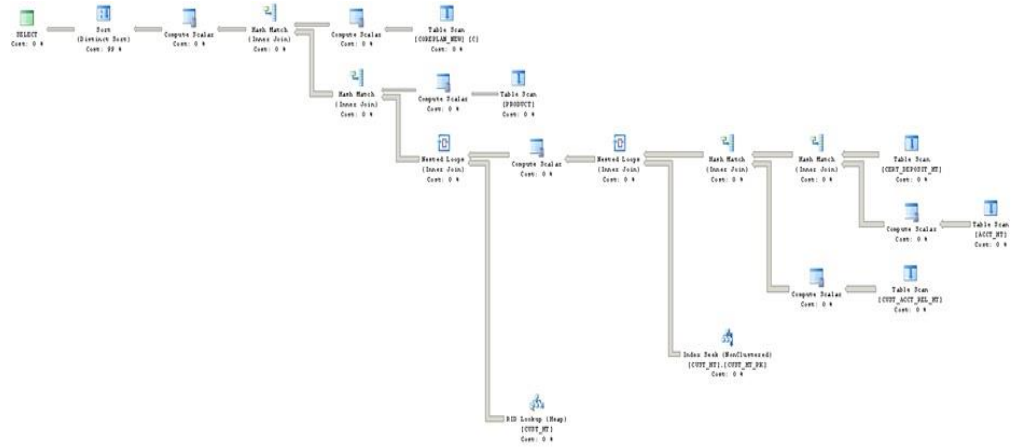
Different query characteristics and different tables being used in the joins impact the run time differently. Query8 runs for a much shorter time with compression turned on, whereas other queries display different behavior. In most cases, compression turned on for the four key tables seems to work better.The key difference between Query1, Query2, and Query8 seems to be the additional CUST_PERS_HT table involved in the join and the selection of different grouping sets. The compression setting should be set based on the most common queries.

Conclusion

This paper discusses a Fast Track configuration that we built. The configuration scaled well according to the demands put on it by the different tracks. We were able to deliver excellent performance numbers for serial runs both before and after optimization. Additionally, the performance numbers for volumized data scaled according to the growth in size of the data. We did additional testing on different approaches for loading the data, that is, BULK INSERT vs. the **bcp** utility. We also loaded data into databases with nonclustered and clustered primary keys and were able to show the workload performance numbers for both. The compression scenario delivered impressive compression ratios and indicated areas where it could be used for query performance optimization. The backup scenario proved that the speed and backup compression will be key wins for manageability of the data warehouse.We were able to conclude that the performance of the Fast Track configuration was better than current production by many orders of magnitude.We were able to deliver all of the above with a small team of six and complete the work of setting up the Fast Track config, translating the Oracle code, writing and optimizing the code, running tests, and documenting results within a short span of time. The performance was delivered by an older generation DL785 G5 Fast Track configuration. We believe we have satisfied and even exceeded the criteria of the POC by demonstrating great performance, amazing price/performance, and scalability.Given the opportunity to deliver the enterprise data warehouse, we can deliver workload performance far surpassing your expectations with the latest G7 hardware. Fast Track 3.0 specification with the G7 family of hardware takes advantage of the latest CPU technologies such as additional cores, higher clock speeds and cache sizes, plus larger disk sizes (600 GB vs. 300 GB) that provide more capacity with the same number of enclosures. This specification is expected to be out soon.We believe we can deliver the best performance through our expertise in optimization and migrating code from different database platforms.

Appendix B – Fast Track Queries, Their Plans, and Tuning Suggestions

Q1 – Coreplan – QueryPlan – Not Tuned



Execution Results

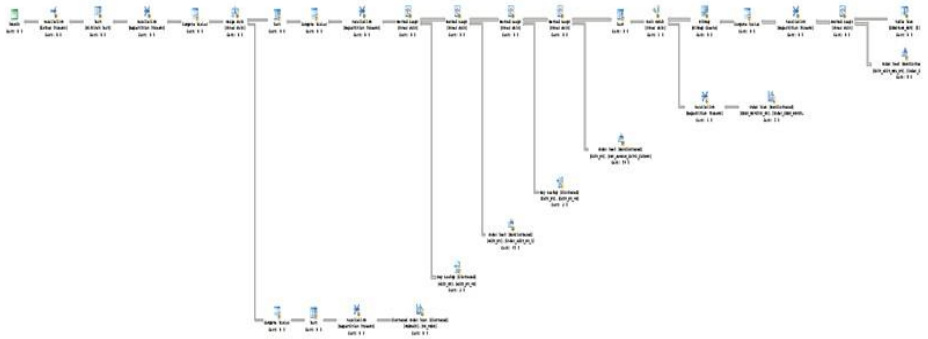
The execution time was 2 minutes.

Q1 – Coreplan

Optimization - Data Fixes

Remove leading and trailing spaces ahead of time rather than as part of the query.

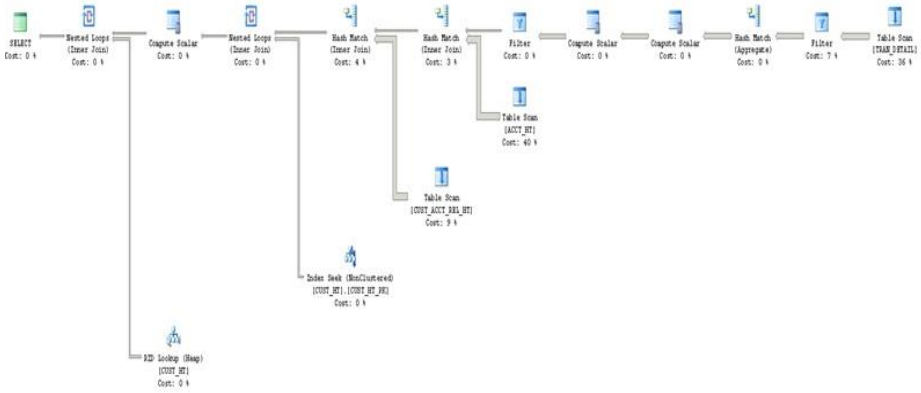
Query Plan



Execution Results

The execution time was 2 seconds.

Q2 – FINS_CASHDEP - Query Plan – Not Tuned



Execution Results

The execution time was 3 minutes and 2 seconds.

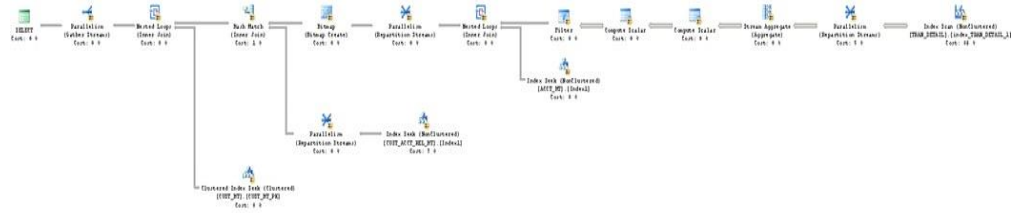
Q2 – FINS_CASHDEP

Optimization – Code Changes

Remove the function on the right side of the search condition. Create a parameter and assign the value from the function to it. Refer to the parameter directly.

```
declare @startdate datetimeset @startdate = DATEADD(D, -5, '31-jul-10') -- old way            --(TRAN_DETAIL.TXD_BUS_DT BETWEEN DATEADD(D, -5, ssma_oracle.to_date2('31-jul-10', 'dd-mon-yy')) AND ssma_oracle.to_date2('31-jul-10', 'dd-mon-yy')
-- better way             --(TRAN_DETAIL.TXD_BUS_DT BETWEEN DATEADD(D, -5, '31-jul-10') AND '31-jul-10'             --Best way - eliminate the function altogether.            (TRAN_DETAIL.TXD_BUS_DT BETWEEN @startdate AND '31-jul-10'
```

Query Plan

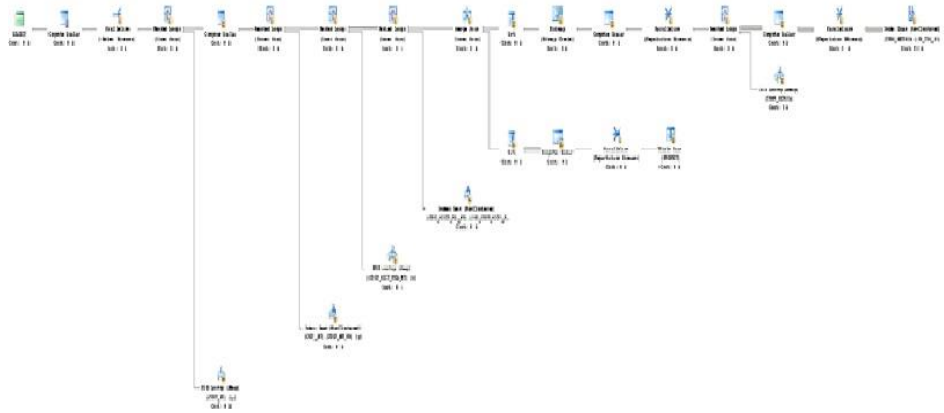


Execution Results

The execution time was 2 seconds.

Q3 – FINS_FDPLACE – Query – Not Tuned

Query Plan



Execution Results

The execution time was 2 seconds.

Q3 – FINS_FDPLACE

Optimization - Schema Changes

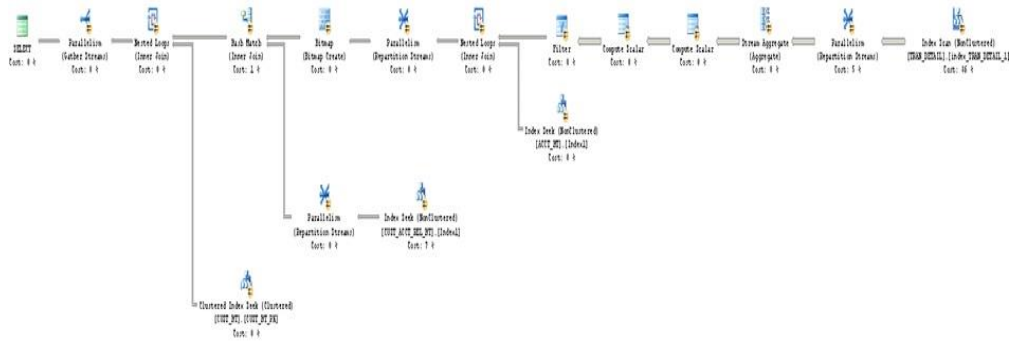
Precompute the columns that are used in function calls as part of search condition. Ideally, heavy functions should not be run on columns (especially in large tables) as part of a query. The calculated columns commonly used in queries should be setup as part of an ETL operation or as a computed column, persisted or otherwise.

```
ALTER TABLE dbo.TRAN_DETAIL ADD
      DT_CHOPPED_TXD_ACCT_NO_9_3 varchar(3) NULLgo    ALTER TABLE
dbo.TRAN_DETAIL ADD     DT_CHOPPED_TXD_ACCT_NO_10_3 varchar(3) NULLgo update
TRAN_DETAIL set DT_CHOPPED_TXD_ACCT_NO_9_3 =
substring(TRAN_DETAIL.TXD_ACCT_NO, 9, 3),DT_CHOPPED_TXD_ACCT_NO_10_3 =
substring(TRAN_DETAIL.TXD_ACCT_NO, 10, 3)
```

Optimization - Code Changes

Refer to the precomputed column directly in the query.

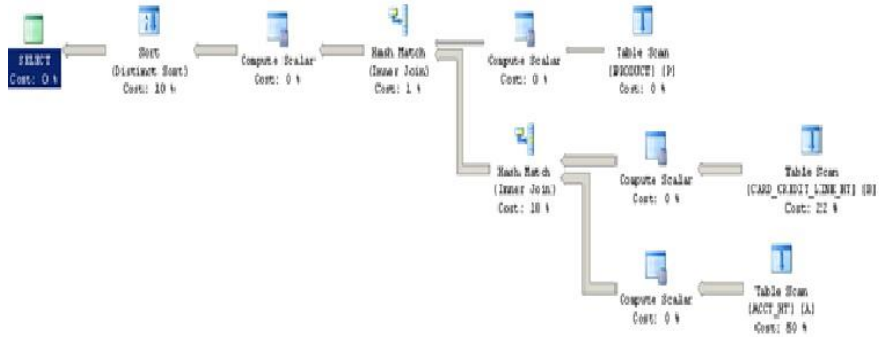
Query Plan



Execution Results

The execution time was 1 second.

Q4 – MUTIARA_CPS - Query Plan – Not Tuned



Execution Results

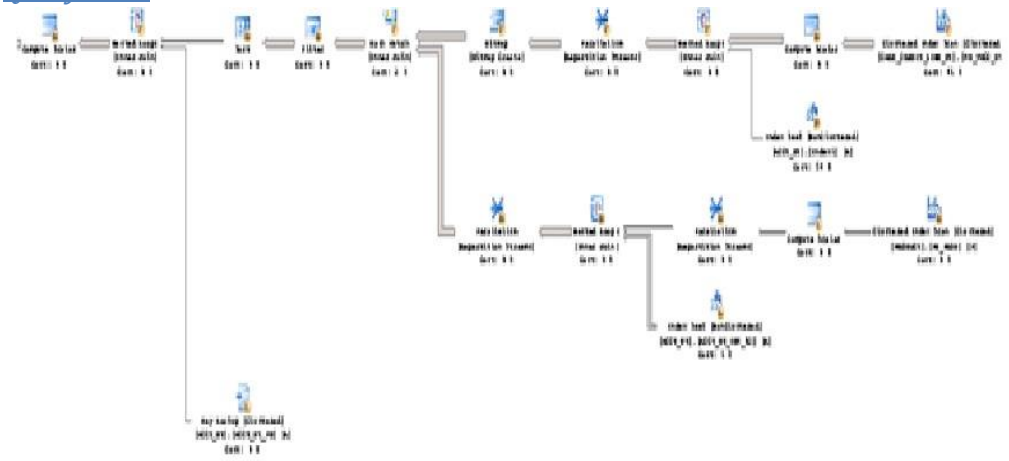
There were multiple attempts. The best execution time was 2 minutes and 14 seconds.

Q4 – MUTIARA_CPS

Optimization - Code Changes

Replace references to Oracle-like function calls from the SQL Server Migration Assistant (SSMA) tool with native Transact-SQL calls. `(ROUND(ssma_oracle.datediff(A.ACCT_BUS_CYC_DT, A.ACCT_LIMIT_APPR_DT) / 30,0)) AS MD_MTH_ON_BOOK, (ROUND(ssma_oracle.datediff(A.ACCT_BUS_CYC_DT, A.ACCT_LIMIT_APPR_DT) / 30,0)) AS MD_LN_AGE,` was changed to `(ROUND((convert(float(53),convert(datetime,A.ACCT_BUS_CYC_DT) - convert(datetime,A.ACCT_LIMIT_APPR_DT)) ) / 30,0)) AS MD_MTH_ON_BOOK, (ROUND((convert(float(53),convert(datetime,A.ACCT_BUS_CYC_DT) - convert(datetime,A.ACCT_LIMIT_APPR_DT)) ) / 30,0)) AS MD_LN_AGE,` Replace the old-style join with a JOIN statement.`FROM dbo.ACCT_HT AS A LEFT OUTER JOIN dbo.PRODUCT AS P ON A.ACCT_PROD_CD = P.PROD_PROD_CD, dbo.CARD_CREDIT_LINE_HT AS BWHERE A.ACCT_ACCT_NO = B.VM_ACCT_NO AND` was changed to `FROM dbo.ACCT_HT AS A LEFT OUTER JOIN dbo.PRODUCT AS P ON A.ACCT_PROD_CD = P.PROD_PROD_CD JOIN dbo.CARD_CREDIT_LINE_HT AS B ON A.ACCT_ACCT_NO = B.VM_ACCT_NO WHERE` Another possible fix would have been to create a temp table or variable table that contained the values for the IN clause and forming a join rather than a subquery. The temp/variable table would have contained an inclusion list (IN) rather than an exclusion list (NOT IN). This fix was not put in place for the query code due to time limitation.

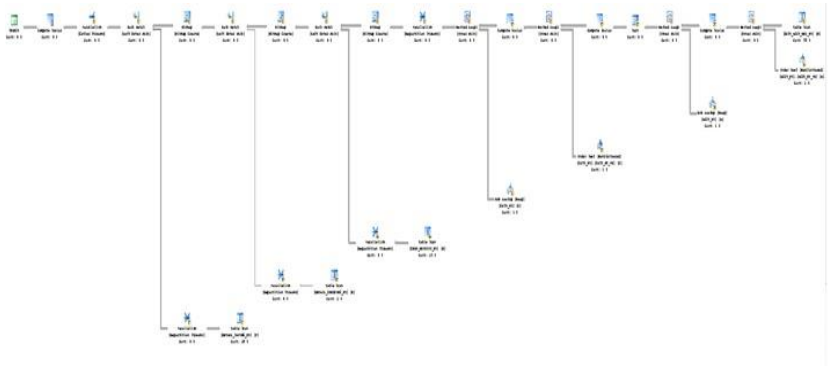
Query Plan



Execution Results

The execution time was 1 minute and 18 seconds.

Q5 – PIDM_APP1 - Query Plan – Not Tuned



Execution Results

The execution time was 4 seconds.

Q5 – PIDM_APP1

Optimization-Code Changes

There are multiple CASE statements like this one that can be replaced. These statements should be replaced by ETL processes that build the necessary computed columns if the computed columns are frequently used. It's not ideal to have multiple CASE statements within a query statement.

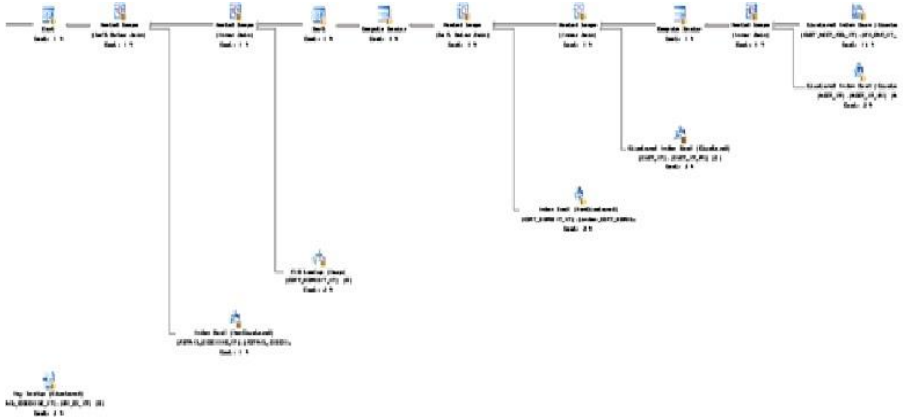
```
CASE WHEN A.ACCT_APPL_SYS_ID = 'STS' AND substring(A.ACCT_ACCT_NO, 9, 3) IN ( '001', '003', '007', '008' ) THEN 2 WHEN A.ACCT_APPL_SYS_ID = 'IMS' AND substring(A.ACCT_ACCT_NO, 10, 3) IN ( '201' ) THEN 2 WHEN A.ACCT_APPL_SYS_ID = 'STS' AND substring(A.ACCT_ACCT_NO, 9, 3) IN ( '002', '004', '013', '009' ) THEN 1 WHEN A.ACCT_APPL_SYS_ID = 'IMS' AND substring(A.ACCT_ACCT_NO, 10, 3) IN ( '202' ) THEN 1 END AS BANK_TYP
```

The following is a low-impact change that precalculates the CONVERT statement in a variable and feeds it into the search condition.

```
WHERE A.ACCT_ACCT_ID = B.CAR_R_ACCT_ID AND C.CUS_KEY = B.CAR_CUST_KEY AND A.ACCT_APPL_SYS_ID IN ( 'STS', 'IMS' ) AND B.CAR PRIME NO = 'Y'/*onli take primary holder*/ AND (B.CAR EXPIRATION DT >= '31-JUL-2010' OR B.CAR_EXPIRATION_DT = CONVERT(datetime2, '01/01/0001', 101))
```

We did not apply these changes to our query because it was already high-performing.

Query Plan



Execution Results

The execution time was 2 seconds.

Stored Procedures

Comments

CTSM_ME_PROD_PARTITION performed approximately 18 times faster than it did in the existing production environment without optimizations. MUTIARA_MAIN_JUL10 performed approximately 21 times faster without optimizations. DCS_MERGED_IND_SP consists of three heavy procedures being called by a large base proc. The three procedures; all used cursors to perform precalculations. This is not an ideal way to perform the operations. Kpi_Run_On_Age performs heavy CASE-based operations that will affect the time taken to execute the queries.

Stored Procedure 1 - CTSM_ME_PROD_PARTITION

Execution Results

The execution time was 01:06 (1 minute and 6 seconds).

Optimization – Code Changes

We converted the execute statements to direct Transact-SQL statements. For example, `EXECUTE ('TRUNCATE TABLE CTSM')` was changed to `TRUNCATE TABLE CTSM`.We also converted old-style joins to Transact-SQL joins. We applied the query hint FORCE ORDER to specify that the join order indicated by the query syntax should be preserved during query optimization.We converted Oracle-like function calls from SSMA to direct Transact-SQL calls. `INSERT dbo.CTSM_LOG(LOG_TIME, LOG_PROCESS_NAME) VALUES (ssma_oracle.to_char_date(sysdatetime(), 'DD-MON-YYYY HH:MI:SS AM'), 'CTSM START')` was changed to `INSERT dbo.CTSM_LOG(LOG_TIME, LOG_PROCESS_NAME) VALUES (CONVERT(varchar(32), sysdatetime(), 109), 'CTSM START')`

Execution Results

The execution time was 1 minute and 17 seconds.

Stored Procedure 2 - MUTIARA_MAIN_JUL10

Execution Results

The execution time was 11 minutes and 29 seconds.

Optimization – Code Changes

We converted Oracle-like SSMA function calls to native Transact-SQL calls.We converted old-style joins to Transact-SQL join statements.We consolidated individual UPDATE statements into a single statement.

Execution Results

The execution time was 06:40 (6 minutes and 40 seconds).

Stored Procedure 3 - DCS_MERGED_IND_SP

Execution Results

The query ran for a long time, and it was terminated after 3 hours.

Optimization - Data Fixes

We fixed data by removing leading and trailing spaces. The data contained either leading or training spaces. This forced the code to constantly trim left and right in this fashion. We converted EXECUTE statements to direct Transact-SQL statements.As a second phase of fixes, code using cursors in the

procedures DCS_IND_2B_DUP_STS, DCS_IND_2B_DUP_IMS, and DCS_IND_2A_DUP_IMS were replaced with straight Transact-SQL statements.

Execution Results

The execution time was between 3 and 5 minutes.

Stored Proc 4 - Kpi_Run_On_Age

Execution Results

The execution time was 1 hour, 15 minutes, and 41 seconds.

Optimization – Code Fixes

We replaced calls to Oracle-like SSMA function calls with a custom view.

```
WHEN floor(MONTHS_BETWEEN_PER_BUS_CYC_DT_PER_CUST_DOB / 12) BETWEEN  0 AND 12 THEN '12 YRS AND BELOW'
```

The following views were created to emulate the MONTHS_BETWEEN function.

```
CREATE VIEW CUST_PERS_HT_MONTHS_BETWEEN_PER_BUS_CYC_DT_PER_CUST_DOBASSELECT -- MONTHS_BETWEEN PER_BUS_CYC_DT PER_CUST_DOBcase when ( (dateadd(dd, -day(dateadd(mm,1,PER_BUS_CYC_DT)), dateadd(mm,1,PER_BUS_CYC_DT)) = PER_BUS_CYC_DT) and (dateadd(dd, - day(dateadd(mm,1,PER_CUST_DOB)), dateadd(m,1,PER_CUST_DOB)) = PER_CUST_DOB) ) or (day(PER_BUS_CYC_DT) = day(PER_CUST_DOB))      then 12*(year(PER_BUS_CYC_DT) - year(PER_CUST_DOB)) + month(PER_BUS_CYC_DT) - month(PER_CUST_DOB)      else 12*(year(PER_BUS_CYC_DT) - year(PER_CUST_DOB)) + month(PER_BUS_CYC_DT) - month(PER_CUST_DOB) +convert(numeric(38,19), PER_BUS_CYC_DT - dateadd(mm,datediff(mm,0,PER_BUS_CYC_DT),0) - (PER_CUST_DOB - dateadd(mm,datediff(mm,0,PER_CUST_DOB),0)) ) / 31end as MONTHS_BETWEEN_PER_BUS_CYC_DT_PER_CUST_DOBFROM CUST_PERS_HTgo
```

Execution Results

The execution time was between 2 and 4 minutes.

Section 4: Performance

Tuning the Performance of Backup Compression in SQL Server 2008

Overview

Backup compression is a new feature in SQL Server 2008 that can help provide smaller sized backups and reduce backup time. This document provides guidance related to tuning options for backup performance. All of the information and test results presented here were done specifically by using the backup compression feature of SQL Server 2008; however, they apply broadly to any backup scenario whether backup compression is used or not. They also apply to restore operations; however, restore will not be covered in depth in this document. For an introduction to the backup compression feature, see Backup Compression, in SQL Server Books Online.

Benefits of Backup Compression

One major benefit of backup compression is space saving. The size of the compressed backup is smaller than that of the uncompressed backup, which results not only in space savings, but also in fewer overall I/O operations during backup and restore operations. The amount of space you save depends upon the data in the database, and a few other factors, such as whether the tables and indexes in the database are compressed, and whether the data in the database is encrypted. To determine the effectiveness of backup compression, the following query can be used:

```
SELECT
  b.database_name 'Database Name',
  CONVERT (BIGINT, b.backup_size / 1048576 ) 'UnCompressed Backup Size (MB)',
  CONVERT (BIGINT, b.compressed_backup_size / 1048576 ) 'Compressed Backup
Size (MB)',
  CONVERT (NUMERIC (20,2), (CONVERT (FLOAT, b.backup_size) /
  CONVERT (FLOAT, b.compressed_backup_size))) 'Compression Ratio',
  DATEDIFF (SECOND, b.backup_start_date, b.backup_finish_date) 'Backup
Elapsed Time (sec)'
FROM
  msdb.dbo.backupset b
WHERE
  DATEDIFF (SECOND, b.backup_start_date, b.backup_finish_date) > 0
  AND b.backup_size > 0
ORDER BY
  b.backup_finish_date DESC
```

Table 1 shows the output of the above query after taking an uncompressed and a compressed backup.

Database Name	Uncompressed Backup Size (MB)	Compressed Backup Size (MB)	Compression Ratio	Backup Time (Seconds)	Comments
BCTEST	292705	95907	3.05	1705	Compressed backup
BCTEST	292705	292705	1	3348	Uncompressed backup

Table 1: Comparing compressed and uncompressed backup

For an uncompressed backup, the compression ratio is 1 and the Compressed Backup Size reports the same value as the Uncompressed Backup Size. The higher the compression ratio, the greater the space saving. For our test database, we achieved a compression ratio of 3.05 and saved approximately 67% space for the backup. Figure 1 illustrates the benefits of backup compression in terms of space and time. Databases using the Transparent Database Encryption (TDE) feature of SQL Server 2008 may not see as high backup compression ratios (in most cases, it will be close to 1) due to the fact that encrypted data does not lend itself well to compression.

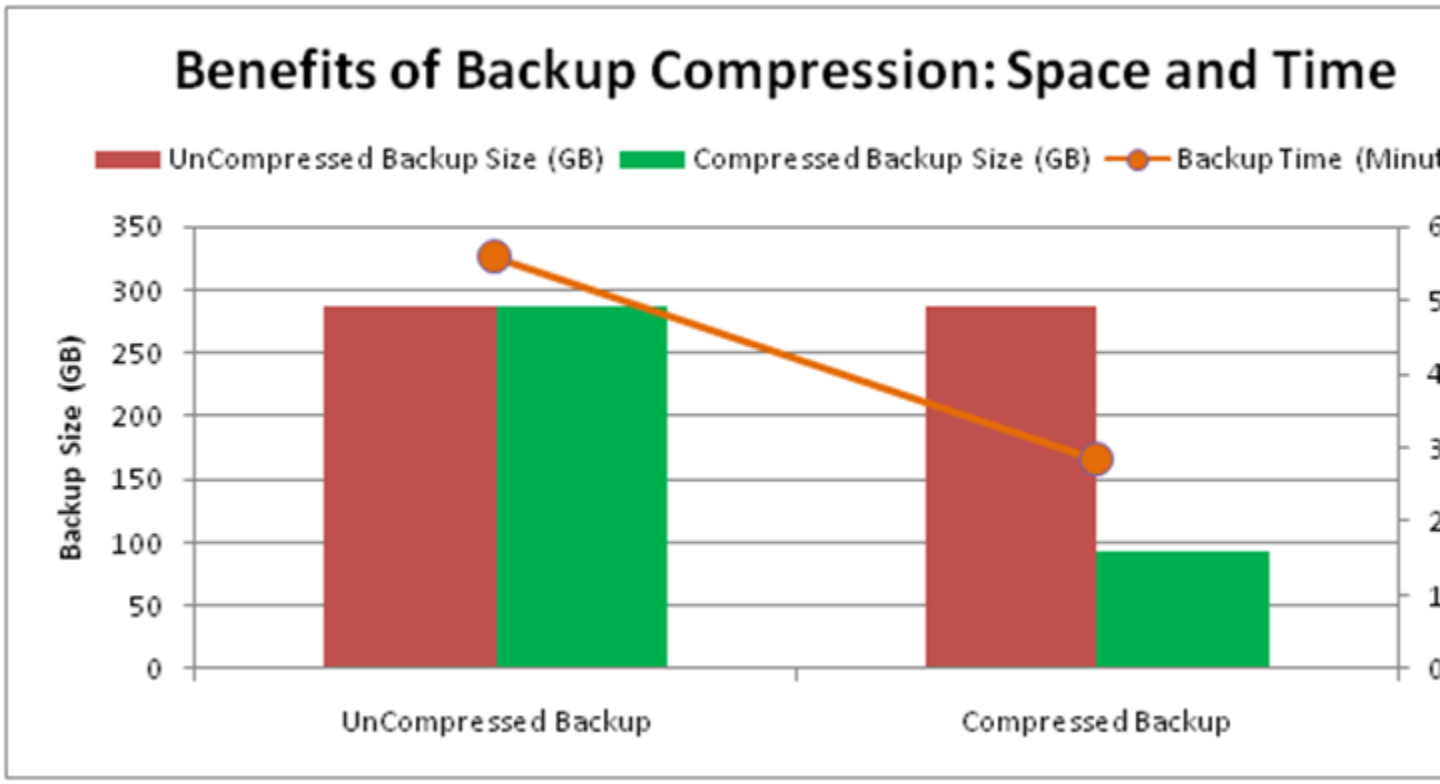


Figure 1: Benefits of backup compression in terms of space and time

As illustrated in Figure 1, a compressed backup is smaller in size, and hence requires fewer write I/Os to the backup media. This results in reduced backup time. Backup is an I/O intensive operation so reduction in I/O will be beneficial to performance.

Test Workload and Test Environment

Our test environment for these results consisted of a database representing an OLTP stock trading application with a database size (excluding free space) of approximately 300 GB. The hardware used for the testing was a 4-socket dual core DELL 6950 server and an EMC Clariion (CX700) storage array with 4GB of cache allocated 80% to write operations. The volume/LUN layout is shown in Table 2.

Volume	Purpose	RAID Level	Number of Disks

E:	Data files	1+0	8
F:	Data files	1+0	8
G:	Data files	1+0	8
H:	Data files	1+0	8
L:	Log files	1+0	8
M:	Backup files	1+0	8
P:	Backup files	1+0	2
Q:	Backup files	1+0	2
R:	Backup files	1+0	2
S:	Backup files	1+0	2

Table 2: Disk volume layout for backup compression tests

In each of the disk volumes shown in Table 2 there was no sharing of physical disks between any of the volumes. There were 32 total disks for the data, 8 for the log and 16 for the backup files.

SQL Server data files were striped such that each filegroup contained four data files—each file of a filegroup was placed on a separate data volume.

Determining Achievable Throughput of Backup Compression

Whenever you attempt to tune the performance of backup compression, it is important to first understand the achievable theoretical throughputs of the given hardware configuration, specifically the storage. The throughput of backup compression is bound by the CPU resources, the throughput of the input devices, or the throughput of the output devices.

One method of determining the throughput of a given configuration is to perform backup to a NUL device multiple times by varying the BUFFERCOUNT setting. Use the default BUFFERCOUNT first, and then explicitly specify BUFFERCOUNT in subsequent tests—each time increasing BUFFERCOUNT to a higher value. For example:

```
BACKUP DATABASE [BCTEST] TO DISK = 'NUL' WITH COMPRESSION

BACKUP DATABASE [BCTEST] TO DISK = 'NUL' WITH COMPRESSION, BUFFERCOUNT = 50
```

Default BUFFERCOUNT is determined dynamically by SQL Server based on the number of database volumes and output devices. Default values for BUFFERCOUNT are chosen so they will apply to a broad range of environments which is why tuning these on high end systems may be necessary to obtain optimal performance.

More information on tuning the BUFFERCOUNT parameter will be given later in this document. The BACKUP section in SQL Server Books Online also contains related information.

At some point during the above tests, you will observe one of the following, which will determine the limits of the hardware:

- The total CPU utilization will be near 100%.
- The disk throughput will remain constant, while the latency increases.

If total CPU utilization nears 100% across all cores, this is your bounding resource. If the total CPU is less than 100%, the observed throughput of the reads is your maximum achievable throughput. Once you have this value, you can then perform a backup to a real output device. If you do not achieve the same throughput attained by using BACKUP to NUL, write throughput to the output device is your bottleneck.

Figure 2 illustrates the throughput we achieved as we increased BUFFERCOUNT during our BACKUP to NUL tests.

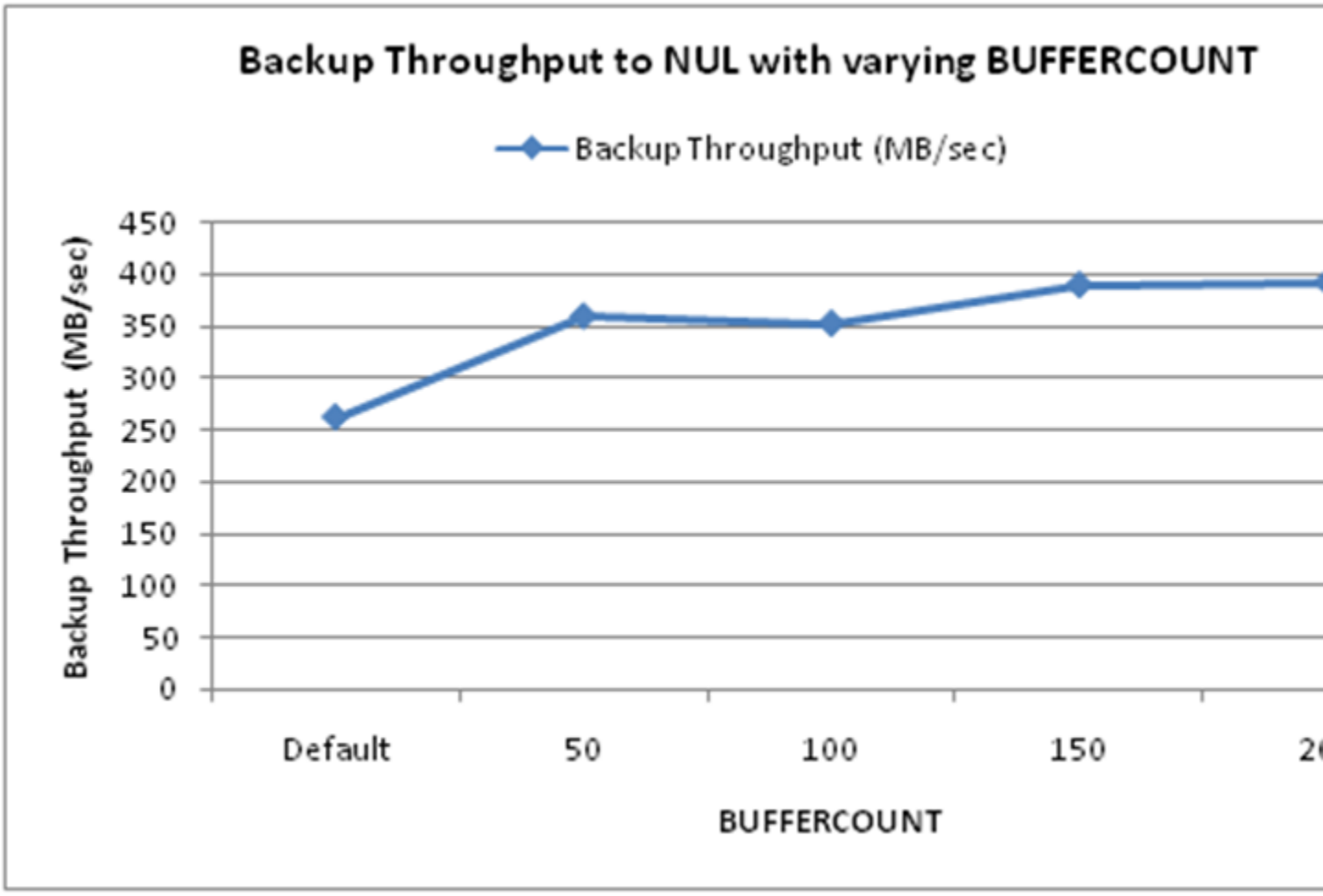


Figure 2: Backup throughput to NUL device with varying BUFFERCOUNT

Similarly, the performance monitor graph in Figure 3 illustrates the fact that I/O on the read device is our bounding resource. This performance monitor log was run continuously across all iterations of our tests. Our achieved throughput was approximately 400 MB/sec before the read operations became I/O bound.

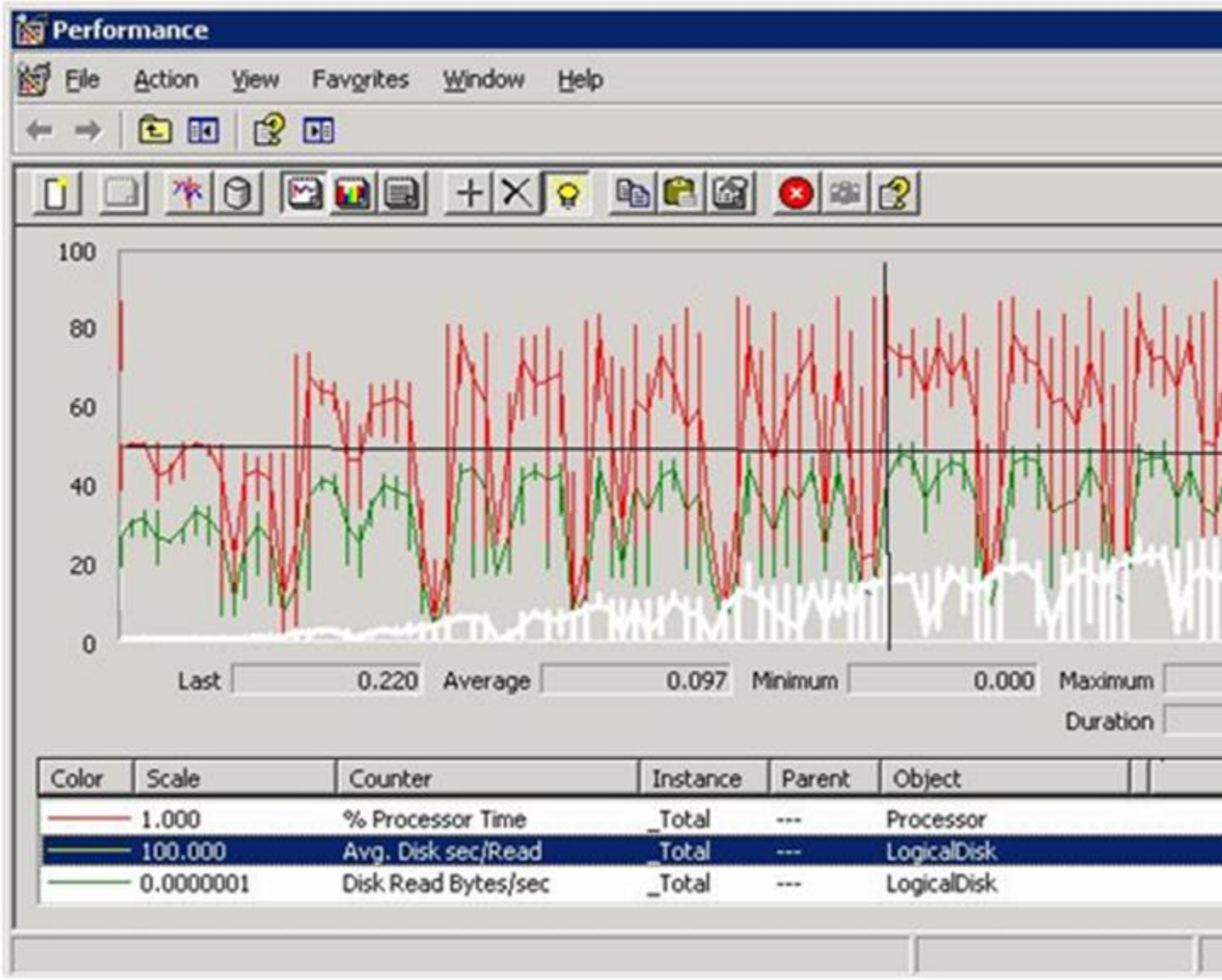


Figure 3: Disk throughput and latency during backup compression

Notice that near the intersection of the black straight lines there is a leveling off in the throughput (Disk Read Bytes/sec counter), while the latency (Avg. Disk sec/Read counter) continues to increase. This indicates a bottleneck on the I/O resources.

Ultimately, in our tests with a real output device this was the throughput we achieved, which means the bottleneck was on the I/O performance of our input devices.

Considerations for Tuning Backup Performance

Several considerations come into play when you attempt to increase the performance of compressed backup operations. The goal of our testing was to determine the highest throughput obtainable with the simplest configuration. Performance of backup operations in SQL Server is influenced by the following:

- Database disk volume and backup device layout
- Size of I/Os and number of outstanding I/O requests
- Hardware configuration

Database Volume and Backup Device Layout

Database volume and the device layout have an impact on backup performance, irrespective of whether the backup is compressed. For backup operations, there is a single reader thread per database volume (drive letter or mount point volume), and a single writer thread per backup device. Figure 4 illustrates the performance differences of backup compression between the following configurations. In this article, a backup device is synonymous with a file.

- Single backup volume (M) with a single backup device
- Single backup volume (M) with four backup devices
- Four backup volumes (P, Q, R, and S) with four backup devices

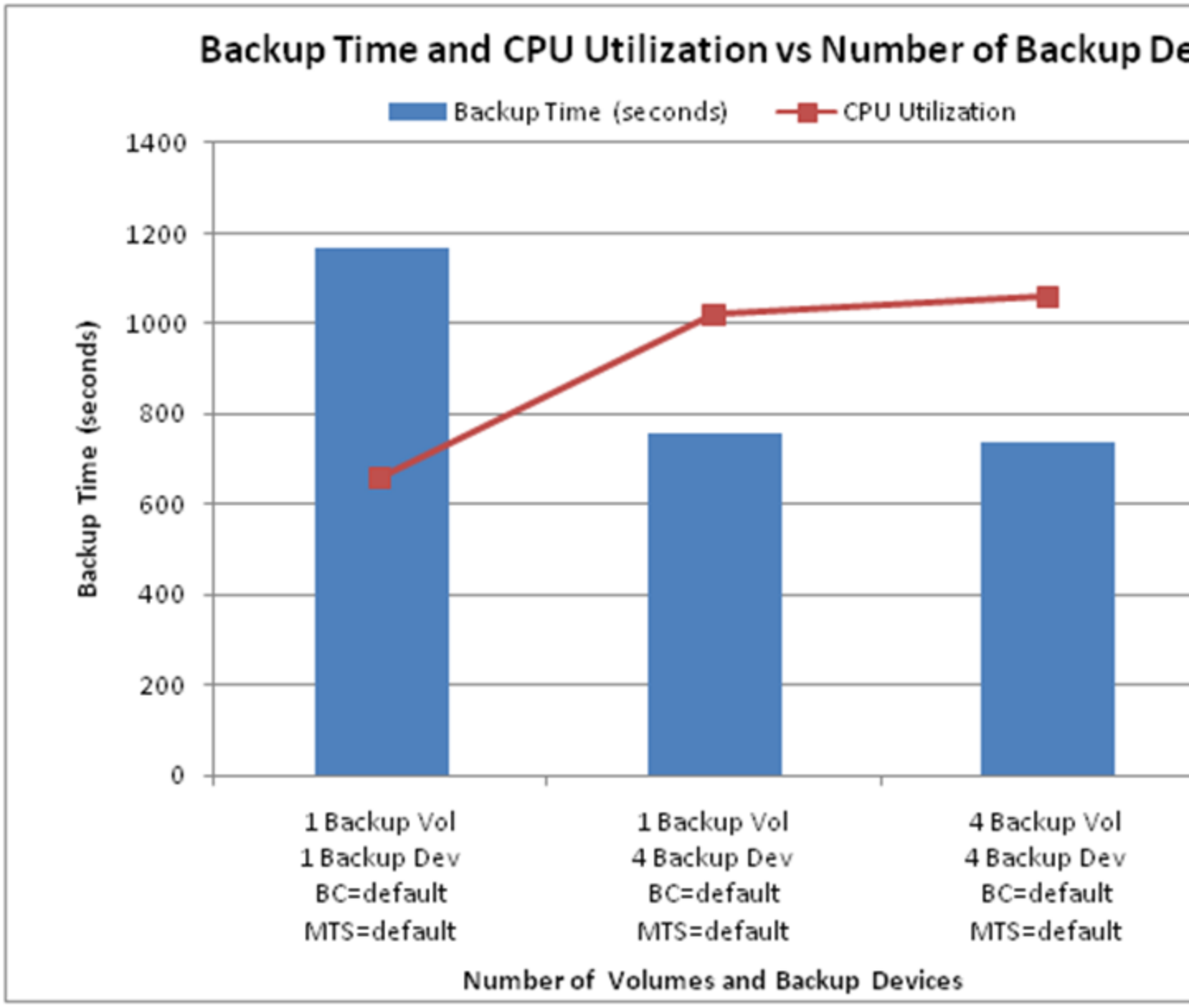


Figure 4: Backup compression performance versus number of backup devices

In Figure 4, BC refers to BUFFERCOUNT and MTS refers to MAXTRANSFERSIZE. These settings are discussed in more detail later in this document.

As shown in Figure 4, having multiple backup devices improves backup performance. We observed very slight improvement by having the multiple backup devices on separate volumes. This is likely because our test was bound by read throughput as illustrated in Figure 3.

BUFFERCOUNT and MAXTRANSFERSIZE were left at the default values for the test results in Figure 4. By default we observed MAXTRANSFERSIZE to be an average of 512 KB, and BUFFERCOUNT was 13 and 28 for 1 device and 4

devices respectively. SQL Server determines the default setting for BUFFERCOUNT based on the number of reader volumes and output devices. Using multiple backup devices results in an implicit increase in the BUFFERCOUNT value.

The increase in BUFFERCOUNT, in addition to the increased number of threads writing to the backup files explains why the performance of multiple devices on a single volume increases (described more later). Results using other values are shown later in Figure 6.

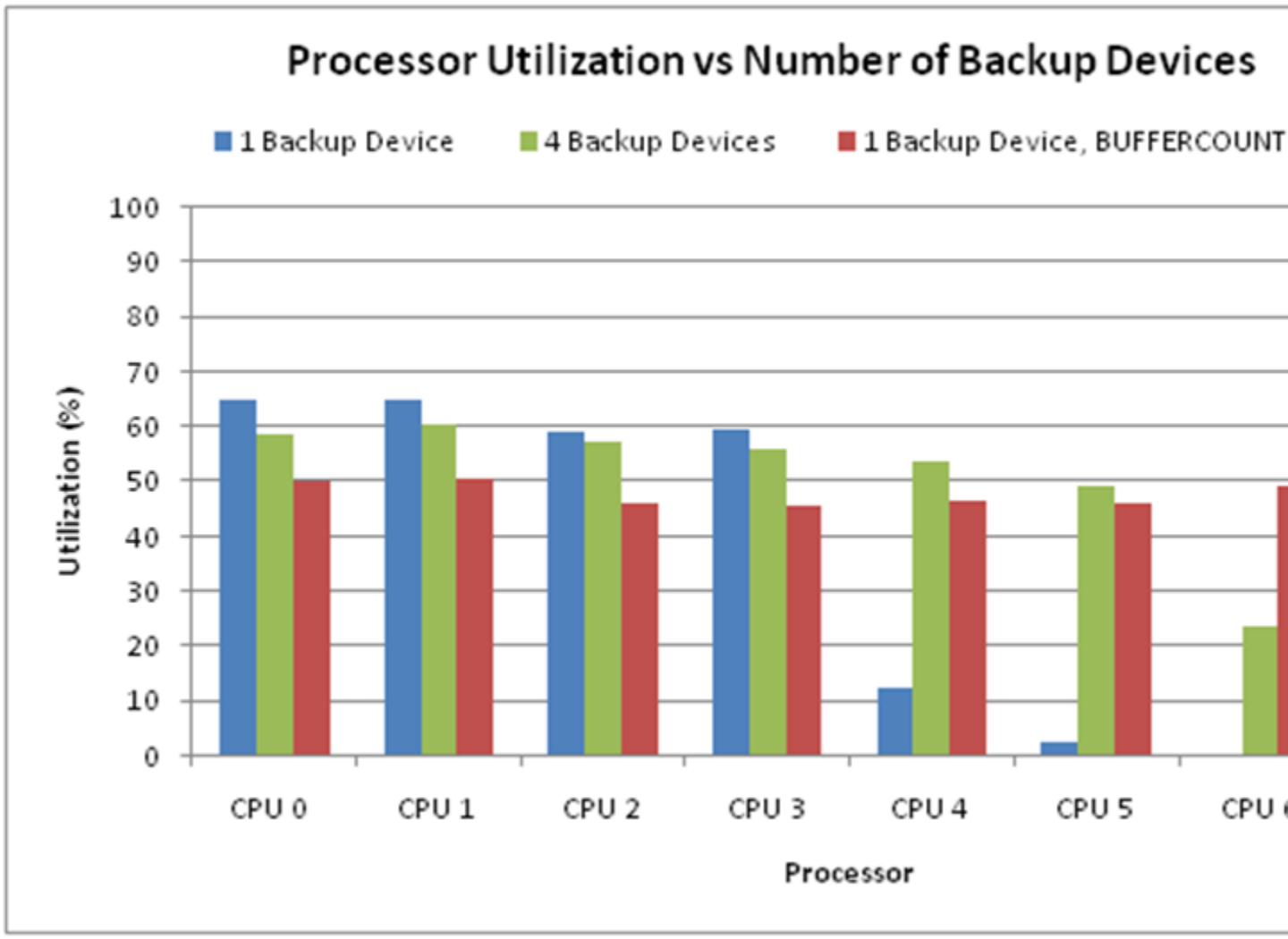


Figure 5: Parallelism influence on CPU utilization

The number of threads used for compression operations is dynamically determined and comes from the SQLOS thread pool. In our case, having either 1) increased BUFFERCOUNT with a single output device, or 2) multiple output devices, resulted in greater throughput and more concurrent compression operations. As shown in Figure 5, either of these increased parallelism and overall CPU usage across all CPUs in our server. A greater number of backup devices improves backup performance. However, it adds a management complexity. You have now multiplied the number of backup files to track. If you lose one of these files, the entire backup is useless.

Using backup compression results in higher CPU utilization when compared to uncompressed backups; this is due to the compression of the data. This cost is generally offset by the benefits of backup compression discussed earlier in this paper.

For backups that are executed concurrently with user workload, reducing the number of backup devices or BUFFERCOUNT may reduce the overall CPU used by the backup operation, leaving more CPU resources for the workload. This trade-off can be made to accommodate scenarios where CPU is a constrained resource that must be preserved for user workload.

Size of the I/Os and the Number of Buffers Used by Backup

In some scenarios, tuning the size of the I/O issued by backup / restore operations as well as the number of internal buffers used for the transfer of data may help increase throughput independently of the underlying volume / device layout. These are controlled by the MAXTRANSFERSIZE and BUFFERCOUNT options of the backup and restore commands. When tuning these, keep in mind the following:

- Memory used for backup buffers comes from virtual address space, which resides outside the buffer pool. The potential amount of memory used for these operations is equal to MAXTRANSFERSIZE * BUFFERCOUNT. Care should be taken on 32-bit systems because specifying values for these settings that are too high may result in out-of-memory errors since there is a limited amount of virtual address space available for this. On 32-bit systems, the default amount of virtual address space outside of the buffer pool is 256 MB.
- Choosing an initial value for BUFFERCOUNT can be difficult if you do not know the default value. This default value is dynamic and determined by SQL Server at run time. The default value is influenced by the number of backup devices and the number of database volumes. Testing is necessary to determine an optimal value. Running with trace flags 3605 & 3213 will result in output to the ERRORLOG containing the number of buffers used for backup operations.
- With the exception of very small databases and snapshot backups of VLDB using VDI technologies, the default MAXTRANSFERSIZE is 1 MB.
  Note: Snapshot backups taken by using VDI integrated storage solutions cannot use backup compression.
- I/O size issued for backup / restore operations ranges from 64 KB up to MAXTRANSFERSIZE, based on the allocation of data within a data file. MAXTRANSFERSIZE refers to the size of the I/O operation issued to read data from the database files. Only contiguous regions of data on the disk are read. The continuity of the underlying data pages has an impact on the size of the I/Os. Therefore, the actual I/O size you observe may be less than the value of MAXTRANSFERSIZE specified. The default value for MAXTRANSFERSIZE is 1 MB.

Figure 6 extends the data in Figure 4 by including tuning of MAXTRANSFERSIZE and BUFFERCOUNT. Figure 6 shows backup time and CPU utilization for six different configurations. As illustrated in Figure 6, throughput comparable to that attained by using multiple devices can be achieved when using a single device on a single LUN by adjusting the MAXTRANSFERSIZE and BUFFERCOUNT parameters.

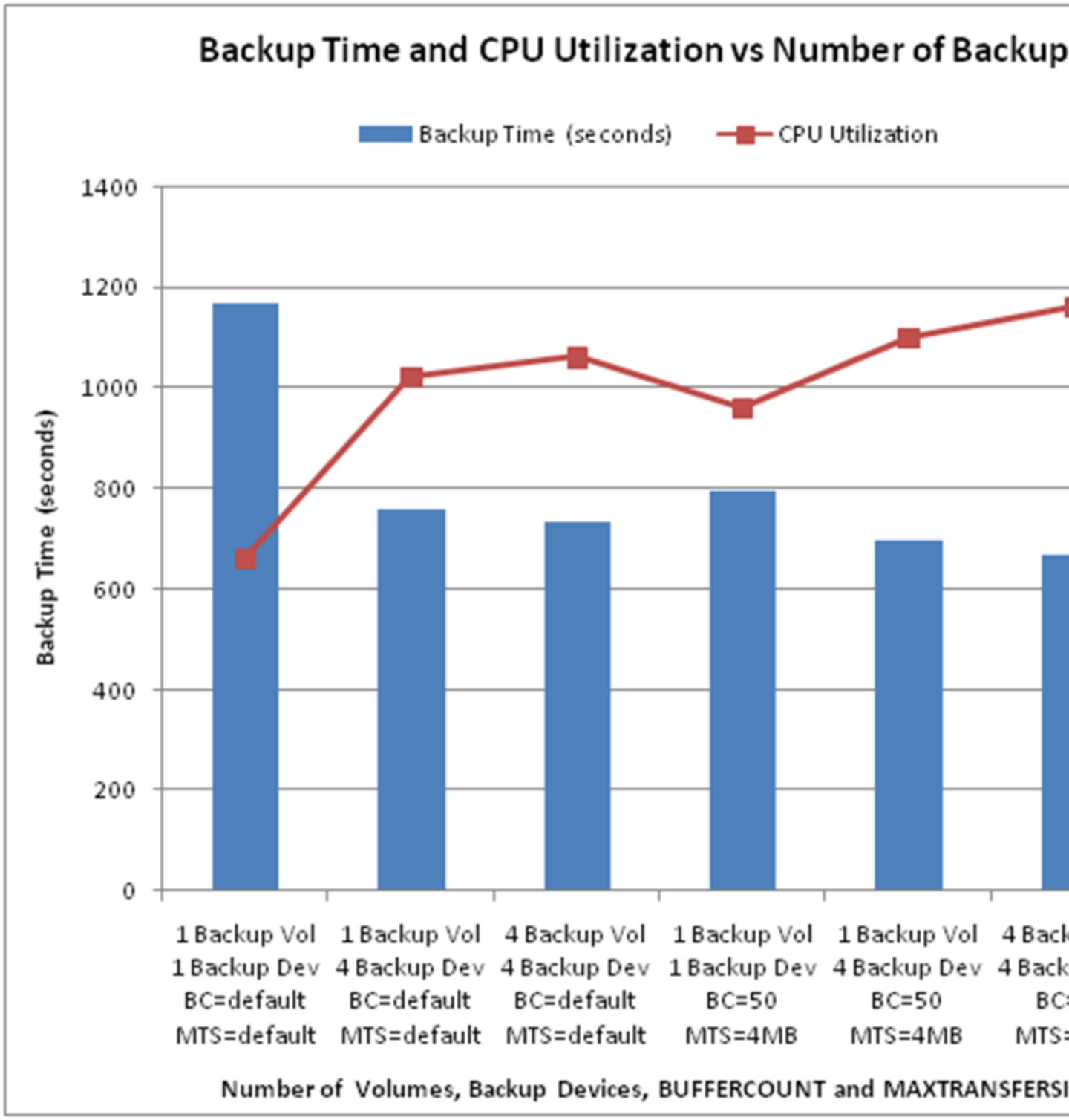


Figure 6: Backup time and CPU utilization (average for all CPU's) versus device/volume configuration, MAXTRANSFERSIZE and BUFFERCOUNT

Although tuning MAXTRANSFERSIZE may result in slightly higher throughput, it also results in much larger I/O sizes. Larger I/O sizes of the backup operation when mixed with a concurrent OLTP workload may result in slower overall

I/O response times. The recommended tuning approach is to adjust BUFFERCOUNT first, and then determine if tuning MAXTRANSFERSIZE will provide any additional performance advantage. Keep in mind when tuning this parameter that increasing the value may increase the size of the I/O with limited increase in overall throughput. Using multiple output devices as opposed to tuning these parameters may be a simpler approach to achieving greater throughput. However, this also may introduce the additional management overhead of having to maintain multiple backup files. Due to time constraints, we did not test all configurations in our scenario.

Figures 7 shows the read and write throughput attained during these tests. The write throughput is less than the read throughput because of compression.

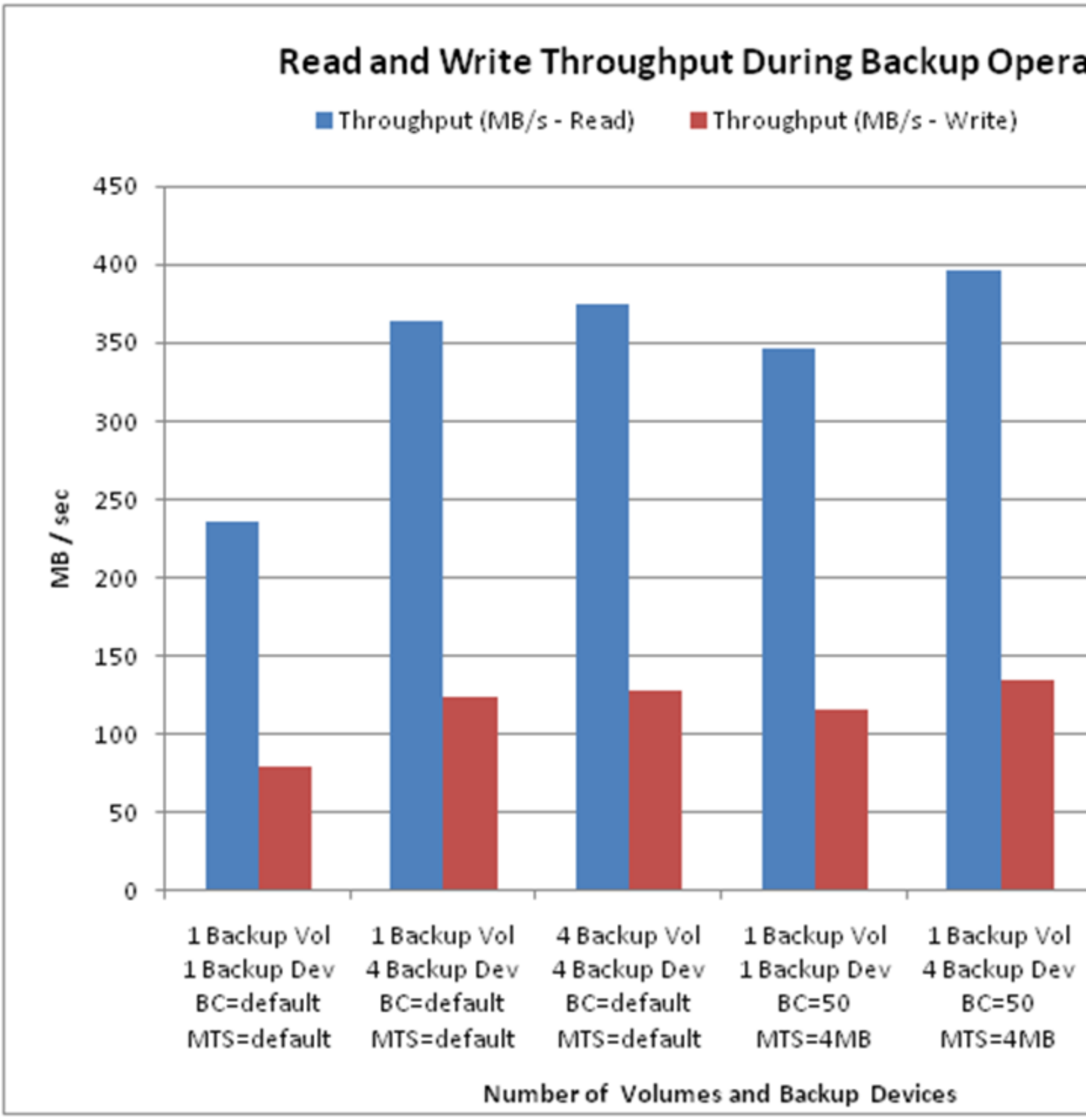


Figure 7: Read and write throughput during backup operations

Hardware Configuration

Backup compression operations are parallelized dynamically up to the number of CPUs on the machine so there is a potential performance benefit to a machine with more physical CPUs. Similarly, the physical characteristics of the I/O configuration including the number of disks, cache memory available for I/O operations, and the throughput attainable on the I/O path, have a significant effect on performance. I/O operations related to backup / restore may ultimately be bound by the limits of the hardware. It is important to understand the potential throughput of your specific configuration in order to know when you are exhausting the capabilities of the hardware.

Recommendations

This article provides a good amount of detail into the internals of backup compression. A summary of our recommendations for tuning backup compression and restore are:

- Use the method of backing up to NUL to determine the potential throughput attainable on your particular configuration.
- Performance of backup operations may be increased by utilizing multiple backup devices and/or increasing BUFFERCOUNT. Using multiple devices over a single device with increased BUFFERCOUNT may offer slight additional benefit; however, use of a single device with increased BUFFERCOUNT may provide nearly the same throughput without the need to manage multiple backup files.
- Keep in mind that adjusting backup parameters to increase the performance of backup may have adverse impact to user workload running on the system. Always consider what is best for your particular scenario.
- MAXTRANSFERSIZE should be considered a secondary tuning approach. In our tests this provided little practical benefit over the default used by BACKUP
- Consider the parallelism benefits of volume design on backup / restore operations at database creation time. This is one of the many things to consider when determining how many volumes to use for a SQL Server database.
- Monitor your I/O and CPU utilization during tests to ensure that any performance bottlenecks you encounter are not hardware related.
- Be careful while increasing the value of the BUFFERCOUNT parameter on 32-bit systems because specifying too high values for this may result in out-of-memory errors, since there is a limited amount of virtual address space available for this. On 32-bit systems, the default amount of virtual address space outside of the buffer pool is 256 MB.

Keep in mind that the results described here indicate the performance that is attainable in one specific configuration. Performance may be much higher or lower in different configurations. As an additional data point, Backup More Than 1GB per Second Using SQL2008 Backup Compression on the SQLCAT blog illustrates the performance attainable on very high-end systems.

Summary

There are many approaches to tuning the performance of backup in SQL Server 2008. Understanding how volume and device layout, and number of CPUs influence parallelism, as well as understanding other tuning options can provide you with the ability to significantly increase the performance of backup and restore. The decisions on which approach to use depend largely on what is optimal for your given environment.

Appendix A: Test Hardware and Software

Server

DELL PowerEdge 6950

- 4 socket dual core
- AMD Opteron 2.8 GHz
- x64
- 32 GB RAM

Storage

EMC Clariion CX700

- 10K SCSI drives
- 4 GB cache (80% write, 20% read)
- 2 HBA's

Software

- Windows Server 2003 Enterprise Edition Service Pack 2 x64
- SQL Server 2008 February CTP x64

SQL Server Indexing: Using a Low-Selectivity BIT Column First Can Be the Best Strategy

Overview

When discussing indexing strategies, Microsoft SQL Server customers frequently claim that you should never place indexes on columns with few values or columns of data types such as BIT or gender. One customer pointed me to sys.dm_db_missing_index_details dynamic management view (DMV), claiming that if they followed the guidance to put the equality column first, a BIT column would be the first column in the index, and this would be a problem.

In fact, there are times when using a column with few values as the first column in an index is the best option, and this includes times when a BIT column or a gender column is first in the index.

This technical note shows that columns such as BIT columns can sometimes be the best choice for an index and that SQL Server can determine when this is the case. This technical note uses an example to demonstrate this and to show that SQL Server does know that the index with the equality column first is the better choice—even if the equality column is a BIT column.

NOTE: This note is intended for experienced database administrators (DBAs) and software developers who have an understanding of SQL Server fundamentals.

Compare: Place the BIT Column First

Consider the following query:

```
SELECT COUNT(*)

FROM testTable

WHERE bitCol = 1 AND intCol BETWEEN 2 AND 5
```

In this case:

- **bitCol is an equality column**. You are looking for one specific value in bitCol.
- **intCol is an inequality column**. You are looking for a range of values in intCol.

Some guidance recommends putting the most cardinal column first in an index. Other guidance recommends that you put the equality column first in an index. In this case, this means putting a BIT column first.

To see which is the right choice, examine a very small set of data to see the differences between using a unique index on (bitCol, intCol) and a unique index on (intCol, bitCol). Consider a table with intCol values from 1 to 10 and bitCol values of 0 and 1.

Sort with the Inequality Column First
The index on (intCol, bitCol) might result in the following logical data arrangement:

intCol	bitCol			
1	0			
1	1			
2	0			
2	1			First row meeting conditions
3	0			
3	1			
4	0			Rows Evaluated
4	1			
5	0			
5	1			
6	0			Row to determine out of range
6	1			
7	0			
7	1			
8	0			
8	1			
9	0			
9	1			
10	0			
10	1			

Table 1: Data ordered by (intCol, bitCol)

Starting from the first row that satisfies the condition in the query's WHERE clause and counting to the last row that satisfies the condition, you can see that seven rows must be evaluated to return four rows. The next row after the last one satisfying the condition must always be evaluated in a range query to know that the end of the range has been reached, so *eight rows must actually be evaluated to find the four rows that actually satisfy the condition*.

Sort with the Equality Column First
Now look at the same data indexed by (bitCol, intCol):

bitCol	intCol			
0	1			
0	2			
0	3			
0	4			
0	5			
0	6			
0	7			
0	8			
0	9			
0	10			
1	1			
1	2			First row meeting conditions
1	3			Rows evaluated
1	4			
1	5			
1	6			Row to determine out of range
1	7			
1	8			
1	9			
1	10			

Table 2: Data ordered by (bitCol, intCol)

Looking at the span between the first and last row that satisfy the query when the query is indexed like the example in Table 2, you can see that there are four rows in the range and one row to determine that

the range has been exceeded, so *five rows must be evaluated to return four rows that satisfy the conditions*.

What Does This Mean?
The data set used in the example above includes every possible combination for the range of data. This makes it the worst case scenario for both index possibilities.

You can see that in this worst case scenario putting the equality column (bitCol) first requires the evaluation of about half as many rows to return the same data set. This is true because there are only two possible values in bitCol. As the number of possible values for that equality column increases, the worst case scenario for indexing on the inequality column first becomes increasingly more “expensive” than indexing on the equality column first.

The best case scenario for putting the intCol first while returning the same result set would be if the original data set had no rows with a bitCol value of 0. If this were the data set, then the best case for putting intCol first requires evaluation of the same number of rows as putting the bitCol first. Therefore, the best case for indexing with the inequality column first is the same as best and worst case scenario of indexing with the equality column first.

This is true with other value combinations also. In any scenario other than best case, it is better to put the equality column first, and this is true even if the equality column is a BIT column.

To summarize:

- Sorting on the inequality column first can never be more efficient than sorting on the equality column first.
- There is only one case in which sorting on equality column first or inequality column first is equally efficient; in all other cases, sorting on the equality column first is better than sorting on the inequality column first.

Looking at a flat representation of a small data set, it is clear that you should put the equality column first in indexes to support queries with equality and inequality conditions. This may include times when a BIT column, or other column with low selectivity, appears first in an index.

Translating to a SQL Server Index Structure

A SQL Server index is a B+ tree: The root and intermediate pages of the index contain keys and pointers to lower-level pages to faciliitate navigation to the correct child page. All of the index data is stored on the leaf pages. While the data in the leaf page of a non-clustered index is different from the data in the leaf page of a clustered index, no data is returned until the leaf level of the index is reached.

The headers of index pages also contain pointers to the next and previous page at that level. On an index seek for a range of values such as the range of values returned by the query in the example above, the keys at each level of the index are compared to the values in the condition to locate the beginning of the range. The pointer to the correct child page is then followed, and the process is repeated until the leaf level of the index is reached. At this point, a range scan begins.

Values are evaluated at the leaf level of the index. If all the values on a page are read and the end of the range has not yet been reached, then the pointer to the next page is followed; the data on the next leaf page is then read and evaluated to find rows that meet the filter criteria. In this way, the leaf level of the index is scanned from the beginning to the end of the range, much as was done in Tables 1 and 2 to find the rows to be returned for the query.

The common recommendation to always put the most selective column first is based on the idea that you can eliminate more branches in the root and intermediate page levels by evaluating only one value, rather than evaluating the first key and then proceeding to subsequent key values in the index. SQL Server reads at most one page at the root and each intermediate level of the index for each range evaluated. Medium to large indexes may be only three to six levels deep, so placing the most selective column first may help SQL Server eliminate a few comparisons, and thus save a few CPU cycles on three to six pages, in most cases.

However, once the leaf level is reached and scanning begins on the requested range, placing the inequality column first can result in the reading of hundreds or even thousands of extra pages to satisfy the same query. Clearly, the tradeoff at the leaf level can quickly overwhelm the advantages gained at the root and intermediate levels on queries that return ranges of values.

An Example to Demonstrate Tradeoffs with the Choice of First Column

To demonstrate these tradeoffs, start with the following script. You should run the script on a test or development server.

```
USE tempdb
GO
IF (SELECT OBJECT_ID('dbo.testTable') IS NOT NULL)
      DROP TABLE dbo.testTable
GO
CREATE TABLE dbo.testTable
(
      charCol                        CHAR(200)
,     bitCol                         BIT
,     intCol                         INT
)
GO


-- insert some test data into the table:
```

```
SET NOCOUNT ON


DECLARE @val INT
set @val = 0
WHILE @val < 100000
BEGIN
            INSERT INTO dbo.testTable VALUES (@val, 0, @val)
            INSERT INTO dbo.testTable VALUES (@val, 1, @val)
            SET @val = @val + 1
END
```

After the script is run, the populated table is properly set up in tempdb. At this point, the table has no indexes, so any access, with or without a WHERE clause, requires a full table scan.

To see how much time and how many page reads are required for this data access, set STATISTICS TIME and STATISTICS IO to “on” with the following script:

```
SET STATISTICS TIME ON
SET STATISTICS IO ON
```

Establish a Baseline
Execute a count query with a filter specifying an equality condition for the BIT column and an inequality condition for the INT column, similar to that used in the first example. This count query is used with all iterations to show the effect of different indices and options.

```
SELECT COUNT(*)
FROM testTable
WHERE bitCol = 1 AND intCol BETWEEN 100 AND 300
```

After running the count query, check the Messages tab for information about the number of reads and the wall clock CPU time needed for execution. These metrics are used to gauge the effectiveness of the index options.

For example, when the count query was executed in a test, the following critical lines of message output were obtained:

Table 'testTable'. Scan count 1, logical reads 5556, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

```
 SQL Server Execution Times:

   CPU time = 31 ms,  elapsed time = 31 ms.
```

This means SQL Server read 5,556 data pages and used 31 milliseconds (wall clock) of CPU resources to resolve the count query with no indexes on the table; this is the baseline result.

Create a Clustered Index with the More-Selective Column First
Next, create the clustered index with the integer intCol column first. Note that there are 10,000 distinct values in the intCol column and only two values in the bitCol column. If you use selectivity as the criterion, this index is designed as follows:

```
CREATE CLUSTERED INDEX testTable_intCol_bitCol on testTable(intCol, bitCol)
```

Re-run the count query with the same WHERE clause:

```
SELECT COUNT(*)

FROM testTable

WHERE bitCol = 1 AND intCol BETWEEN 100 AND 300
```

When the query was executed in a test, the following critical lines from the messages output were obtained:

```
Table 'testTable'. Scan count 1, logical reads 15, physical reads 0, read-
ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
reads 0.


 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 1 ms.
```

The scan count of 1 indicates only one navigation from root to leaf level of the index was necessary because only one range is scanned in this query. When the table is indexed with the more-selective intCol first in the index, nine pages are read to resolve the query. The process took 1 ms to execute.

Create a Clustered Index with the Equality Column First

Drop the existing index and create a new clustered index with the BIT column (bitCol) first:

```
DROP INDEX testTable_intCol_bitCol on testTable


CREATE CLUSTERED INDEX testTable_bitCol_intCol ON testTable(bitCol, intCol)
```

Run the query again:

```
SELECT COUNT(*)

FROM testTable

WHERE bitCol = 1 AND intCol BETWEEN 100 AND 300
```

When the query was executed in a test, the following critical information was returned in the Messages tab:

```
Table 'testTable'. Scan count 1, logical reads 10, physical reads 0, read-
ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
reads 0.


 SQL Server Execution Times:

   CPU time = 0 ms,  elapsed time = 0 ms.
```

By putting the less-selective bitCol first in the index, you reduce the number of total pages read to resolve this query from 15 to 10. The elapsed time to execute the query is now too small to measure. This improvement occurs because the less-selective column is the equality column in this query.

NOTE: Depending on the computer you use for these tests, the elapsed time may actually be immeasurably small for both queries. The number of page reads is the better indicator of the amount of data that must be read—and thus the amount of work required—to resolve this query.

As you can see from the example, if you are indexing for queries that have both equality and inequality conditions, putting the equality column first is more important than putting the more-selective column first. This can also be demonstrated using the SQL Server index structures.

Does SQL Server Know This?

Customers frequently think that SQL Server does not use an index if the selectivity of the first column is too low. The example above demonstrates that this is not the case, but does SQL Server know that an index with low selectivity on the first column may be the more efficient index?

Test with Two Non-Clustered Indexes

For an answer, drop the clustered index on the table above and create two non-clustered indexes to represent the data. The only difference between the two indexes is which column is first. When queried in this way, SQL Server determines which index it will use for the query.

```
-- Drop the clustered index:

DROP INDEX testTable_bitCol_intCol on testTable
```

```
-- create the two nonclustered indexes


CREATE NONCLUSTERED INDEX testTable_bitCol_intCol
      ON testTable(bitCol, intCol) INCLUDE (charCol)


CREATE NONCLUSTERED INDEX testTable_intCol_bitCol
      ON testTable(intCol, bitCol) INCLUDE (charCol)
```

To produce the estimated execution plan, press **Ctrl-L** in Management Studio or execute the query after selecting **Include Actual Execution Plan**. Hover over **Index Seek** to see which index was selected.

In a test, after both indexes were created, the following estimated execution plan was obtained:

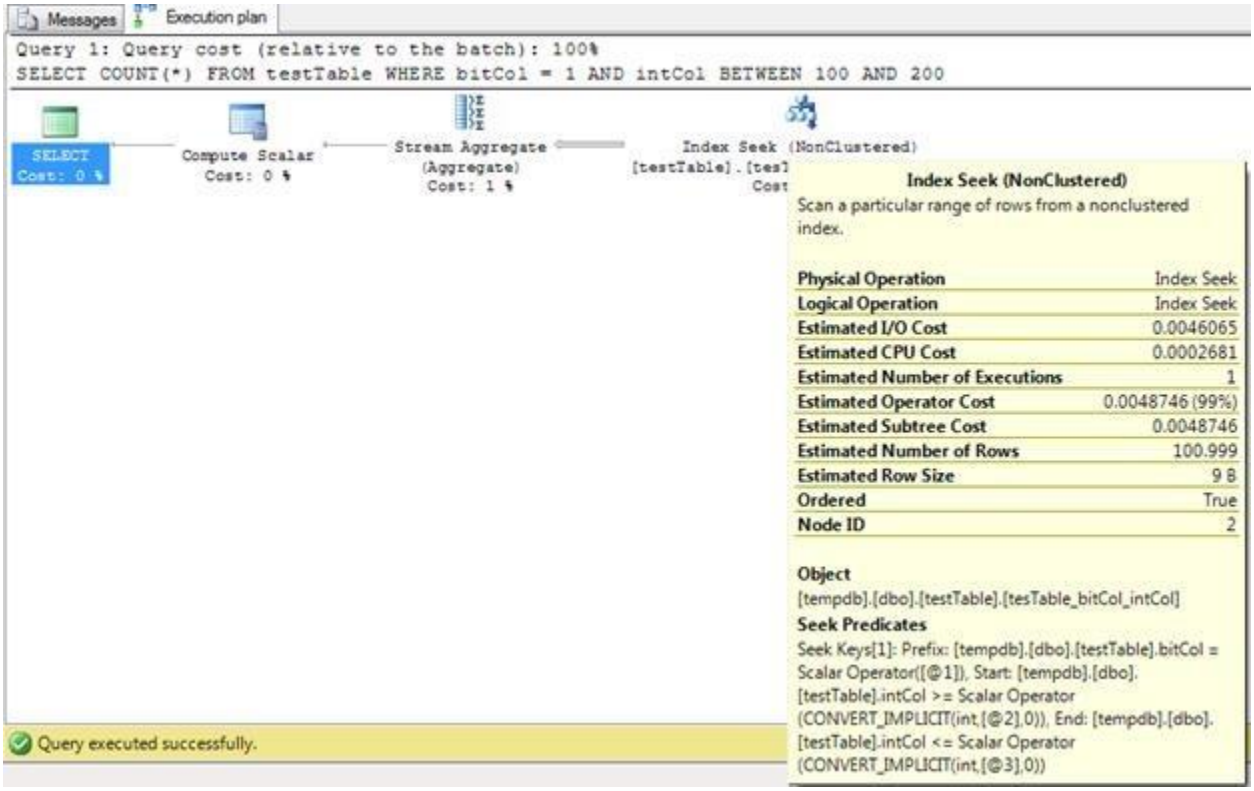


Figure 1. Estimated execution plan showing which index was selected

Below **Object** in the ToolTip, you can see that tempdb.dbo.testTable.tesTable_bitCol_intCol was the index chosen to perform the seek. SQL Server therefore chose the index with the BIT column first. This clearly demonstrates that SQL Server does know that the index with the equality column first is the better choice.

How Does SQL Server Know That the Index Is Useful?

One frequently voiced objection to putting a low-selectivity column first in an index is that statistics are only maintained for the first column. This is true of the histogram, but the histogram is not the only factor evaluated when deciding if an index is useful for a query with multiple filter conditions in the WHERE clause or JOIN clause. Regardless of which column comes first in the index, SQL Server also evaluates statistics on the second column when generating an execution plan. The following example shows the process used by SQL Server.

To start, execute the following script to clear the work that was done previously, including auto-created statistics and reset:

```
SET STATISTICS TIME OFF

SET STATISTICS IO OFF

USE tempdb

GO

IF (SELECT OBJECT_ID('dbo.testTable')) IS NOT NULL

      DROP TABLE dbo.testTable

GO

CREATE TABLE dbo.testTable

(

      charCol                        CHAR(200)

,     bitCol                         BIT

,     intCol                         INT

)

GO


-- insert some test data into the table:

SET NOCOUNT ON


DECLARE @val INT

set @val = 0

WHILE @val < 100000
```

```
BEGIN
            INSERT INTO dbo.testTable VALUES (@val, 0, @val)
            INSERT INTO dbo.testTable VALUES (@val, 1, @val)
            SET @val = @val + 1
END

-- create the two nonclustered indexes

CREATE NONCLUSTERED INDEX testTable_bitCol_intCol
      ON testTable(bitCol, intCol) INCLUDE (charCol)

CREATE NONCLUSTERED INDEX testTable_intCol_bitCol
      ON testTable(intCol, bitCol) INCLUDE (charCol)
```

Execute the Query and Examine the Execution Plan
Next, execute the following query again to force SQL Server to compile an execution plan and decide which index to use:

```
SELECT COUNT(*)
FROM testTable
WHERE bitCol = 1 AND intCol BETWEEN 100 AND 300
```

In a test, the following estimated execution plan was obtained:

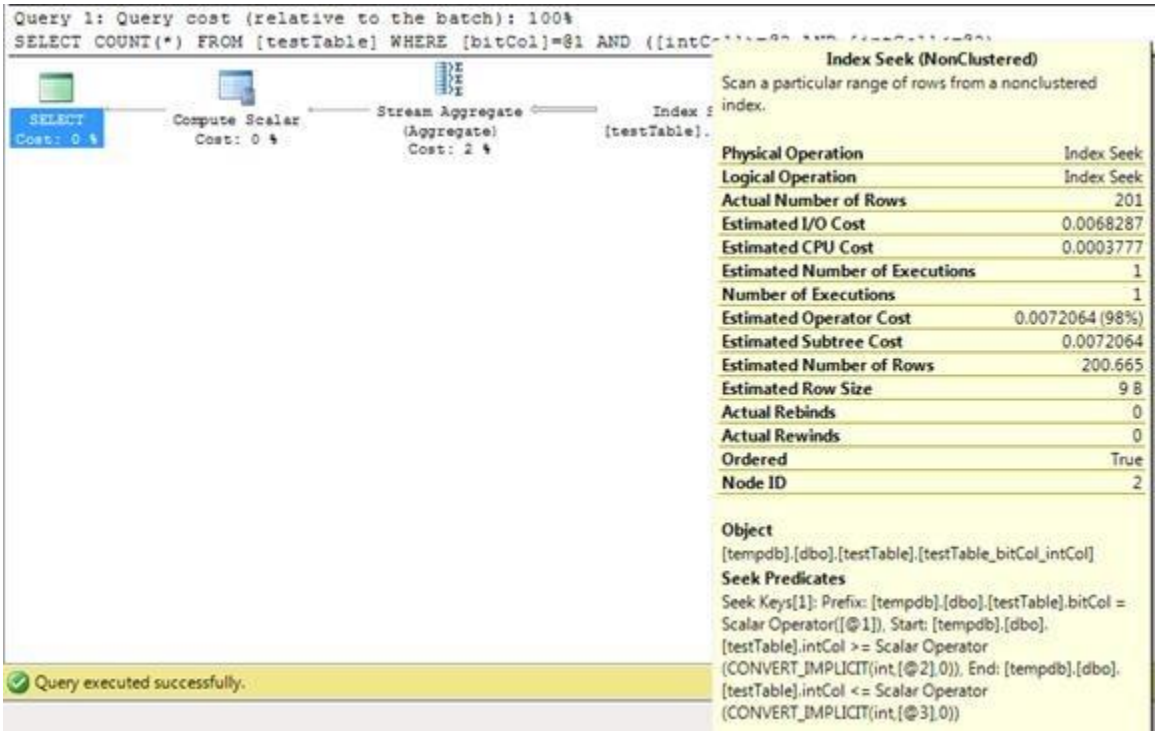


Figure 2. Estimated execution plan with the non-clustered indexes

Note that the index with bitCol as the first column is chosen again. Note also that the estimated number of rows, calculated from the statistics, is very close to the actual number of rows.

Examine the Statistics

Using the following statements, you can see the statistics that exist on the testTable after executing this query:

```
SELECT * FROM sys.stats

WHERE [object_id] = OBJECT_ID('testTable')
```

In the test, the following results were obtained:

Table 3. Statistics obtained from running the example query

object_id	name	stats_id	auto_created	user_created	no_recompute	has_filter	filter_definition
453576654	testTable_bitCol_intCol	2	0	0	0	0	NULL
453576654	testTable_intCol_bitCol	3	0	0	0	0	NULL

No automatically generated statistics have yet been created, but SQL Server was able to estimate the cardinality for the seek operation above very accurately.

Look next at the statistics for testTable_bitCol_intCol by executing the following:

```
DBCC SHOW_STATISTICS('testTable', 'testTable_bitCol_intCol')
```

(See DBCC SHOW_STATISTICS (Transact-SQL) for details of the results sets returned by DBCC SHOW_STATISTICS.)

Note that densities are calculated and stored for bitCol, the combination of bitCol, intCol. However, the histogram is calculated using only the first column of the index. There are only two possible values for a non-null BIT column, so only two rows exist for this entry, as follows:

Table 4. Statistics for testTable_bitCol_intCol

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
0	0	100000	0	1
1	0	100000	0	1

With statistics showing that 100,000 values exist for 0 and 100,000 values exist for 1, how does SQL Server come to an estimate of 200 rows for a seek with the predicate above?

Drop One Index and Run the Query Again

The question can be best answered by dropping one of the indexes and running the query again. Execute the following query:

```
-- drop the index that is not being used

DROP INDEX testTable_intCol_bitCol ON testTable

Then execute the query again capturing the actual execution plan:

SELECT COUNT(*)

FROM testTable

WHERE bitCol = 1 AND intCol BETWEEN 100 AND 300
```

The test resulted in the following estimated execution plan:

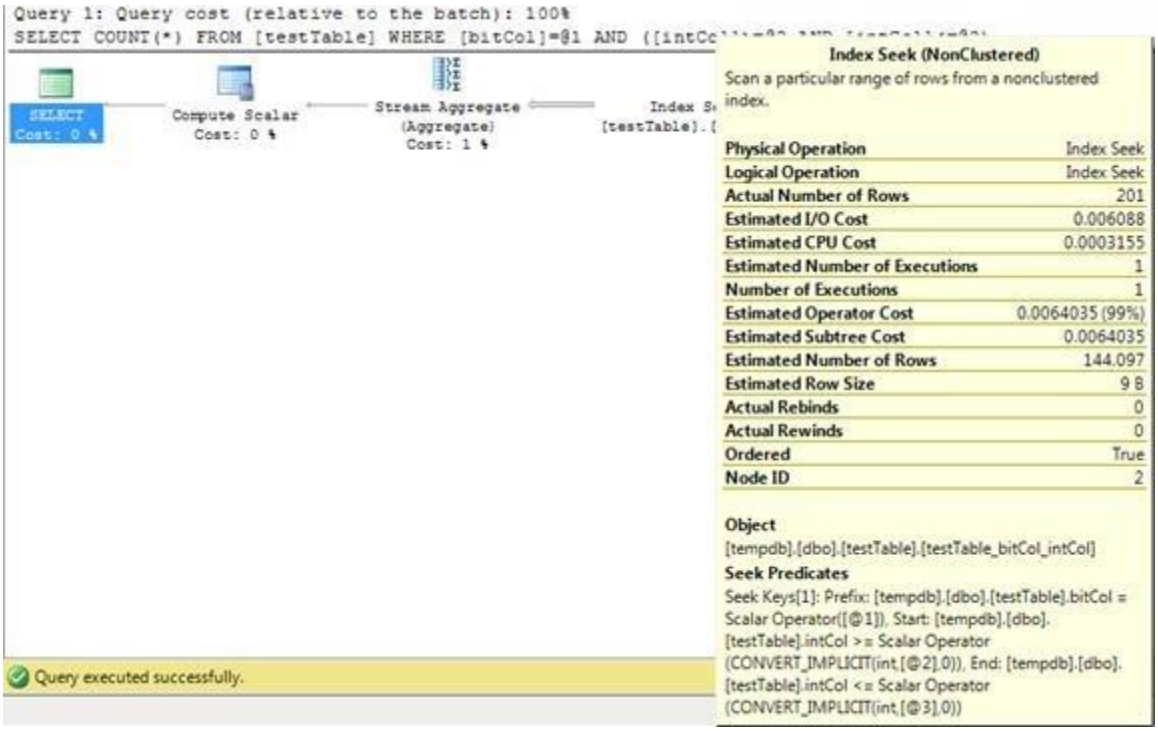


Figure 3. Estimated execution plan after one index dropped

Notice the same index was used on the same data, but the estimated number of rows has changed. This indicates that something has changed with the statistics that are used to generate the query plan, even though the index that was used for a seek was not changed.

Determine What Changed

To see what changed in the statistics, execute the following query again:

```
SELECT * FROM sys.stats

WHERE [object_id] = OBJECT_ID('testTable')
```

You should get the following results:

object_id	name	stats_id	auto_created	user_created	no_recompute	has_filter	filter_definition
453576654	testTable_bitCol_intCol	2	0	0	0	0	NULL
453576654	_WA_Sys_00000003_1B0907CE	3	1	0	0	0	NULL

In the first pass, the testTable_intCol_bitCol index was in place, and that index had statistics on intCol. When that index was dropped, no statistics existed for intCol; for SQL Server to estimate the cardinality

that included results from intCol, therefore, it had to create statistics for that column. This is only possible because AUTO_CREATE_STATISTICS was enabled on the database where this table exists.

The difference in the estimates between using the statistics on testTable_intCol_bitCol and using auto-created statistics is sampling; when testTable_intCol-bitCol was created, statistics were created with FULLSCAN, but auto-creation and auto-updating of statistics always use sampling.

To verify this, run the following on your test server:

```
-- update the statistics with fullscan
-- this will ensure the statistics created by
-- auto-create stats is updated with the same
-- sampling as the index was when it was created
UPDATE STATISTICS testTable WITH FULLSCAN


-- get rid of all query plans in cache to
-- force SQL to compile a new query plan next time
-- the query is esecuted
DBCC FREEPROCCACHE
```

Now, execute the SELECT statement again, capturing the actual execution plan. In the test, the result was the following execution plan:

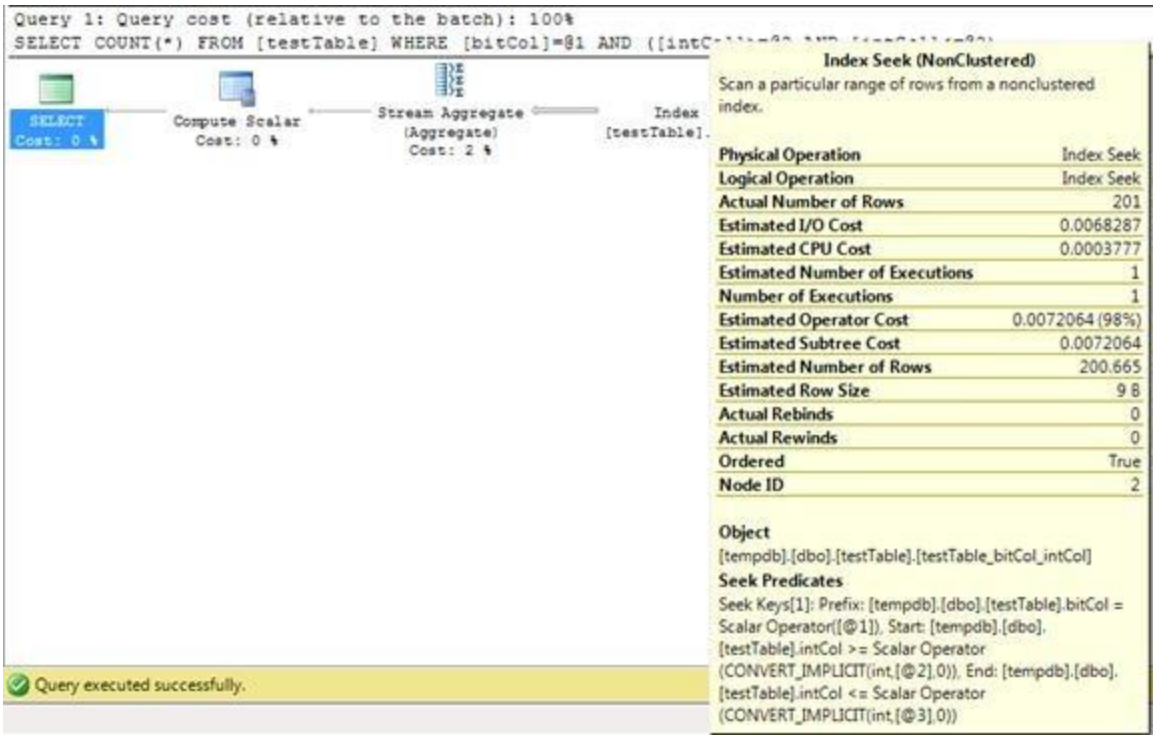


Figure 4. Execution plan after running FULLSCAN

Since the new auto-created column statistics are updated with FULLSCAN this time, SQL Server has the same quality of statistics on the second column as it had when testTable_intCol_bitCol existed. The estimate of rows to be returned from the seek is once again almost exactly the same as the actual result.

In Summary
Regardless of which column comes first in the index, SQL Server needs to evaluate statistics on both columns in the filter condition when determining the usefulness of an index—the statistics on the first column alone are not sufficient. To find statistics on the second column when a query plan is generated, SQL Server first looks to see if an index exists that has that column as its first column. If such an index exists, SQL Server uses the statistics on that index. If such an index does not exist and no statistics have been created on the second column and if AUTO_CREATE_STATISTICS is enabled on that database, SQL Server creates statistics on the second column and use these auto-created statistics when generating a query plan.

Recommendations

Placing a low-selectivity column first in an index can cause problems in the same scenarios in which placing any other column first in an index causes problems.

Index for the Workload as a Whole

Indexes should be designed for your overall workload rather than for any one particular query. So how many indexes should you have on your table? There is really is no single answer, but if you create the index on (bitCol, intCol), then the index is used for seeks for queries with WHERE clauses such as:

```
WHERE bitCol = 1 AND intCol BETWEEN 100 AND 200
```

Or WHERE clauses such as:

```
WHERE bitCol = 1 AND intCol = 150
```

But would not be useful for queries with WHERE clauses such as:

```
WHERE intCol BETWEEN 100 AND 200
```

In other words, the index is useful for a seek only if the first column of the index appears as a limiting condition in the query. If you have a mix of both types of queries in your workload, then you need to evaluate whether it is worthwhile to maintain both indexes. You must decide if having both indexes in place provides enough additional benefit to "pay" for the extra work needed to maintain the extra index on data modifications.

Create and Maintain Statistics

Be sure SQL Server has the statistics available or has the ability to create the statistics needed for generation of optimal query plans. Cost-based optimization depends on statistics. Use AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS where possible. Even if auto-created or auto-updated statistics are not used, SQL Server must have updated statistics created with a sufficient sampling to generate optimal query execution plans.

Conclusion

Understanding the nature of the queries being executed is essential to choosing proper indexes. Examining the cardinality or selectivity of a column alone is not sufficient to choose an optimal indexing strategy.

When choosing indexes to optimize queries with both equality and inequality conditions, equality columns should be placed first—even if the equality column is not as selective as the inequality column. This is true even if the equality column is a BIT column. Note that placing the inequality column first in an index can never be more efficient than placing the equality column first.

When you next encounter an execution plan with a missing index suggestion that has a BIT column first, consider it carefully against the query that generated it. SQL Server is probably right about the BIT column being the best choice for the first column.

Top Tips for Maximizing the Performance & Scalability of Dynamics AX 2009 systems on SQL Server 2008

Authors: Peter Scharlock, Mark Prazak
Reviewers: Kevin Cox, Mike Ruthruff, Norberto Garcia, Sri Srinivasan

Introduction:

This article summarizes the top tips for maximizing the performance and scalability of the Microsoft Dynamics AX 2009 application when utilizing SQL Server 2008. These tips are the result of collaboration between the Dynamics AX Performance Team & the SQL Server Partner Advisory Team based on optimizing real customer workloads and extensive benchmarking efforts. The tips encompass recommended best practices, as well as exposing newly released or modified, Dynamics and SQL Server code.

1. Maximize Cursor Performance

The Dynamics AX application uses the well-known SQL Native Client ODBC API for its interaction with the SQL Server 2008 database. The AX programming model calls API Server Cursors extensively for paging through data, and in most cases this provides great performance, scalability, and reliability. However, during performance testing we have uncovered numerous areas for optimizing cursor performance. In most cases, the AOS servers utilize the fastest and most optimal SQL Server cursor type: Fast Forward-Only with the Autofetch (FFO) option. See documentation here: http://msdn.microsoft.com/en-us/library/ms187502.aspx

As stated in the SQL 2008 Books Online topic listed above, Fast Forward-only cursors are never converted, however it is possible that under certain conditions a 'static-type' execution plan may be created nonetheless, which essentially disables the optimal behavior. These conditions include; the resultset includes one or more 'blobs', and/or not having the appropriate indexes to optimally satisfy the cursor.

Maximize Auto-Fetch capability by paying attention to:

Table design considerations

Limit the use of AX 2009 specific 'container' and 'memo' data-types (which utilize the SQL Server 'image' and 'text' data-types). SQL Server will degrade to a less optimal cursor type/behavior if the query contains either of these data-types. Retrieving these data-types will create a much more expensive static query plan. If you do need to add / retrieve these data-types try to make sure that these queries retrieve a single row or small number of rows, this will limit the time that it takes to create the static dataset in tempdb.

Keep the row length small to maximize the number of rows that get returned by the Auto-Fetch cursor

SQL Cumulative Updates (CU)

Joint scalability testing between the SQL and AX performance engineering teams revealed a SQL issue that prevented SQL Server from using prefetch / read-ahead logic under certain cases even when the optimal cursor type was generated by SQL Server. This issue has been fixed (in SQL 2005 and SQL 2008) and documented in the following Knowledge Base article: http://support.microsoft.com/kb/973877

Evaluate whether this CU is a requirement for your implementation. As per SQL Server sustained engineering policy, this change will be automatically rolled into the next available SQL Server Service Pack.

AOS Configuration

The 'Maximum Buffer Size' setting can be adjusted to increase the number of rows retrieved in a single fetch operation. The default setting is 24 (KB). You can increase this setting but do so in relatively small increments, such as 8K at a time, and monitor AOS memory consumption; make certain there is not a substantial increase in memory for the AX32Serv.exe process. After increasing this setting, you should monitor SQL Server:Batch Requests/Sec to determine if there has been a material decrease in round trips to SQL Server.

Pessimistic Concurrency Considerations

Currently, X++ SELECT statements issued under AX's pessimistic locking will request a **dynamic cursor** and not a FFO cursor, so we recommend: optimistic locking for this reason in addition to the concurrency benefits it provides.

There is a pending design change in AX4, AX2009 and the future AX6 that will request FFO cursors even when pessimistic locking is used. An announcement of this change will be made on the AX Performance blog: http://blogs.msdn.com/axperf when completed.

2. Index Management

Like most other SQL Server applications, Dynamics AX takes full advantage of indexes. However, there are a number of very important considerations when adding and/or modifying indexes on an AX database;

Indexes must be created/modified using the Application Object Tree (AOT) function within AX

Once the AX object model is aware of the new Index or Index modification, an AX synchronization process must be run; this process creates/modifies the actual SQL Server Index(es). NOTE: Index changes made directly against the SQL Server database run the risk of being dropped during a subsequent synchronization run.

Observe basic and accepted practices regarding clustered indexes:

All tables should have a clustered index.

Indexes that maximize the benefit of clustering are those which are frequently used to return a range of rows.

The clustered index key should not include columns which are updated.

Consider the key length when choosing a clustered index. A long clustered key can inflate the size of non-clustered indexes on the same table. This is because every non-clustered index 'embeds' the clustering key as the mechanism to find the path back to the actual data row.

Unique Indexes

Ensure that any index which is known to be unique is defined as such in the AOT (as primary, or AllowDuplicates:No). Unique indexes serve as cache lookup keys for AX record-level caching. NOTE: in AX4 only the primary key is a cache lookup key.

Index Fragmentation considerations

Many indexes defined in the AX schema do not cause any performance degradation even if fragmentation levels are very high. As a rule of thumb, we can postpone rebuilding AX indexes that are used exclusively for locating a single row.

Examples of such indexes are:

RecId index on any table

TransIdIdx on SalesLine table

We can use the following criteria for identifying such indexes:

Are unique, due to AllowDuplicates::No or are designated as the primary key.

Are generally composed of two key columns, one of them being DataAreaId.

The following query can be used to locate indexes meeting these criteria:

```
select OBJECT_NAME(id), name from

sysindexes

where indexproperty(id, name, 'IsUnique') = 1 and keycnt = 2   and INDEX_COL(OBJECT_NAME(id), indid, 1) =

'DATAAREAID'
```

Fragmentation is not the only reason for rebuilding indexes; The DBA should also consider rebuilding indexes whenever large amounts of data have been removed from the AX database. Doing so will reclaim space and make it available for reuse.

Consider rebuilding indexes for tables affected by the following processes:

The AX default cleanup processes are run. These processes are module specific and can generally be located under Periodic->Cleanup.

Intelligent Data Management Framework (IDMF) is used to purge or archive data.

Missing Indexes

SQL Server has the ability to identify missing indexes by way of a number Dynamic Management Objects. These objects can be monitored to detect cases where the Query optimizer believes that adding an index would help performance. These DMO's and their use are documented here: http://msdn.microsoft.com/en-us/library/ms345524(v=SQL.100).aspx

NOTE: whenever an index is added to the AX schema, it should be carefully tested by the DBA before being put into the production environment.

3. SQL Server statistics

The SQL Server query optimizer is tasked with creating optimal query plans based on a defined set of heuristics. A major piece of these heuristics is the SQL Server statistics; they help the optimizer by supplying details about the data distribution, last time stats were updated, and so on.More about SQL Statistics found in the following Books-Online topic here: http://msdn.microsoft.com/en-us/library/ms190397(v=SQL.100).aspx For the SQL query optimizer to do its job effectively the statistics must be kept up-to-date; the following tips are recommended guidelines for AX.

Enable Auto-Update statistics

Enabling auto-updates statistics allows the SQL Server Engine to automatically update statistics based on specific threshold values. Follow guidance provided in the Books-Online topic listed above.

Enable Auto-Create statistics

Enabling auto-create statistics allows the SQL Server Engine to automatically create new statistics on columns as necessary. Follow guidance provided in the Books-Online topic listed above.

There is a startup cost associated with the creation of new statistics due to the Auto-Create statistics setting. This cost can result in high CPU on a new AX deployment due to the large number of composite indexes that exist in the AX database. The additional CPU consumption will subside after a short time (a few hours) of a steady AX user workload.

You can preempt much of this statistics startup cost through the use of the system stored procedure sp_createstats. You can use sp_createstats to initially create statistics on all columns participating in an index:

```
sp_createstats @indexonly = 'indexonly'
```

Auto-update async should be left at its default OFF setting. It was previously advised to enable Auto-update async but this recommendation has now been retracted. We have identified several processes, such as Update Inventory on Hand and General Ledger Post, which have the potential to degrade if Auto-update async is enabled. Also, ongoing performance testing did not show a material benefit to any AX process if Auto-update async is enabled.

4. Parameterized Query considerations

Before a query, batch, stored procedure, trigger, prepared statement, or dynamic SQL statement begins execution on an instance of Microsoft SQL Server, it is compiled into a query plan by the SQL Server Database Engine query optimizer. Then the plan is executed to produce a result-set. The compiled query plans are stored into a part of SQL Server memory that is called the plan cache. Reusing complied query plans offer significant performance benefits because they save resources by not having to compile every time the query is executed. One way to further enhance this benefit is by using Parameterized queries, that is; Queries are compiled with parameters rather than literal values. By doing this the query can be compiled for a single parameter value, and yet be reused for many other parameter values.

By default, Dynamics AX parameterizes all queries to take advantage of these performance enhancements but there are a number of conditions where parameterized queries may not be the best choice. For example, to be effective, parameterized queries work best when;

The distribution of the data being retrieved is evenly distributed

The query result-sets contain approximately the same number of rows

The set of used parameter values would typically generate the same query plan

As you can see, there are numerous conditions where parameterized queries may not be the best choice for a specific AX implementation. These conditions are generally related to a phenomenon known as parameter sniffing. Details about parameter sniffing can be found in the following blog:
http://blogs.msdn.com/queryoptteam/archive/2006/03/31/565991.aspx

Basically, when creating a query plan, the SQL Server optimizer uses the parameter values passed the 'first time' to create the cached execution plan for this query and this plan will be utilized for all subsequent calls to this query including parameter values that might have prompted a different query execution plan. For example; let's suppose we have a simple parameterized query that does the following: select * from table1 where country = @Param1. Now imagine that the query is compiled and cached where the first invocation passes the value 'USA' for the parameter @Param1. Now let's suppose the second invocation uses the value 'Norway' as the parameter value. Using the points above as a guide, what could be the result?

The distribution of the data being retrieved is evenly distributed.

In this case, everything would work fine, however if either value was highly skewed, that is; many more of one value than the other, then we have typically seen that an optimal plan will be created for the first passed value, but this plan may not be the best choice for the 'next value'.

The query result-sets contain approximately the same number of rows.

This is very similar to the description above

The set of used parameter values would typically generate the same query plan.

Again, this is very similar to the description above

Because these are data conditions generally out of the control of the AX application, there are a number of ways to successful address these conditions:

AX Server configuration

The AX Server Configuration Utility has two configuration settings pertaining to query parameterization. These are located on the utility's Database Tuning tab under Auto Generation Options:

Use literals in join queries from forms and reports

Use literals in complex joins from X++

Checking either of these settings will suppress parameterization of queries described in the respective setting label. The default for both settings is OFF and it is not recommended to change either one.

X++ ForcedLiterals

ForceLiterals is a find option that can be used with SELECT statements issued through X++. When present on a SELECT statement ForceLiterals will suppress parameterization and literal values will appear in the WHERE clause. The effect of ForceLiterals is that the SQL statement will recompile each time it is executed.
To accomplish the same effect for a form's datasource, we include the following in the datasource's init method:

```
this.query().literals(true);
```

In both cases where we specify literals to be used, we must exercise caution that doing so does not introduce the security threat of SQL injection; the literal values should not be derived from a user accessible field on a form. We must also consider the cost of repeated compilation when forcing literals in place of parameterization.

NOTE: the XML representation of a query plan (captured using DMO's or by saving an execution plan in XML): http://msdn.microsoft.com/en-us/library/ms190646.aspx

identifies the parameter values that were used to compile the query, as well as the current runtime parameter value. Comparing these values can be highly valuable in diagnosing parameter sniffing issues.

Plan Guides force specific behavior for certain 'problematic' queries. Books-online documentation:

http://technet.microsoft.com/en-us/library/ms190417.aspx

Use of plan guides to pass the following:

OPTION(RECOMPILE)

OPTION(OPTIMIZE FOR…)

OPTION(OPTIMIZE FOR UNKNOWN)

NOTE: using this method against the AX 2009 derived queries is complex because of the extensive use of API cursors. This method should be used in moderation.

SQL Server trace flag: 4136

This new trace flag effectively disables the parameter sniffing process at a global SQL Server level, while preserving the benefits of plan cache reuse.
The following KB Article provides more information about how to obtain the Cumulative Update to enable the traceflag, as well as details under which conditions the traceflag is ignored. KB:
http://support.microsoft.com/kb/980653

Evaluate whether *your AX 2009 implementation* will benefit from this trace flag.

NOTE: this traceflag is to be used where parameter sniffing is problematic because of skewed data or other reasons pointed out in the following blog: http://blogs.msdn.com/queryoptteam/archive/2006/03/31/565991.aspx it should be used only under fully tested conditions that have proven that the traceflag is beneficial for your specific workload.

5. Concurrency considerations

The AX application supports many (hundreds or thousands of) users accessing the database at the same time. When some of these users attempt to modify the same data in a database at the same time, a system of controls must be implemented so that modifications made by one person do not adversely affect those of another person. This is called concurrency control. There are a number of important considerations to achieve optimal concurrency control when using the AX application:

Read Committed Snapshot Isolation (RCSI): http://technet.microsoft.com/en-us/library/ms177404.aspx

RCSI is the recommended setting for the AX database. This concurrency model allows 'writers to block readers' but also 'readers do not block writes'. This model provides for excellent performance while also ensuring that that database operations maintain transactional consistency. NOTE: this is not the default SQL Server setting!
When upgrading to Microsoft Dynamics AX 2009, some AX upgrade scripts require RCSI to be disabled during the upgrade process. If RCSI is turned off for the upgrade it must be re-enabled before the actual OLTP workload is restarted. RCSI should be enabled during the entire Microsoft Dynamics AX 2012 upgrade process in order to prevent additional blocking during pre-processing and delta-processing script execution.

Lock Escalation

Lock escalation is a SQL Server mechanism that converts many fine-grain locks into fewer coarse-grain locks. This reduces system overhead; however it also increases the probability of concurrency contention. This trade-off has often caused blocking and sometimes even deadlocking in the AX application. Previous versions of SQL Server permitted controlling lock escalation (trace flags 1211 and 1224), but this was only possible at an instance-level granularity.
SQL Server 2008 offers a better solution. A new ALTER TABLE SET (LOCK_ESCALATION = { AUTO | TABLE | DISABLE }) option has been introduced to control lock escalation at a table level. Now, lock escalation can be disabled at a table by table basis. Details can be found here: http://msdn.microsoft.com/en-us/library/ms190273.aspx

NOTE: Lock escalation can be monitored by a trace event as described in the following article: http://msdn.microsoft.com/en-us/library/ms190723.aspx

Pessimistic locking on custom tables

SQL Server has a pessimistic concurrency mode: Snapshot Isolation; however the AX application team implements its own pessimistic concurrency model to more easily satisfy the AX object model requirements. It is important to follow the specific AX pessimistic implementation guidelines when you have created your own custom AX tables: http://msdn.microsoft.com/en-us/library/bb190073(AX.10).aspx

Conclusions:

Utilizing standard SQL Server 2008 and AX 2009 best practices generally provide good performance. This article goes into much more depth; pointing out very specific functionality that will allow the AX 2009 DBA to maximize the performance, scalability, and reliability on their implementation on SQL Server 2008. Although this article is meant for current/future AX customers, much of the information presented below is applicable to other ISV applications as well.

Top SQL Server 2005 Performance Issues for OLTP Applications

OLTP work loads are characterized by high volumes of similar small transactions.

It is important to keep these characteristics in mind as we examine the significance of database design, resource utilization and system performance. The top performance bottlenecks or gotchas for OLTP applications are outlined below.

Database Design issue if….

- Too many table joins for frequent queries. Overuse of joins in an OLTP application results in longer running queries & wasted system resources. Generally, frequent operations requiring 5 or more table joins should be avoided by redesigning the database.
- Too many indexes on frequently updated (inclusive of inserts, updates and deletes) tables incur extra index maintenance overhead. Generally, OLTP database designs should keep the number of indexes to a *functional minimum*, again due to the high volumes of similar transactions combined with the cost of index maintenance.
- Big IOs such as table and range scans due to missing indexes. By definition, OLTP transactions should not require big IOs and should be examined.
- Unused indexes incur the cost of index maintenance for inserts, updates, and deletes without benefiting any users. Unused indexes should be eliminated. Any index that has been used (by select, update or delete operations) will appear in sys.dm_db_index_usage_stats. Thus, any defined index not included in this DMV has not been used since the last re-start of SQL Server.

CPU bottleneck if…

- Signal waits > 25% of total waits. See sys.dm_os_wait_stats for Signal waits and Total waits. Signal waits measure the time spent in the runnable queue waiting for CPU. High signal waits indicate a CPU bottleneck.
- Plan re-use < 90% . A query plan is used to execute a query. Plan re-use is desirable for OLTP workloads because re-creating the same plan (for similar or identical transactions) is a waste of CPU resources. Compare SQL Server SQL Statistics: batch requests/sec to SQL compilations/sec. Compute plan re-use as follows: Plan re-use = (Batch requests - SQL compilations) / Batch requests. Special exception to the plan re-use rule: Zero cost plans will not be cached (not re-used) in SQL 2005 SP2. Applications that use zero cost plans will have a lower plan re-use but this is not a performance issue.
- Parallel wait type cxpacket > 10% of total waits. Parallelism sacrifices CPU resources for speed of execution. Given the high volumes of OLTP, parallel queries usually reduce OLTP throughput and should be avoided. See sys.dm_os_wait_stats for wait statistics.

Memory bottleneck if…

- Consistently low average page life expectancy. See Average Page Life Expectancy Counter which is in the Perfmon object SQL Server Buffer Manager (this represents is the average number of seconds a page stays in cache). For OLTP, an average page life expectancy of 300 is 5 minutes. Anything less could indicate memory pressure, missing indexes, or a cache flush.

- Sudden big drop in page life expectancy. OLTP applications (e.g. small transactions) should have a steady (or slowly increasing) page life expectancy. See Perfmon object SQL Server Buffer Manager.
- Pending memory grants. See counter Memory Grants Pending, in the Perfmon object SQL Server Memory Manager. Small OLTP transactions should not require a large memory grant.
- Sudden drops or consistenty low SQL Cache hit ratio. OLTP applications (e.g. small transactions) should have a high cache hit ratio. Since OLTP transactions are small, there should not be (1) big drops in SQL Cache hit rates or (2) consistently low cache hit rates < 90%. Drops or low cache hit may indicate memory pressure or missing indexes.

IO bottleneck if…

- High average disk seconds per read. When the IO subsystem is queued, disk seconds per read increases. See Perfmon Logical or Physical disk (disk seconds/read counter). Normally it takes 4-8ms to complete a read when there is no IO pressure. When the IO subsystem is under pressure due to high IO requests, the average time to complete a read increases, showing the effect of disk queues. Periodic higher values for disk seconds/read may be acceptable for many applications. For high performance OLTP applications, sophisticated SAN subsystems provide greater IO scalability and resiliency in handling spikes of IO activity. Sustained high values for disk seconds/read (>15ms) does indicate a disk bottleneck.
- High average disk seconds per write. See Perfmon Logical or Physical disk. The throughput for high volume OLTP applications is dependent on fast sequential transaction log writes. A transaction log write can be as fast as 1ms (or less) for high performance SAN environments. For many applications, a periodic spike in average disk seconds per write is acceptable considering the high cost of sophisticated SAN subsystems. However, sustained high values for average disk seconds/write is a reliable indicator of a disk bottleneck.
- Big IOs such as table and range scans due to missing indexes.

Top wait statistics in sys.dm_os_wait_stats are related to IO such as ASYNCH_IO_COMPLETION, IO_COMPLETION, LOGMGR, WRITELOG, or PAGEIOLATCH_x.

Blocking bottleneck if…

- Index contention. Look for lock and latch waits in sys.dm_db_index_operational_stats. Compare with lock and latch requests.
- High average row lock or latch waits. The average row lock or latch waits are computed by dividing lock and latch wait milliseconds (ms) by lock and latch waits. The average lock wait ms computed from sys.dm_db_index_operational_stats represents the average time for each block.
- Block process report shows long blocks. See sp_configure “blocked process threshold” and Profiler “Blocked process Report” under the Errors and Warnings event.
- Top wait statistics are LCK_x. See sys.dm_os_wait_stats.
- High number of deadlocks. See Profiler “Graphical Deadlock” under Locks event to identify the statements involved in the deadlock.

Network bottleneck if…

- High network latency coupled with an application that incurs many round trips to the database.
- Network bandwidth is used up. See counters packets/sec and current bandwidth counters in the network interface object of Performance Monitor. For *TCP/IP frames actual bandwidth is computed as packets/sec * 1500 * 8 /1000000 Mbps.*

Table-Valued Functions and tempdb Contention

Overview

tempdb contention can be caused by using multi-statement table-valued functions (TVFs) in certain parts of queries, such as the WHERE clause of a query or the column list of a SELECT query. With a multi-statement TVF, a table variable is created and dropped on each call to the TVF, leading to potentially thousands of table variable creations, allocations, and deallocations for a single query. Simultaneous execution of these queries that have multi-statement TVFs can create contention, and this contention can negatively affect the performance of SQL Server even if best practices for **tempdb** configuration are applied.

This technical note explores the detection, cause, and elimination of **tempdb** contention caused by using multi-statement TVFs in certain parts of queries. The paper uses examples to demonstrate several options for reducing contention and provides extensive links for more detailed information.

NOTE: This technical note is intended for experienced database administrators (DBAs) and software developers who have an understanding of Microsoft SQL Server fundamentals.

Detecting tempdb Contention

When troubleshooting a problem with SQL Server performance, the first question to ask is: "What is SQL Server waiting for?" For the answer, you can use the Dynamic Management View (DMV) sys.dm_exec_requests.

Querying the sys.dm_exec_requests DMV returns information about each request that is executing within SQL Server. The query shows you information about currently executing user processes that are waiting: what each process is currently doing and what each process is waiting for.

The following code shows such a query.

```
SELECT

      r.session_id

      ,r.status

      ,r.command

      ,r.database_id

      ,r.blocking_session_id

      ,r.wait_type

      ,AVG(r.wait_time) AS [WaitTime]

      ,r.wait_resource
```

```
FROM
	sys.dm_exec_requests AS r
	INNER JOIN
	sys.dm_exec_sessions AS s ON(r.session_id = s.session_id)
WHERE
	r.wait_type IS NOT NULL
	AND s.is_user_process = 1
GROUP BY GROUPING SETS((
	r.session_id
	,r.status
	,r.command
	,r.database_id
	,r.blocking_session_id
	,r.wait_type
	,r.wait_time
	,r.wait_resource), ())
```

NOTE: You may also find it helpful to query sys.dm_os_wait_stats for evidence of PAGELATCH waits. While the sys.dm_os_wait_stats DMV does not link PAGELATCH waits to a specific session or wait resource, high levels of PAGELATCH waits may indicate that further investigation into allocation contention is needed.

For example, running the query on a poorly performing SQL Server may return the following result set.

Table 1. First result set showing contention on tempdb

session_id	status	command	database_id	blocking_session_id	wait_type	WaitTime	wait_resource
109	suspended	INSERT	5	600	PAGELATCH_UP	29	2:1:1
118	suspended	INSERT	5	600	PAGELATCH_UP	24	2:1:1
120	suspended	INSERT	5	600	PAGELATCH_UP	16	2:1:1
150	suspended	INSERT	5	600	PAGELATCH_UP	24	2:1:1
165	suspended	INSERT	5	600	PAGELATCH_UP	24	2:1:1
175	suspended	INSERT	5	600	PAGELATCH_UP	1	2:1:1
206	suspended	INSERT	5	600	PAGELATCH_UP	6	2:1:1

210	suspended	INSERT	5	600	PAGELATCH_UP	5	2:1:1
242	suspended	INSERT	5	600	PAGELATCH_UP	30	2:1:1
247	suspended	INSERT	5	600	PAGELATCH_UP	15	2:1:1
259	suspended	INSERT	5	600	PAGELATCH_UP	7	2:1:1
260	suspended	INSERT	5	600	PAGELATCH_UP	7	2:1:1
305	suspended	INSERT	5	600	PAGELATCH_UP	29	2:1:1
358	suspended	INSERT	5	600	PAGELATCH_SH	1	2:1:1
361	suspended	INSERT	5	600	PAGELATCH_UP	24	2:1:1
370	suspended	INSERT	5	600	PAGELATCH_UP	17	2:1:1
401	suspended	SELECT	5	600	PAGELATCH_UP	24	2:1:1
419	suspended	INSERT	5	600	PAGELATCH_UP	17	2:1:1
450	suspended	INSERT	5	600	PAGELATCH_UP	16	2:1:1
482	suspended	INSERT	5	600	PAGELATCH_UP	21	2:1:1
492	suspended	INSERT	5	600	PAGELATCH_UP	26	2:1:1
510	suspended	INSERT	5	600	PAGELATCH_UP	10	2:1:1
518	suspended	INSERT	5	600	PAGELATCH_UP	29	2:1:1
540	suspended	INSERT	5	600	PAGELATCH_UP	24	2:1:1
561	suspended	INSERT	5	600	PAGELATCH_UP	21	2:1:1
562	suspended	INSERT	5	600	PAGELATCH_UP	29	2:1:1
607	suspended	INSERT	5	600	PAGELATCH_UP	29	2:1:1
609	suspended	INSERT	5	600	PAGELATCH_UP	22	2:1:1
617	suspended	INSERT	5	600	PAGELATCH_UP	9	2:1:1
656	suspended	INSERT	5	600	PAGELATCH_UP	7	2:1:1
662	suspended	INSERT	5	600	PAGELATCH_UP	17	2:1:1
665	suspended	INSERT	5	600	PAGELATCH_UP	25	2:1:1
667	suspended	INSERT	5	600	PAGELATCH_UP	11	2:1:1
NULL	NULL	NULL	NULL	NULL	NULL	18	NULL

NOTE: When using grouping sets with one empty set of parentheses as in the preceding example, one row has NULLs for all columns except the aggregated column. The aggregated column in this one row is an aggregate of the aggregations. In the example above, the WaitTime column is averaged for each process, and the grouped row is an average of the averages.

In the earlier example, many processes are blocked by session_id 600, but because none of these queries are waiting for locks, this is not considered lock blocking. The wait_type shows that the suspended queries are waiting for PAGELATCH_UP and shows that the suspended queries have been waiting for an average of 18 milliseconds (ms). While 18 ms is a relatively short period of time, the large number of processes being blocked by a single resource means that cumulative wait degrades system performance and leads to long waits for the query response for users.

The database_id in the earlier example is 5, but this is not the location of the resource for which the processes are waiting. To find the location of the bottleneck, see the wait_resource column. In the earlier example, every process is waiting for a resource identified as 2:1:1. For a PAGELATCH, this resource description follows the format **DatabaseID:fileID:pageID**.

A databaseID of 2 is always **tempdb**. A fileID of 1 is always the primary data file for a database (fileID 2 is the transaction log). These results therefore show that there is contention on pageID 1 of the primary data file on **tempdb**.

From the results in Table 1 and recognizing that the contention is on fileID 1, an experienced DBA may decide that what is needed is one equally sized data file for each core, as recommended in Storage Top 10 Best Practices (http://technet.microsoft.com/en-us/library/cc966534.aspx). However, a query for **tempdb** files on this 8-core computer shows that it already has 8 equally sized **tempdb** data files, as shown in Table 2.

```
SELECT DB_NAME(database_id) as database_name, physical_name, size

FROM sys.master_files

WHERE DB_NAME(database_id) = 'tempdb'
```

Table 2. Results from sys.master_files shows tempdb with eight equally sized data files

database_name	physical_name	size
Tempdb	d:\mssql_tempdb\tempdev.mdf	32,768
tempdb	D:\MSSQL_TEMPDB\templog.ldf	64
tempdb	D:\MSSQL_TEMPDB\tempdev2.ndf	32,768
tempdb	D:\MSSQL_TEMPDB\tempdev3.ndf	32,768
tempdb	D:\MSSQL_TEMPDB\tempdev4.ndf	32,768
tempdb	D:\MSSQL_TEMPDB\tempdev5.ndf	32,768
tempdb	D:\MSSQL_TEMPDB\tempdev6.ndf	32,768
tempdb	D:\MSSQL_TEMPDB\tempdev7.ndf	32,768
tempdb	D:\MSSQL_TEMPDB\tempdev8.ndf	32,768

(Note that this computer also has eight processor cores, though this is not visible here. The number of cores can usually be obtained by running `msinfo32` from a command prompt.)

NOTE: You can query sys.database_files from any database to get a full list of the files and information about whether the files are transaction logs or rows. For more information about this file catalog view, see sys.database_files (Transact-SQL) (http://msdn.microsoft.com/en-us/library/ms174397.aspx).

For the preceding example, a subsequent query of sys.dm_exec_requests returns the following results, which show the contention moving to different files in **tempdb**:

Table 3. Subsequent queries show contention moving to different files in tempdb

session_id	status	command	database_id	blocking_session_id	wait_type	WaitTime	wait_resource
53	suspended	INSERT	2	541	PAGELATCH_UP	31	2:3:1
65	suspended	INSERT	5	541	PAGELATCH_UP	37	2:3:1
113	suspended	SELECT	5	541	PAGELATCH_UP	24	2:3:1
152	suspended	INSERT	5	541	PAGELATCH_UP	24	2:3:1
179	suspended	INSERT	5	541	PAGELATCH_UP	5	2:3:1
182	suspended	INSERT	5	541	PAGELATCH_UP	24	2:3:1
228	suspended	INSERT	5	541	PAGELATCH_UP	34	2:3:1
882	suspended	INSERT	5	740	PAGELATCH_UP	38	2:3:1

(Rows removed for brevity)

932	suspended	INSERT	5	740	PAGELATCH_UP	26	2:3:1
NULL	NULL	NULL	NULL	NULL	NULL	23	NULL

Further querying shows that the contention migrates among all of the data files periodically and does not always appear on pageID 1.

For example, a query may return the following results, which show the contention on a different pageID.

Table 4. Contention sometimes shows on pageID 3 instead of pageID 1

session_id	status	command	database_id	blocking_session_id	wait_type	WaitTime	wait_resource
118	suspended	INSERT	5	493	PAGELATCH_UP	2	2:7:3
120	suspended	INSERT	5	493	PAGELATCH_UP	5	2:7:3
150	suspended	INSERT	5	490	PAGELATCH_UP	28	2:7:3
259	suspended	INSERT	5	609	PAGELATCH_UP	28	2:7:3
260	suspended	INSERT	5	609	PAGELATCH_UP	9	2:7:3
296	suspended	INSERT	5	609	PAGELATCH_UP	25	2:7:3
305	suspended	INSERT	5	609	PAGELATCH_UP	7	2:7:3
321	suspended	INSERT	5	609	PAGELATCH_UP	0	2:7:3
351	suspended	INSERT	5	609	PAGELATCH_UP	36	2:7:3
876	suspended	INSERT	5	609	PAGELATCH_UP	2	2:7:3

(Rows removed for brevity)

NULL	NULL	NULL	NULL	NULL	NULL	16	NULL

Understanding the Contention

After you have detected **tempdb** contention, you can start to determine what this contention means.

Overview of PFS, GAM, and SGAM Pages

Every data file begins with the same series of pages, which keep track of file allocation and usage:

- **PageID 1 is the Page Free Space or PFS page.**
  The PFS page keeps track of which pages are allocated. It also keeps track of how full (in percentage) those pages are.
- **PageID 2 is the Global Allocation Map or GAM page.**
  The GAM page keeps track of which extents are allocated and which are available for allocation.
- **PageID 3 is the Shared Global Allocation Map or SGAM page.**
  Because it is inefficient to allocate a full extent for small objects, the first eight pages for any object are allocated on shared extents, and subsequent space for a table or index is allocated on uniform extents. The SGAM page keeps track of which extents are shared extents.

You can verify the type of any page contained in the SQL Server buffer by querying sys.dm_os_buffer_descriptors.

```
SELECT DB_NAME(database_id) as dbname,page_type

FROM sys.dm_os_buffer_descriptors

WHERE database_id = 2 AND file_id = 1 AND page_id = 1
```

You should change the database_id, file_id, and page_id to reflect the page you are interested in. For example, querying sys.dm_os_buffer_descriptors for the wait_resource in Table 1 shows that pageID 1 is a PFS page:

Table 5. Results of buffer descriptors

dbname	page_type
tempdb	PFS_PAGE

Whenever data is inserted into a table that does not have a clustered index, the PFS page must be accessed to determine which page contains enough free space to hold the row that is being inserted. If data is being inserted by several processes simultaneously, all of these processes need to find free space in which to place the data.

These processes can all insert data into separate tables without contention on any Index Allocation Map (IAM) pages because each table or index has its own set of IAMs. Each process, however, must refer to the PFS page to find a page with enough free space to hold the row(s) being inserted, and each process must then update the PFS page after data is inserted. If a large number of separate processes are inserting data into heap tables, and those tables are in an area of a data file where allocation is tracked by a single PFS, the referral to the PFS page and the subsequent updating of this page can become a bottleneck.

Because many different processes can create and drop objects at the same time, and because by default the first eight pages are allocated in shared extents, each process must refer to the SGAM to find available shared extents, and each process must then update the SGAM when the extent is no longer used. With a large number of processes dealing with rapid allocation and deallocation of pages on small tables or indexes, SGAM updating can become a bottleneck.

In the earlier example, both PFS and SGAM contention can occur, and both can be caused by the same set of processes.

Trace Flag 1118 Fails to Resolve the Issue

When DBAs identify SGAM contention, they typically enable trace flag 1118. After turning trace flag 1118 on in the earlier example, contention no longer shows on SGAM; contention may show on additional PFS pages, however, and the total contention may actually increase.

Table 6. Enabling trace flag 1118 results in a change in the pattern of contention

session_id	status	command	database_id	blocking_session_id	wait_type	WaitTime	wait_resource
65	suspended	INSERT	5	932	PAGELATCH_UP	20	2:8:24264
113	suspended	INSERT	5	932	PAGELATCH_UP	9	2:8:24264
152	suspended	INSERT	5	932	PAGELATCH_UP	15	2:8:24264
179	suspended	INSERT	5	932	PAGELATCH_UP	9	2:8:24264
182	suspended	INSERT	5	932	PAGELATCH_UP	7	2:8:24264
210	suspended	INSERT	5	932	PAGELATCH_UP	12	2:8:24264
212	suspended	INSERT	5	932	PAGELATCH_UP	24	2:8:24264
214	suspended	INSERT	5	932	PAGELATCH_UP	12	2:8:24264
(Rows removed for brevity)							
791	suspended	INSERT	5	932	PAGELATCH_UP	24	2:8:24264
NULL	NULL	NULL	NULL	NULL	NULL	20	NULL

Querying sys.dm_os_buffer_descriptors shows that page 24264 is another PFS page. If you continue querying in this example, you may notice that the contention migrates among PFS pages across the **tempdb** data files, and the total contention has not been reduced.

Identifying the Cause of the Contention

A DBA can usually reduce or even eliminate **tempdb** contention, but there are some scenarios in which the **tempdb** contention cannot be resolved by the DBA. The earlier example is such a case because the contention is caused by what is happening in the query l

sys.dm_exec_requests contains a sql_handle column, a statement_start_offset column, and a statement_end_offset column. You can use these columns to examine the waiting processes and see which queries are causing the contention. To use these columns, modify the query that has been used to get the waiting processes as follows.

```
SELECT r.session_id, r.status, r.command,

r.database_id, r.blocking_session_id, r.wait_type,

r.wait_time, r.wait_resource, t.text,

stmt = SUBSTRING(t.text, (r.statement_start_offset/2) + 1,

      CASE r.statement_end_offset

            WHEN -1 THEN DATALENGTH(t.text)

            ELSE (r.statement_end_offset - r.statement_start_offset)/2

      end)

FROM sys.dm_exec_requests r join sys.dm_exec_sessions s on r.session_id =
s.session_id

CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) t

WHERE wait_type IS NOT NULL and s.is_user_process = 1
```

This query returns the information you typically see with the waiting processes and returns the full batch text in the TEXT column and the specific statement within each batch in the STMT column.

In the example, the results show many rows that are executing a function and a few that are executing a stored procedure, as follows.*

```
CREATE FUNCTION Person.USR_GetTopDuplicateCustomer  (@base  int)

RETURNS @retval TABLE

(

      AddressLine1  NCHAR(600)  NOT NULL,

      AddressLine2  NCHAR(600)  NULL,

      City    NVARCHAR(30)  NOT NULL,

      StateProvinceID  INT     NOT NULL,

      CT     INT     NOT NULL

)

AS

BEGIN


      INSERT INTO @retval
```

```
      select AddressLine1, AddressLine2, City, StateProvinceID, ct = COUNT(*)
      from Person.Address o
      WHERE AddressID >= @base
      GROUP BY AddressLine1, AddressLine2, StateProvinceID, City
      HAVING COUNT(*) > 1

      return
end
```

The specific line in the function is the text of the STMT column.*

```
      INSERT INTO @retval
      select AddressLine1, AddressLine2, City, StateProvinceID, ct = COUNT(*)
      from Person.Address o
      WHERE AddressID >= @base
      GROUP BY PersonID, AddressLine1, AddressLine2, StateProvinceID, City
      HAVING COUNT(*) > 1
```

Another statement within another batch is also running among the contending statements.*

```
SELECT DISTINCT
      (Select top 1 AddressLine1
            from Person.USR_GetTopDuplicateCustomer(AddressID)) as
AddressLine1,
      (Select top 1 ISNULL(AddressLine2, '')
            from Person.USR_GetTopDuplicateCustomer(AddressID)) as
AddressLine2,
      (SELECT TOP 1 City
            FROM Person.USR_GetTopDuplicateCustomer(AddressID)) as City,
      (SELECT TOP 1 StateProvinceID
            FROM Person.USR_GetTopDuplicateCustomer(AddressID)) as
StateProvinceID,
```

```
      (Select TOP 1 ct

            FROM Person.USR_GetTopDuplicateCustomer(AddressID)) as ct

FROM Person.AllAddresses   a

WHERE AddressID IN

      (SELECT AddressID FROM Person.USR_IsInTop1000Customers(@base))
```

The TEXT column shows the full batch text and indicates that this statement is found in a stored procedure named "Person.GetNextDuplicateCustomerSet."

At this point in the example, you can see the call stack by querying sys.dm_exec_requests. The stored procedure Person.GetNextDuplicateCustomerSet contains a SELECT query that calls the function "Person.USR_IsInTop1000Customers" once for each row evaluated in the SELECT query. If the name of the function is correct, you expect this to result in a maximum of 1,000 rows. The Person.USR_GetTopDuplicateCustomer function is then called five times for each of those 1,000 rows—one call for each column returned by the correlated subqueries in the column list. Although the Person.USR_GetTopDuplicateCustomer TVF is simple, it is not constructed as an inline TVF.

Each call to a multi-statement TVF requires SQL Server to create a table variable to hold the return values, allocate pages to that table as required, search for free space, and update the corresponding PFS page as data is added. When the table variable goes out of scope, SQL Server must deallocate the space used by that variable. All of this work on table variables occurs in **tempdb**. As larger numbers of users begin to use the system, and as the size of the data contained in the table variables increases, contention develops in the SGAM and PFS pages used to track allocation and space available for these table variables.

This **tempdb** contention causes SQL Server to be unable to use its available resources efficiently. To improve the throughput and to scale the application further, you must eliminate this "hot spot contention."

NOTE: Although there are calls to TVFs in the column list in this function, there are cases of **tempdb** contention caused by queries with calls to TVFs only in the WHERE clause.

When Did Contention Start?

In the earlier example, contention appeared with as few as three concurrent users, and average wait times for latches became higher than 10 ms with as few as 25 concurrent users. As the number of concurrent users increased, both the number of processes waiting for latches and the average time each process waited for a latch increased linearly. Changing the amount of data being stored in the table variables also had a linear effect on latch wait time for any observed load level.

Reducing or Eliminating Contention

When you have identified the cause of the contention, you can start to determine how to reduce or eliminate it.

In the earlier example, there is nothing further a DBA can do with configuration to alleviate the contention. Changes must be made to the queries and to the method used to build the desired result

set. Assuming that the existing queries result in the correct data set, you should concentrate on pulling this data set more efficiently.

The options to eliminate the contention are:

- Take advantage of the new syntax (as of Microsoft SQL Server 2005) to eliminate the need for the correlated subqueries and to lower the number of times the TVFs are called.
- Use a JOIN clause to replace the correlated subquery in the WHERE clause.
- Eliminate the use of TVFs for the data in the SELECT list.

Each of these solutions may work in some scenarios.

Eliminating the Correlated Subqueries in the Selected Columns

Depending on the query, you can eliminate calls to TVFs in a SELECT list by doing any of the following:

- Use an APPLY operator in the FROM clause instead of individual correlated subqueries for each column.
- Pull the data into a temporary or staging table and performing an appropriate JOIN to that table.
- Rewrite the query so that temporary objects are not necessary to pull the appropriate data.

Using an APPLY Operator

In queries such as that in the earlier example, a single TVF is called multiple times in a series of correlated subqueries, and each call to that function uses the same parameter. Prior to SQL Server 2005, this was the only way to use this TVF to retrieve the data. SQL Server 2005, however, introduced the APPLY operator, and this provides another option.

In the example, the query is constructed such that it uses the TVF in correlated subqueries on each column: The table variable is built and dropped once for each column when the function is called on each row. A CROSS APPLY or OUTER APPLY clause offers an advantage over the multiple correlated sub-queries in the SELECT list because one table variable is returned for each row. In the example, the function is called for each of the five columns, so it is possible to reduce the number of times the table variable is returned for each row from five to one by using an APPLY operator.

Following is an example of writing a query to use the APPLY operator.

NOTE: You may need to modify the TVF to ensure that it only returns one row when only the first row of the results set is used.

```
SELECT DISTINCT

        d.AddressLine1

      , d.AddressLine2

      , d.City

      , d.StateProvinceID

      , d.ct
```

```
FROM Person.AllAddresses   a

CROSS APPLY Person.USR_GetTopDuplicateCustomer(AddressID) d

WHERE AddressID IN

      (SELECT AddressID FROM Person.USR_IsInTop1000Customers(@base))
```

This query replaces a call to Person.USR_GetTopDuplicateCustomer for each column in each row with a single call to Person.USR_GetTopDuplicateCustomer for the row. This reduces the number of calls to that stored procedure by 80 percent.

Note, however, that Person.USR_IsInTop1000Customers still gets called for each row of the results set, and because the argument used to call it never changes, the same result set is returned every time. You can therefore further reduce the potential for contention by obtaining this result set only once, as follows.

```
-- retrieve the list of top 1,000 customers first


SELECT AddressID INTO #top1000Customers

FROM Person.USR_IsInTop1000Customers(@base)


-- use the temp table

-- instead of calling the TVF for every row

SELECT DISTINCT

        d.AddressLine1

      , d.AddressLine2

      , d.City

      , d.StateProvinceID

      , d.ct

FROM Person.AllAddresses   a

CROSS APPLY Person.USR_GetTopDuplicateCustomer(AddressID) d

WHERE AddressID IN

      (SELECT AddressID FROM #top1000Customers)
```

NOTE: Multi-statement TVFs are very different from inline TVFs. Temporary objects are not created by inline TVFs as they are with multi-statement TVFs. Inline TVFs may be another option to consider, but

note that even when using inline TVFs, using the correlated subqueries in the column list of the SELECT is inefficient. You should therefore use the CROSS APPLY clause when possible with inline TVFs.

Using Temporary Tables with JOINS

Another option is to rewrite the procedure so that the multi-statement TVF is not called for each column, or even for each row. Sometimes queries using multi-statement TVFs are written to reuse the logic in a TVF; with SQL Server, however, you must consider how the data is retrieved.

In the earlier example, the purpose of the query is to determine which addresses out of the next 1,000 addresses appear more than once in the table. Although the user-defined function makes it easy to reuse logic, the extra overhead of allocating and deallocating space for the table variable being returned by the function puts an additional load on SQL Server and causes a bottleneck in **tempdb** where the TVFs are created.

A JOIN clause can often replace the logic of the correlated subqueries in the column list of the SELECT query. In the earlier example, a JOIN on a single object eliminates the need to make calls to the TVF in the column list. If the data used in the SELECT lists can be pulled into a temporary table once per execution of the stored procedure, a JOIN can then be used to include this data in the final result set rather than using the TVFs in the query.

Using temporary staging tables provides a way to break down the logic if it is too complex to be efficiently performed in a single query. Using temporary tables also lets temporary objects be created only once per execution, and this reduces the potential for **tempdb** contention even more than using CROSS APPLY does.

Following is an example of how you can use temporary tables with joins for this particular example.

```
-- get the next 1,000 addresses


SELECT TOP 1000 AddressID

INTO #top1000Customers

FROM Person.Address WHERE AddressID >= @base

ORDER BY AddressID


-- get the list of duplicate addresses


select AddressLine1, AddressLine2, City, StateProvinceID, ct = COUNT(*)

INTO #duplicateAddresses

FROM Person.Address

WHERE AddressID IN (SELECT AddressID FROM #top1000Customers)
```

```
GROUP BY AddressLine1, AddressLine2, StateProvinceID, City
HAVING COUNT(*) > 1



SELECT a.AddressID, a.AddressLine1, ISNULL(a.AddressLine2, '') AddressLine2,
      a.City, a.StateProvinceID, b.ct
FROM Person.Address a JOIN
#top1000Customers ta ON a.AddressID = ta.AddressID
JOIN
#duplicateAddresses b ON a.AddressLine1 = b.AddressLine1
      AND ((a.AddressLine2 = b.AddressLine2)
OR (a.AddressLine2 IS NULL AND b.AddressLine2 IS NULL))
      AND a.City = b.City AND a.StateProvinceID = b.StateProvinceID
```

Rewriting the Query to Eliminate the Need for Temporary Objects

When steps that perform significant transformations produce the query results, using a temporary table gives SQL Server a way to calculate statistics on intermediate result sets and makes query performance more predictable. However, with a simple query, the results can be efficiently limited within the query without creating a temporary object.

In this example, the logic of the query can be expressed as follows.

```
/*
      get the list from this set of AddressIDs where duplicates exist,
      and give the count of the occurrences of this in the
      next 1,000 addresses
*/


SELECT a.AddressID, a.AddressLine1, ISNULL(a.AddressLine2, '') AddressLine2,
      a.City, a.StateProvinceID, b.ct
FROM Person.Address a JOIN
(
      SELECT TOP 1000 AddressID FROM Person.Address WHERE AddressID >= @base
```

```
      ORDER BY AddressID
) ta ON a.AddressID = ta.AddressID -- limit to 1,000
JOIN
(
      select AddressLine1, AddressLine2, City, StateProvinceID, ct = COUNT(*)
      from Person.Address
      WHERE AddressID >= @base
      GROUP BY AddressLine1, AddressLine2, StateProvinceID, City
      HAVING COUNT(*) > 1
) b ON a.AddressLine1 = b.AddressLine1
      AND ((a.AddressLine2 = b.AddressLine2)
            OR (a.AddressLine2 IS NULL AND b.AddressLine2 IS NULL))
      AND a.City = b.City AND a.StateProvinceID = b.StateProvinceID
```

In this example, the additional work created by multiple calls to the multi-statement TVF is eliminated by rewriting the query. This significantly increases the performance of each individual execution, and more importantly, it removes the “hotspot” on **tempdb**.

No contention develops on **tempdb** when the same load is applied with the rewritten stored procedure, which returns the same data. Contention does not develop on **tempdb** even when the load is increased to 100 times as many users as the original load. Query performance improves from about 45 seconds per execution with the original query to less than 1 second per execution with the rewritten query. However, as frequently occurs with performance tuning, a bottleneck on a different resource develops at the higher level of throughput.

Conclusion

Transact-SQL programming or querying practices that rapidly create and drop temporary objects can create significant bottlenecks on **tempdb**, reducing SQL Server throughput even when best practices for configuration of **tempdb** are followed.

Multi-statement TVFs create table variables as return values. Using multi-statement TVFs that are called once or more for each row processed in a query can lead to **tempdb** contention because of the number of times the table variables must be created and dropped in rapid succession.

In the examples described in this white paper, the contention tended to appear on one file at a time, but contention migrated periodically from file to file. Contention was most often observed on the PFS page, but contention occasionally appeared on the SGAM page also.

tempdb contention can be reduced or eliminated by altering the Transact-SQL programming so that these TVFs are not called on each row. Alternately, you can create a temporary table once for each execution if statistics are needed on an intermediate result set, or you can perform the entire operation in a single query, removing the function logic to this query. The single-query alternative was demonstrated in this white paper, eliminating the use of **tempdb** and therefore any contention on **tempdb**. Avoiding the multi-line TVF produced the best query performance and eliminated the **tempdb** contention.

Because of the **tempdb** contention that multi-statement TVFs can cause, you should avoid using multi-statement TVFs when other ways of deriving the result set are available.

Maximizing Throughput with TVPs

Introduction

This technical note looks at considerations of whether to use the SqlBulkCopy, or Table Valued Parameters (TVPs) in a customer scenario encountered as part of a CAT engagement. The decision of which is better depends on several considerations which will be discussed. TVPs offer several performance optimization possibilities that other bulk operations do not allow, and these operations may allow for TVP performance to exceed other bulk operations by an order of magnitude, especially for a pattern where subsets of the data are frequently updated.

Executive Summary

TVPs and MERGE operations make a powerful combination to minimize round trips and batch insert and update data with high throughput. Parallel operation on naturally defined independent sets of data can be performed efficiently like this. The TVP makes optimizations possible that are not possible with bulk insert or other operations types. To get the most out of the operation, you must optimize your underlying table as well as your method for inserting and updating the data. The principles followed in this case emphasize these points:

- Do not create artificial keys with IDENTITY when it is not necessary. This creates a point of contention on heavy, parallel insert operations.
- If old data key values will not expire, use a MERGE operation instead of DELETE and INSERT. This minimizes data operations; rebalancing and page splits, and the amount of data that must be replicated. If old data key values will expire, then test two operations of MERGE followed by a deletion of only the expired keys rather than a DELETE and INSERT of the full data set.
- If not all the data will be changed, modify the “WHEN MATCHED” portion of the MERGE statement to also check that the data that *may* change *has* changed, and only update the data that is actually changed. This minimizes the number of rows of data that are actually modified, and thus minimizes the amount of data that must be replicated to secondaries in Windows Azure SQL Database environments.

Although these are best practices in any environment they become increasingly important in a shared environment such as Windows Azure SQL Database.

Scenario

In a recent engagement, a problem was encountered in performance of inserting and updating data in Windows Azure SQL Database. The scenario was:

- Data from hundreds of thousands of devices needs to be stored in Windows Azure SQL Database
- Each device stores approximately 8000 rows of configuration data across three tables in the database
- Data for each device is updated approximately once per day

- Only the most current data is stored in Windows Azure SQL Database
- The data must be processed at a sustained rate of six devices per second (Approximately 48,000 rows per second)

The first concept tried was to delete the data for each device first, then use the BulkCopy API to insert the new rows. Several worker role instances were used to scale out the processing of the data into the database. However; when running this against an Azure SQL Database, this did not give the performance the scenario demanded.

The second approach was to use Table Valued Parameters (TVPs) with stored procedures to do the processing. In the stored procedures, the data was validated first. Next, all existing records were deleted for the device being processed, and then the new data was inserted. This did not perform better than the previous bulk insert option.

We were able to improve the process to meet the performance demands by making optimizations to the tables themselves, and to the stored procedures in order to minimize the lock, latch, and Windows Azure SQL Database specific contention the process initially encountered.

Optimizing the Process

Several optimizations were made to this process.

First, the underlying tables contained identity columns, and data needed to be inserted in sets from several different processes. This created both latch, and lock contention. Latch contention was created because each insert is performed only on the last page of the index, and several processes were trying to insert to the last page simultaneously. Lock contention is created because the identity column was the primary key and clustered index key, so all processes had to go through the process of having the identity value created, then only one at a time could insert. To remedy this type of contention, other values within the data were used as a primary key. In our example, we found composite keys of DeviceID and SubCondition in one table, and a combination of three columns in the second that third tables that could be used to maintain entity integrity. Since the IDENTITY column was not really necessary, it was dropped.

An example of the optimization of the table is

Original Table Definition:

```
CREATE TABLE dbo.Table1
(
	RecordID	BIGINT		NOT NULL	IDENTITY(1, 1),
	DeviceID	BIGINT		NOT NULL,
	SubCondition	NVARCHAR(4000)	NOT NULL,
	Value		NVARCHAR(MAX)	NOT NULL,
	SubValue	TINYINT		NOT NULL,
	CONSTRAINT [pk_Table1] PRIMARY KEY CLUSTERED
```

```
        (
                RecordID ASC

        )
)
```

Optimized Table Definition:

```
CREATE TABLE dbo.Table1
(
        DeviceID     BIGINT         NOT NULL,
        SubCondition NVARCHAR(4000) NOT NULL,
        Value        NVARCHAR(MAX)  NOT NULL,
        SubValue     TINYINT        NOT NULL,
        CONSTRAINT [pk_Table1] PRIMARY KEY CLUSTERED
        (
                DeviceID ASC,
               SubCondition ASC
        )
)
```

The second suboptimal part of the described process is that the stored procedure deleted the old data for a device first, then re-inserted the new data for the device. This is additional maintenance of a data structure as deletes can trigger re-balancing operations on an index, and inserts can result in page splits as pages are filled. Making two data modifications, each with implicit maintenance work that must be done, should be avoided when the data operation can be done with one operation. A “MERGE” operation can be used in place of a DELETE then INSERT provided the updated set of data will not omit previous rows of data. In other words, this works if no SubConditions for any DeviceID in the example table will expire.

Any time data is modified in Windows Azure SQL Database, it must be replicated to two replicas. The DELETE then INSERT method was inefficient in this as well since both the delete and the insert operation must be replicated to the Azure SQL Database replicas. Using a MERGE with only the WHEN MATCHED and WHEN NOT MATCHED conditions will eliminate this double operation, and thus eliminates half of

the data replication, but it still modifies every row of data. In the case of this scenario, at most, 10% of the incoming data would actually be different from existing data. By adding an additional condition to the MATCHED condition so that it reads “WHEN MATCHED AND (source.data <> target.data)” only the rows that contained actual data differences were modified, which means that only the data that was actually changed in the incoming data needed to be replicated to secondaries. Making this modification minimized SE_REPL_SLOW_SECONDARY_THROTTLE, SE_REPL_ACK, and other SE_REPL_* wait types.

The last area of optimization we took was to ensure efficient joining with minimal chance for contention among processes. This action was taken because the optimizer tended to want to scan both source and target tables to perform a merge join when processing the MERGE operation. This was not only inefficient, but caused significant lock contention. To eliminate this contention, the query was hinted with “OPTION (LOOP JOIN)”.

An example of the MERGE written to minimize the amount of data that must be processed into the tables is:

```
CREATE PROCEDURE [dbo].[TVPInsert_test]
      @TableParam  TVPInsertType_test READONLY
AS
BEGIN



        MERGE dbo.Table1 AS target
        USING @TableParam AS source
              ON target.DeviceID = source.DeviceID
                    and target.SubCondition = source.SubCondition
        WHEN MATCHED AND
                    (Source.Value != target.Value
                          OR Source.SubValue != Target.SubValue)
              THEN
              UPDATE SET Value = Source.Value, SubValue = Source.SubValue
        WHEN NOT MATCHED THEN
              INSERT (DeviceID, SubCondition, Value, SubValue)
                    VALUES (Source.DeviceID, Source.SubCondition
                          , Source.Value, Source.SubValue)
        OPTION (LOOP JOIN)
```

```
END
```

The Table Value Type definition created for use with this stored procedure:

```
CREATE TYPE dbo.TVPInsertType_test AS TABLE
(
	DeviceID	BIGINT		NOT NULL,
	SubCondition	NVARCHAR(4000)	NOT NULL,
	Value		NVARCHAR(MAX)	NOT NULL,
	SubValue	TINYINT		NOT NULL,
	PRIMARY KEY CLUSTERED
	(
		DeviceID ASC,
		XPath ASC
	)
)
```

NOTE: Check the properties of joins before using join hints. Loop joins can explode in cost when scans, including range scans, are performed on the inner table (the table accessed second). However; the join in the MERGE is primary key to primary key. In this case, there is no chance for range scans on the inner table, and therefore, the risk of cost explosion on the loop join is eliminated.

Testing the Optimization

Performance was tested by running multiple concurrent processes to process data. Elapsed time was measured as only the time it took to execute the three stored procedures for the in-processing of the new data. In different tests, the amount of data changed in each incoming data set varied so that measurements could be taken with 10%, 18%, 25%, or 100% modified data. Since 10% was determined to be the most that would ever be seen on any particular processing day, the changed percentage of 10% was used as the main indicator of the amount of improvement the optimizations would yield, and other percentages were used to give an indication of what might happen should an exceptional day produce much different data.

To test headroom, the tests were run with 4, 8, 12, and 16 concurrent processes. 8 was considered to be the number of worker roles that would normally be processing data, so this was the main test of record. Testing with 12 and 16 concurrent processes allowed us to determine if it was likely that adding worker roles improved or hurt throughput, and thus evaluate whether bursts above the normal level of processing could be handled by scaling out the worker role tier.

In the tests, data was processed with no delay between data sets, and the elapsed time to process the data into the database was recorded. Initially, 8000 tests were run and statistics taken on it to give the indication. However; the number of tests was reduced to 1000 when comparing just the original stored procedures with the optimized stored procedures because the original method produced so much contention that it became obvious with the lower number of tests that the optimizations were worthwhile.

The comparison between stored procedures with 8 concurrent processes was:

Milliseconds	Original Stored Procedures and Tables	Optimized Stored Procedures and Tables
AVG	13095.27	422.91
Median	12553.00	356.00
Standard Deviation	3489.15	255.88
Max	32306	2750
Min	4460	210
MS/Device*	1636.91	52.86

*** MS/Device was calculated as the average time/number of concurrent processes. It can be read as "On average, one device was processed every ___ milliseconds." One device every 53 milliseconds is well within the requirement of 6 devices per second.**

Conclusion

The question of whether to use SqlBulkCopy or TVP is not always a question of which operates faster. When not all the data that is received actually changes data in the table, using a TVP as a parameter for a stored procedure, and optimizing appropriately can lead to very significant performance advantages over other methods that must delete and insert full sets only.

Additionally, ensuring the underlying tables do not make use of IDENTITY columns or other order-forcing mechanism allows for data modifications to be spread over multiple database pages, thus removing the potential for contention for the single, last-page where ordered writes of new data will be performed.

Resolving PAGELATCH Contention on Highly Concurrent INSERT Workloads

Introduction

Recently, we performed a lab test that had a large OLTP workload in the Microsoft Enterprise Engineering Center. The purpose of this lab was to take an intensive Microsoft SQL Server workload and see what happened when we scaled it up from 64 processors to 128 processors. (Note: This configuration is supported as part of the Microsoft SQL Server 2008 R2 release.). The workload had highly concurrent insert operations going to a few large tables.

As we began to scale this workload up to 128 cores, the wait stats captured were dominated by PAGELATCH_UP and PAGELATCH_EX. The average wait times were tens of milliseconds, and there were a lot of waits. These waits were not expected, or they were expected to be a few milliseconds only.

In this TechNote we will describe how we first diagnosed the problem and how we then used table partitioning to work around it.

Diagnosing the Problem

When you see large waits for PAGELATCH in **sys.dm_os_wait_stats**, you will want to do the following. Start your investigation with **sys.dm_os_waiting_tasks** and locate a task waiting for PAGELATCH, like this:

```
SELECT session_id, wait_type, resource_description
FROM sys.dm_os_waiting_tasks
WHERE wait_type LIKE 'PAGELATCH%'
```

Example Output:

session_id	wait_type	resource_description
42	PAGELATCH_EX	7:1:122
46	PAGELATCH_EX	7:1:122
48	PAGELATCH_EX	7:1:122
53	PAGELATCH_EX	7:1:122

The **resource_description** column lists the exact page being waited for in the format: **<database_id>:<file_id>:<page_id>**.

Using the **resource_description** column, you can now write this rather complex query that looks up all these waiting pages:

```
SELECT wt.session_id, wt.wait_type, wt.wait_duration_ms
, s.name AS schema_name
, o.name AS object_name
```

```
, i.name AS index_name
FROM sys.dm_os_buffer_descriptors bd
JOIN (
SELECT *
, CHARINDEX(':', resource_description) AS file_index
, CHARINDEX(':', resource_description
  , CHARINDEX(':', resource_description)) AS page_index
, resource_description AS rd
FROM sys.dm_os_waiting_tasks wt
WHERE wait_type LIKE 'PAGELATCH%'
) AS wt
ON bd.database_id = SUBSTRING(wt.rd, 0, wt.file_index)
AND bd.file_id = SUBSTRING(wt.rd, wt.file_index, wt.page_index)
AND bd.page_id = SUBSTRING(wt.rd, wt.page_index, LEN(wt.rd))
JOIN sys.allocation_units au ON bd.allocation_unit_id = au.allocation_unit_id
JOIN sys.partitions p ON au.container_id = p.partition_id
JOIN sys.indexes i ON  p.index_id = i.index_id AND p.object_id = i.object_id
JOIN sys.objects o ON i.object_id = o.object_id

JOIN sys.schemas s ON o.schema_id = s.schema_id
```

The query shows that the page we are waiting for is in a clustered index, enforcing the primary key, of a table with this structure:

```
CREATE TABLE HeavyInsert (
  ID INT PRIMARY KEY CLUSTERED
  , col1 VARCHAR(50)
) ON [PRIMARY]
```

What is going on here, why are we waiting to access a data page in the index?

Background Information

To diagnose what was happening in our large OLTP workload, it's important to understand how SQL Server handles the insertion of a new row into an index. When a new row is inserted into an index, SQL Server will use the following algorithm to execute the modification:

1. Record a log entry that row has been modified.
2. Traverse the B-tree to locate the correct page to hold the new record.
3. Latch the page with PAGELATCH_EX, preventing others from modifying it.
4. Add the row to the page and, if needed, mark the page as dirty.
5. Unlatch the page.

Eventually, the page will also have to be flushed to disk by a checkpoint or lazy write operation.

However, what happens if all the inserted rows go to the same page? In that case, you can see a queue building up on that page. Even though a latch is a very lightweight semaphore, it can still be a contention

point if the workload is highly concurrent. In this customer case, the first, and only, column in the index was a continuously increasing key. Because of this, every new insert went to the same page at the end of the B-tree, until that page was full. Workloads that use IDENTITY or other sequentially increasing value columns as primary keys may run into this same issue at high concurrency too.

Solution

Whenever many threads need synchronized access to a single resource, contention can occur. The solution is typically to create more of the contended resource. In this case, the contended resource is the last page in the B-tree.

One way to avoid contention on a single page is to choose a leading column in the index that is not continually increasing. However, this would have required an application change in the customer's system. We had to look for a solution that could be implemented within in the database.

Remember that the contention point is a single page in a B-tree. If only there was a way to get more B-trees in the table. Fortunately, there IS a way to get this: Partition the table. The table can be partitioned in such a way that the new rows get spread over multiple partitions.

First, create the partition function and scheme:

```
CREATE PARTITION FUNCTION pf_hash (TINYNT) AS RANGE LEFT FOR VALUES (0,1,2)

CREATE PARTITION SCHEME ps_hash AS PARTITION pf_hash ALL TO ([PRIMARY])
```

This example uses four partitions. The number of partitions you need depends on the amount of INSERT activity happening on the table. There is a drawback to hash-partitioning the table like this: Whenever you select rows from the table, you have to touch all partitions. This means that you need to access more than one B-tree – you will not get partition elimination. There is a CPU cost and latency cost to this, so keep the number of partitions as small as possible (while still avoiding PAGELATCH). In our particular customer case, we had plenty of spare CPU cycles, so we could afford to sacrifice some time on SELECT statements, as long as it helped us increase the INSERT rate.

Second, you need a column to partition on, one that spreads the inserts over the four partitions. There was no column available in the table for this in the Microsoft Enterprise Engineering Center scenario. However, it is easy to create one. Taking advantage of the fact that the ID column is constantly increasing in increments of one, here is a simple hash function of the row:

```
CREATE TABLE HeavyInsert_Hash(
  ID INT NOT NULL
  , col1 VARCHAR(50)
  , HashID AS CAST(ABS(ID % 4) AS TINYINT)  PERSISTED NOT NULL)
```

With the **HashID** column, you can cycle the inserts between the four partitions. Create the clustering index in this way:

```
CREATE UNIQUE CLUSTERED INDEX CIX_Hash
ON HeavyInsert_Hash (ID, HashID) ON ps_hash(HashID)
```

By using this new, partitioned table instead of the original table, we managed to get rid of the PAGELATCH contention and increase the insertion rate, because we spread out the high concurrency across many pages and across several partitions, each having its own B-tree structure. We managed to increase the INSERT rate by 15 percent for this customer, with the PAGELATCH waits going away on the hot index in one table. But even then, we had CPU cycles to spare, so we could have optimized further by applying a similar trick to other table with high insert rates.

Strictly speaking, this optimization trick is a logical change in the primary key of the table. However, because the new key is just extended with the hash value of the original key, duplicates in the ID column are avoided.

The single column unique indexes on a table are typically the worst offender if you are experiencing PAGELATCH contention. But even if you eliminate this, there may be other, nonclustered indexes on the table that suffer from the same problem. Typically, the problem occurs with single column unique keys, where every insert ends up on the same page. If you have other indexes in the table that suffer from PAGELATCH contention, you can apply this partition trick to them too, using the same hash key as the primary key.

Not all applications can be modified, something that is a challenge for ISVs. However, if you DO have the option of modifying the queries in the system, you can add an additional filter to queries seeking on the primary key.

> Example: To get partition elimination, change this:
>
> ```
> SELECT * FROM HeavyInsert Hash
> WHERE ID = 42
> ```
>
> To this:
>
> ```
> SELECT * FROM HeavyInsert_Hash
> WHERE ID = 42 AND HashID  = CAST(ABS(42 % 4) AS TINYINT)
> ```

With partition elimination, the hash partitioning trick is almost a free treat. You will still add one byte to each row of the clustered index.

Bulk Loading Data into a Table with Concurrent Queries

Introduction

This article describes load scenarios for the common data warehouse scenario in which many queries read data from a table while the table is loaded. The key question that administrator/developer has to answer before deciding on a strategy is whether the read queries can afford to wait. If they can wait, you won't want to complicate the loading process by running queries simultaneously with the load. However, if you have a business need to run queries while you load data, read on.

For these tests, we used Microsoft® SQL Server® 2008. We created two identical tables, each containing 18 million rows totaling 2.6 GB. One table was organized as a heap, and the other was organized as a clustered index. These tables were used as targets into which data was bulk inserted. The input data file for the bulk insert had 65,535 rows. We were not concerned about performance differences between loading heaps or indexes; we were looking instead at the impact of queries concurrent with loads.

One of the test's goals was to understand whether read committed snapshot isolation (RCSI) makes any difference in query concurrency during loads. Does it help keep readers "unblocked" from the loading process? And how does the WITH TABLOCK option affect the bulk loading process?

Bulk Insert into Heap Table

For our first test, we loaded data into the heap with RCSI turned off (that is, with the default READ COMMITTED isolation level).

BULK INSERT Without TABLOCK hint, Read Committed

To imitate a data warehouse workload, we had five concurrent connections executing random SELECT statements on the table. While the SELECT statements were running, we started the bulk load operation. We used the **sys.dm_exec_requests** dynamic management view (DMV) to monitor requests on the server, and we observed that the BULK INSERT statement was waiting in the queue with LCK_M_IX wait type. Figure 1. shows the output of **sys.dm_exec_requests** while the BULK INSERT statement was being blocked by the SELECT statement. As soon as the BULK INSERT operation started loading data (Figure 2), all SELECT statements were blocked by the BULK INSERT statement until it completed.

Query	session_id	blocking_session_id	wait_type	wait_time	wait_resource
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	51	0	CXPACKET	1476	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	52	0	CXPACKET	1618	
BULK INSERT dbo.SalesOrderHistory_H FROM 'F:\Samples	53	57	LCK_M_IX	54617	OBJECT: 8:373576369:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	56	0	RESOURCE_	7300	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	57	0	CXPACKET	7302	
SELECT substring (st.text, (statement_start_offset/2)+1, C	59	0	NULL	0	

Figure 1 : BULK INSERT is waiting for the SELECT statement to complete

Query	session_id	blocking_session_id	wait_type	wait_time	wait_resource
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	52	55	LCK_M_IS	3027	OBJECT: 8:373576369:0
BULK INSERT dbo.SalesOrderHistory_H FROM 'F:\Samples	53	0	NULL	0	
SELECT [SalesOrderID],[OrderDate] FROM [DW_Test].[dbo	55	53	LCK_M_IS	3038	OBJECT: 8:373576369:0
SELECT [SalesOrderID],[OrderDate] FROM [DW_Test].[dbo	56	55	LCK_M_IS	3039	OBJECT: 8:373576369:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	57	55	LCK_M_IS	3033	OBJECT: 8:373576369:0
SELECT substring (st.text, (statement_start_offset/2)+1, C	59	0	NULL	0	
SELECT [SalesOrderID],[OrderDate] FROM [DW_Test].[dbo	60	55	LCK_M_IS	3045	OBJECT: 8:373576369:0

Figure 2 : BULK INSERT is loading data, and SELECT statements are blocked

BULK INSERT With TABLOCK hint, Read Committed

For this test, the same five SELECT statements were running concurrently when we started the bulk insert, but this time we specified the WITH TABLOCK option. This time the **sys.dm_exec_requests** DMV indicated that the BULK INSERT operation was waiting with the LCK_M_BU wait type (Figure 3). The LCK_M_BU wait type occurs in SQL Server when a task is waiting to acquire a Bulk Update (BU) lock. Just as with our first test, as soon as BULK INSERT started loading data, we saw that the SELECT statements (Figure 4) were blocked by the bulk insert operation.

Query	session_id	blocking_sessi	wait_type	wait_time	wait_resource
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	52	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	53	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	54	57	LCK_M_IS	284	OBJECT: 8:373576369:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	55	57	LCK_M_IS	706	OBJECT: 8:373576369:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	56	57	LCK_M_IS	1319	OBJECT: 8:373576369:0
BULK INSERT dbo.SalesOrderHistory_H FROM 'F:\Samples	57	53	LCK_M_BU	2734	OBJECT: 8:373576369:0
SELECT substring (st.text, (statement_start_offset/2)+1, C	58	0	NULL	0	

Figure 3: BULK INSERT with TABLOCK waiting for the SELECT statement to complete

Query	session_id	blocking_session_id	wait_type	wait_time	wait_resource
BULK INSERT dbo.SalesOrderHistory_H FROM 'F:\Samples\da	52	0	NULL	0	
SELECT substring (st.text, (statement_start_offset/2)+1, CASE	54	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] where	55	52	LCK_M_IS	36585	OBJECT: 8:373576369:0
SELECT [SalesOrderID],[OrderDate] FROM [DW_Test].[dbo].[S	56	55	LCK_M_IS	36561	OBJECT: 8:373576369:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] where	60	55	LCK_M_IS	31888	OBJECT: 8:373576369:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] where	61	55	LCK_M_IS	31891	OBJECT: 8:373576369:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] where	62	55	LCK_M_IS	31892	OBJECT: 8:373576369:0

Figure 4: SELECT statements waiting for the BULK INSERT statement with TABLOCK to complete

For our second test, we did the same test but with RCSI enabled. We wanted to see whether it would produce different results.

BULK INSERT Without TABLOCK Hint, Read Committed Snapshot Isolation

When we loaded data into the heap table without using the TABLOCK hint, we did not observe any waits. We concluded that option provides the maximum amount of concurrency, if you need to load data into a table while you run SELECT queries.

Query	session_id	blocking_session_	wait_type	wait_time	wait_resource
BULK INSERT dbo.SalesOrderHistory_H FROM 'F:\Sample	51	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	53	0	CXPACKET	7799	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	54	0	CXPACKET	5	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	55	0	RESOURCE	34029	
SELECT [SalesOrderID],[OrderDate] FROM [DW_Test].[dbo	56	0	CXPACKET	7	
SELECT [SalesOrderID],[OrderDate] FROM [DW_Test].[dbo	57	0	NULL	0	
SELECT substring (st.text, (statement_start_offset/2)+1, C	58	0	NULL	0	
select * from OpenRowset(TrcData, @traceid, @records)	60	0	TRACEWRI	10	

Figure 5: BULK INSERT into HEAP table with no TABLOCK hint

BULK INSERT with TABLOCK Hint, Read Committed Snapshot Isolation

Despite the fact that we were using RCSI while we were loading data into the table, the **sys.dm_exec_requests** DMV indicated that the BULK INSERT operation was waiting for a LCK_M_IX lock and that it was being blocked by an active SELECT statement.

Two other SELECT queries, which were issued *after* the BULK INSERT command started, were observed waiting on the LCK_M_S lock type.

Query	session_id	blocking_sessi	wait_type	wait_time	wait_resource
select * from OpenRowset(TrcData, @traceid, @records)	52	0	TRACEWRI	0	
SELECT substring (st.text, (statement_start_offset/2)+1, C	54	0	NULL	0	
SELECT [SalesOrderID],[OrderDate] FROM [DW_Test].[dbo	55	56	LCK_M_S	1196	HOBT: 8:72057594040549376 [BULK_OPERATION]
BULK INSERT dbo.SalesOrderHistory_H FROM 'F:\Samples	56	59	LCK_M_IX	1933	HOBT: 8:72057594040549376 [BULK_OPERATION]
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	58	0	RESOURCE	1198	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	59	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	60	56	LCK_M_S	1202	HOBT: 8:72057594040549376 [BULK_OPERATION]
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_H] wh	61	0	NULL	0	

Figure 6: BULK INSERT into HEAP with the TABLOCK hint

As soon as the SELECT statement with session_id 59 (in Figure 6) was completed, BULK INSERT started loading data, with the selects in sessions 55 and 60 continuing to wait until the BULK INSERT command completed.

We did observe another interesting effect. So far we were dealing only with the BULK INSERT command. But SQL Server 2008 introduced bulk optimization the INSERT INTO … SELECT statement. This operation behaved like BULK INSERT in all the tests on the heap table. The only exception was that if we loaded data into the table using INSERT INTO … SELECT with the TABLOCK hint under RCSI mode, none of the readers was blocked.

Isolation level	TABLOCK specified?	TIME to load data (min:sec)	WAIT TYPE
Read Committed	NO TABLOCK	3:34	LCK_M_IX
Read Committed	WITH TABLOCK	1:28	LCK_M_BU*
Read Committed Snapshot Isolation	NO TABLOCK	0:04	NONE
Read Committed Snapshot Isolation	WITH TABLOCK	1:03	LCK_M_IX

*Very short; difficult to measure how long it was taken

Table 1: Duration of data loading into the heap table

BULK INSERT into a Table with Clustered Index

Next, we loaded data into the table with a clustered index and with RCSI turned off. This test didn't show any surprises and behaved as expected: All read operations were blocked while BULK INSERT was loading data into the table.

If RCSI was enabled, no blocking was observed with BULK INSERT running with concurrent read operations. As we expected, enabling RCSI eliminated locks for read operations that allowed simultaneous loading of the data into the table.

Query	session_id	blocking_session_id	wait_type	wait_time	wait_resource
BULK INSERT dbo.SalesOrderHistory_M FROM 'F:\Samples\day25_	52	56	LCK_M_IX	764	PAGE: 8:1:236244
SELECT substring (st.text, (statement_start_offset/2)+1, CASE WHE	54	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Orde	56	0	CXPACKET	3879	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Orde	57	0	CXPACKET	120	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Orde	61	0	CXPACKET	1523	

Figure 7: BULK INSERT into a table with a clustered index, no TABLOCK hint, and RCSI turned off

Query	session_id	blocking_session_id	wait_type	wait_time	wait_resource
BULK INSERT dbo.SalesOrderHistory_M FROM 'F:\Samples\da	52	60	LCK_M_X	11796	OBJECT: 8:2105058535:0
SELECT substring (st.text, (statement_start_offset/2)+1, CASE V	54	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where	56	0	CXPACKET	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where	57	52	LCK_M_IS	3895	OBJECT: 8:2105058535:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where	58	52	LCK_M_IS	6835	OBJECT: 8:2105058535:0
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where	60	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where	61	52	LCK_M_IS	7077	OBJECT: 8:2105058535:0

Figure 8: BULK INSERT into a table with a clustered index, a TABLOCK hint, and RCSI turned off

Query	session_id	blocking_session_id	wait_type	wait_time	wait_resou
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Order	52	0	RESOURCE_SEM	112	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Order	53	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Order	54	0	NULL	0	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Order	55	0	CXPACKET	122	
SELECT * from [DW_Test].[dbo].[SalesOrderHistory_M] where Order	56	0	RESOURCE_SEM	2633	
BULK INSERT dbo.SalesOrderHistory_M FROM 'F:\Samples\day25_1	58	0	NULL	0	
SELECT substring (st.text, (statement_start_offset/2)+1, CASE WHEN	61	0	NULL	0	

Figure 9: BULK INSERT into a table with a clustered index and RCSI enabled

Figure 9 shows SELECT statements reading from the table while a BULK INSERT is running. BULK INSERT behaved similarly both with and without the TABLOCK hint. This helped us conclude that if you are loading data with RSCI enabled, the TABLOCK hint in the BULK INSERT statement doesn't play a significant role, as far as concurrency is concerned.

Isolation level	**TABLOCK specified?**	**TIME to load data (min:sec)**	**WAIT TYPE**
Read Committed	WITH TABLOCK	6:49	LCK_M_IX
Read Committed	NO TABLOCK	10:18	LCK_M_X
Read Committed Snapshot Isolation	WITH TABLOCK	01:01	NONE
Read Committed Snapshot Isolation	NO TABLOCK	0:48	NONE

Table 2: Duration of data loading into the table with a clustered index

The only time that readers were blocked by bulk operations was when data was loaded into an *empty* table with the clustered index built on it. In that special case only, we observed that read operations were blocked by bulk load operations and read operations if they had the LCK_M_SCH_S wait type. This lock was released only after the loading batch was completed.

Conclusion

RSCI can provide great benefits, if you are bulk loading into the table with concurrent readers working on it . In most cases, it provides you with the ability to read while bulk loads are performed. Bulk loading does not affect the size of the version store in **tempdb** under RCSI; however, RCSI cannot be enabled at the table level. It can be enabled on the entire database only. Therefore you should carefully analyze your workload to ensure that other operations on your database will not cause the **tempdb** size to explode.

Note that RCSI will introduce 14 noncompressible bytes into every row in every table in the database (http://blogs.msdn.com/sqlserverstorageengine/archive/2008/03/30/overhead-of-row-versioning.aspx). An alternative strategy for concurrent readers is to execute queries using the READ UNCOMMITTED isolation level (also known as a dirty read), but this requires application changes, and it can deliver transitionally inconsistent results. RCSI requires no application change, and it guarantees consistent results.

For more information about data loading strategies and scenarios, see the Data Loading Performance Guide at http://msdn.microsoft.com/en-us/library/dd425070.aspx

Section 5: Real World Scenarios

Lessons Learned from Benchmarking a Tier 1 Core Banking ISV Solution - Temenos T24

Background

TEMENOS T24 is a complete banking solution designed to meet the challenges faced by financial institutions in today's competitive market. By working with Microsoft, Temenos was able to take advantage of the latest Windows and SQL Server technologies to tune T24 and run it well on Microsoft platform. The lessons learned in the tech note were derived from T24 solution tuning engagement, but most of them apply to other typical OLTP workloads as well.

Benchmark Overview

The benchmark environment, created to reflect real-world retail banking activity volumes, was made up of 25 million accounts and 15 million customers across 2,000 branches. At peak performance, the system processed 3,437 transactions per second (TPS) in online business testing and averaged a record-breaking 5,203 interest accrual and capitalizations per second during COB testing, processing 25 million accounts in less than two hours. The maximum CPU utilization of the database server during the peak hour did not exceed 70%, providing considerable additional capacity. In addition, the testing demonstrated near linear scalability (95 percent) in building up toward the final the hardware configuration.

T24 Architecture

The T24 solution consists of several layers, as shown below, including:

- User access
- Presentation (clients)
- Messaging/connectivity (web servers)
- Application (application servers)
- Database (database servers)

The application layer accepts messages in a Temenos-specific format called Open Financial Services (OFS). All requests, from a web browser or from a non-web client, are translated into the OFS format and then submitted to the application layer. The communication between the messaging/connectivity layer and the application layer depends on the specific deployment and can use various channels, including message queues, web services, and a native direct connection between the two layers.

T24 was originally designed to use jBASE, a multidimensional database that uses records consisting of fields, multi-values (multi-valued lists), and sub-values. OFS messages are transformed into the internal record format and processed by the application layer; the records are then stored in a jBASE database.

When SQL Server, a supported database system, is used, the jBASE records are transformed into XML format (or in some cases left as BLOBs) and are stored in the database.

1. SQL Server File Configuration

1.1 Configure Data Files

The filegroup used for the T24 data should be composed of multiple files. Best practice is to use one file for every two CPU cores on computer systems with 32 or more cores. On computer systems with less than 32 cores, use the same number of files as the number of CPU cores (the ratio should be 1:1). The data files should be equal in size. Note that the out-of-the-box configuration uses only one file in the primary filegroup, so you need to add additional files for optimal configuration.

Pre-allocate enough space in the data files based on the initial size of the computer system. Monitor the database free space and if necessary extend each file simultaneously so that all of the files have the same amount of free space. SQL Server optimizes writes by spreading its write operations across the files based on the ratio of free space among the files, so extending all files at once maintains this optimization.

Leave the autogrowth setting on as an "insurance policy" so that SQL Server does not stop when it runs out of space; however, do not rely on autogrowth to extend the database files as a standard way of operating. While you should not allocate space for the data files in small units, if you allocate in very large units during autogrowth, the application must wait (possibly several minutes) while the space is allocated. Since you cannot control when autogrowth engages, allocate only by the space needed for a few days of operations.

1.2 Configure Log File

The transaction log file, generally a sequentially written file, must be written as quickly as possible—even before the data is written to the data files (the data portion can be rebuilt from the log if necessary). While there is no performance benefit from using more than one file, multiple files can be beneficial for maintenance purposes (for example, if you are running out of space on the log drive). Adding physical devices to support the LUN can benefit performance.

1.3 Configure tempdb Files

SQL Server tempdb files are used for the storage of temporary data structures. The tempdb files are responsible for managing temporary objects, row versioning, and online index rebuilds. T24 uses a read-committed snapshot isolation level as its default isolation level, which uses row versioning. For more information, see Isolation Levels in the Database Engine.

To ensure efficient tempdb operation:

- **Create one tempdb file per physical CPU core**.
  This reduces page free space (PFS) contention.
- **Pre-size the tempdb files, and make the files equal in size**.
- **Do not rely on autogrow**.
- **Use startup trace flag 1118**.
  For more information about this SQL Server trace flag, see the article Concurrency Enhancements for the tempdb Database.

For information on how to set startup settings for SQL Server, see the article Configure Server Startup Options (SQL Server Configuration Manager.

For further information, see the MSDN article Optimizing tempdb Performance.

2. SQL Server Memory Configuration

2.1 SQL Server Memory Settings

Configure the SQL Server "max server memory (MB)" setting by taking the amount of memory allocated to the database system and subtracting one GB for every four cores (round up). This leaves the operating system with enough memory to work efficiently without having to "grab" memory back from SQL Server. For example, if the server has 64 GB of RAM and 24 cores, set the maximum memory to 58 GB (64 GB minus 6 [24 cores divided by 4]).

2.2 Lock Pages in Memory

To reduce SQL Server paging, you can grant the SQL Server service account "Lock Pages in Memory" privilege through the Windows Group Policy editor.

For detailed instructions, see How to reduce paging of buffer pool memory in the 64-bit version of SQL Server on the Microsoft® Support site.

3. Recovery Interval Change

Increasing the recovery interval server configuration option causes the checkpoint process to occur less often. This can reduce the I/O load driven by checkpoints and improve the overall performance. During lab testing, a recovery interval of 5–10 minutes has been determined to be the best setting for T24.

Before changing the recovery interval, you should consider its implication on the mean time to recovery and recovery point objectives. Note that when using failover clustering, a longer recovery interval also influences the failover time of the database instance.

For more information about the recovery interval option, see the article Recovery Interval Option.

4. Use Trace Flag 834

On computer systems with 64 or more CPU cores, use startup trace flag 834. When this trace flag is set, SQL Server uses Windows large-page memory allocations for the buffer pool. Allocating buffer pages is expensive, and turning on trace flag 834 boosts performance.

For more information about this SQL Server trace flag, see Microsoft Support Article 920093

5. Enable Receive-Side Scaling

You should enable Receive-Side Scaling (RSS) on the SQL Server network interface card (NIC) that is serving the application servers. This setting is found on the Advanced Property tab of the network card. Also, be sure that offloading options are enabled. See the Microsoft® Developer Network (MSDN®) articles Introduction to Receive-Side Scaling and Receive-Side Scaling Enhancements in Windows Server 2008 for more information. If your NIC does not support these options, consider replacing it with one that does.

You should configure the maximum number of RSS processors by setting the MaxNumRssCpus registry key value to 8 on a computer system with 32 or more CPU cores. For computer systems with less than 32 cores, use the default setting.

The RSS base CPU number (RssBaseCpu) is the CPU number of the first CPU that RSS can use. RSS cannot use the CPUs that are numbered below the base CPU number. You should set RssBaseCpu carefully so it does not overlap with the starting CPU.

Lab testing has shown good results with setting both registry key values to 8 (on a computer system with more than 32 cores); this means that 8 RSS processors are used starting with core number 8 to process network traffic.

Note: You should use the Windows RSS registry keys to configure these values instead of NIC settings because NIC settings can be overridden by the Windows registry keys.

6. Index Fill-factor Change

In high-volume deployments (installations with 10 million accounts and more) of T24, you should consider a lower fill factor with PAD_INDEX on for indexes on hot tables with high latch contention. Consider a lower fill factor only if there is need to improve the performance and if excessive latch contention has been observed. Lab testing has shown good results using a fill factor of 50% for hot tables.

Page latch contention can be identified by examining the "SQL Server: Wait Statistics – Page Latch waits" performance counter and querying the dynamic management view sys.dm_os_wait_stats using this query:

```
SELECT * FROM sys.dm_os_wait_stats

WHERE wait_type LIKE 'PAGELATCH%'
```

To identify which tables and which pages experience latch contention, you can use the following queries:

```
SELECT *

FROM sys.dm_db_index_operational_stats (DB_ID('T24'), NULL, NULL, NULL)

ORDER BY [page_latch_wait_in_ms] DESC, tree_page_latch_wait_in_ms DESC
```

and

```
SELECT * FROM sys.dm_os_waiting_tasks

WHERE wait_type LIKE 'PAGELATCH%'
```

For more information on the fill-factor option for indexes, see the article Fill Factor.

7. Optimizing T24 XQueries – Promote Key Attributes/Elements to Relational Columns

To improve query performance, start by identifying slow-running queries. The following query selects the top 50 SQL Server statements ordered by the total CPU time (i.e., total amount of CPU time, in microseconds, for all executions of each statement):

```
SELECT TOP 50

  SUM(query_stats.total_worker_time) AS "total CPU time",

SUM(query_stats.total_worker_time)/SUM(query_stats.execution_count) AS "avg CPU Time",

SUM(query_stats.execution_count) AS "executes",
```

```
SUM(query_stats.total_logical_reads) AS "total logical reads",
SUM(query_stats.total_logical_reads)/SUM(query_stats.execution_count) AS "avg logical reads",
SUM(query_Stats.total_logical_writes) AS "total logical writes",
SUM(query_Stats.total_logical_writes)/SUM(query_stats.execution_count) AS  "avg logical writes",
  MIN(query_stats.statement_text) AS "statement text"
FROM
   (SELECT QS.*,
   SUBSTRING(ST.text, (QS.statement_start_offset/2) + 1,
   ((CASE statement_end_offset
      WHEN -1 THEN DATALENGTH(ST.text)
      ELSE QS.statement_end_offset END
        - QS.statement_start_offset)/2) + 1) AS statement_text
    FROM sys.dm_exec_query_stats AS QS
    CROSS APPLY sys.dm_exec_sql_text(QS.sql_handle) as ST) AS query_stats
GROUP BY query_stats.query_hash
     ORDER BY 1 DESC
```

For T24, we have identified some of the XQueries on the list of the top list. An example of T24 XQuery is:

```
SELECT t.RECID,t.XMLRECORD
FROM F_HOLD_CONTROL t
WHERE t.XMLRECORD.exist(N'/row/c2[.="NEW.LOAN.REPORT"]') = 1
```

For XQuery like this, we can use scalar promotion to reduce the query runtime. A single-value field (or even a specific value of a multi-valued field) that is part of the XMLRECORD can be "promoted" as computed column of the table and be used in relational search conditions. Further, a relational index can be created on the computed column to improve the query performance. The detailed steps to promote a single-value XML field are as follows:

1.) Create a persisted computed column for the specific field.
Create a user-defined function that evaluates the value of the field. The return value of the function should be a single scalar value. Using this function, the computed column should be added to the table and persisted.

```
-- scalar promotion of single valued field
CREATE FUNCTION udf_HOLD_CONTROL_C2(@xmlrecord XML)

RETURNS nvarchar(35)

WITH SCHEMABINDING

BEGIN

    RETURN @xmlrecord.value('(/row/c2/text())[1]', 'nvarchar(35)')

END


ALTER TABLE F_HOLD_CONTROL

ADD C2 AS dbo.udf_HOLD_CONTROL_C2(XMLRECORD) PERSISTED
```

2.) **Create non-clustered index on the computed column.**

After creating the persisted computed column, create an index for this column:

```
-- example 1
CREATE INDEX ix_HOLD_CONTROL_C2 ON F_HOLD_CONTROL(C2)
```

Verify optimizations

Verify that the changes are successful and measure the impact of the optimizations.

- **For scalar promotion (promoted and indexed fields):**

✓ Verify the query translation.

Without scalar promotion, T24 uses a query syntax such as:

```
SELECT t.RECID,t.XMLRECORD
FROM F_HOLD_CONTROL t
WHERE t.XMLRECORD.exist(N'/row/c2[.="NEW.LOAN.REPORT"]') = 1
```

The execution of this query usually uses a table scan to retrieve the results.

After promoting the field “c2”, query should become:

```
SELECT t.RECID,t.XMLRECORD

FROM F_HOLD_CONTROL t
```

```
WHERE t.c2 = 'NEW.LOAN.REPORT'
```

In this case, index lookup on ix_HOLD_CONTROL_C2 is used.

- ✓ Prove that the index is used by reproducing the query and verifying the actual execution plan. You can run the query in SQL Server Management Studio and activate the icon “Include Actual Execution Plan” on the SQL Editor toolbar.
  Alternatively, you can use the SET STATISTICS PROFILE ON statement to display execution plan information.

- ✓ Verify the performance of the query has improved.

- ✓ After using the application for a period of time (e.g., couple of hours or days), use the sys.dm_db_index_usage_stats dynamic management view to verify the index usage. Consider the ratio between index reads and index writes, keeping in mind that an index usually improves the performance for read operations but slows down modifications (i.e., inserts, updates, deletes) at the same time.

- ✓ Consider the number of promoted columns and indexes per table. Too many indexes may degrade the overall performance. As a general rule, you should avoid creating more than seven indexes on a table for T24.

- ✓ Do not create XML indexes on T24 XMLRECORD fields. The impact on transaction latency is too high, and the benefit in query performance is usually not significant.

Section 6: Replication

Initializing a Transactional Replication Subscriber from an Array-Based Snapshot

Overview

This article describes how to initialize a transactional replication Subscriber from an array-based snapshot rather than using the native SQL Server snapshot mechanism. Initializing the Subscriber using a SAN-based restore solution is particularly beneficial for very large databases. In this context, I use the term VLDB to mean a database that is typically multi-terabyte and requires specialized administration and management This is primarily because the standard transactional replication initialization process, which is typically restricted by either the network or storage I/O bandwidth, could take longer than the business service-level agreement (SLA) permits because of the time needed to initialize or recover the Subscriber. In contrast, initializing a Subscriber using an array-based snapshot utilizes the Virtual Device Interface (VDI) freeze and thaw mechanism, thereby minimizing recovery time. This procedure is also particularly beneficial in non-production environments that use transactional replication and require repeatable tests with large volumes of data.

Scope

This procedure was performed using Microsoft SQL Server 2005 Enterprise Edition IA64 with Service Pack 2 (SP2) and cumulative update 9 running on Windows Server 2003 Datacenter Edition IA64. The procedure is expected to be identical in SQL Server 2008; however, this was not tested during the exercise. The storage array was provided by Hitachi Data Systems (HDS), and HDS Split Second was used to manage the array-based snapshots (backup and restore of the databases). VERITAS Storage Foundation HA software was used for volume management.

It should be emphasized that even though the hardware listed above was used, the principle of initializing a Subscriber from an array-based snapshot can be performed using other storage array network (SAN) vendor technologies – specific implementation details will vary. Microsoft recommends that customers attempting this procedure work closely with an engineer from the storage vendor to ensure the solution are implemented correctly.

When Is This Technique Useful?

There may be situations where the Publisher, Distributor, and Subscriber need to be restored after data loss. This technique:

.

Minimizes the transactional replication setup time for the Subscriber through the use of the underlying Virtual Device Interface (VDI) storage mechanisms, which reduce the time required to back up and restore large volumes of data.

· Supports repeated benchmark tests to re-establish a test baseline.

· Provides rapid recovery of the Subscriber if data loss has occurred and the database(s) need to be recovered from a point outside of the distribution retention period.

Background

Transactional replication has been available as a feature in SQL Server since version 6.0. Available functionality has grown since this time to include tracer tokens to measure latency, concurrent snapshot processing, and peer-to-peer replication. However, the general premise has remained the same: to replicate a copy or subset of the data to another database. The Publisher, Distributor, and Subscriber terminology is used to describe the nodes within the topology. For more information about replication, see SQL Server Books Online.

A transactional replication Subscriber can be initialized using one of the following mechanisms:

a) Transactional replication concurrent snapshot processing

b) Database snapshot (this requires SQL Server 2005 Enterprise Edition with Service Pack 2)

c) Initialize from log sequence number (LSN) (SQL Server 2008 only)

d) SQL Server backup (initialize from backup)

e) Copy of the data or array-based restore

A summary of the pros and cons for each technique is presented in a table below.

Option (a) – Transactional replication concurrent snapshot processing does not require an outage, and it allows production activity to occur on the Publisher while the initialization process is copying the schema and data to the Subscriber. Concurrent snapshot processing does not hold shared locks during snapshot generation, thereby allowing production activity to continue. However, a schema modification lock (Sch-M) is taken for a brief period. In contrast to option (d) and (e), this procedure also has the benefit of only copying the schema and data that is required rather than making a complete copy of the database.

Option (b) – SQL Server 2005 Enterprise Edition with Service Pack 2 introduced a new snapshot mechanism that permits the initialization of the Subscriber from a database snapshot. The database snapshot functionality was introduced in SQL Server 2005 to provide a read-only static view of the database using sparse files. Before a page is updated in the source database, the original page is copied to the sparse file. Subsequent updates to the same modified page do not prompt a repeat of this procedure. That is, the pre-change copy of a particular page only pushes into the database snapshot once after the database snapshot is created. It is possible to initialize a transactional replication Subscriber from a database snapshot as described above. This procedure is similar to the concurrent snapshot processing described in option (a) in that it permits transactional activity during the initialization procedure. However, although a database snapshot utilizes the sparse file capability of NTFS, with a lot of concurrent updates, the database snapshot may grow. It is important to ensure that sufficient storage space is allocated for the database snapshot sparse files to store data pages that have been updated during the Subscriber initialization period, because the database snapshot stores the pre-updated page images.

Option (c) – Initialize from LSN was introduced in SQL Server 2008 to aid the configuration of peer-to-peer transactional replication topologies. Initializing from an LSN can also be used in disaster recovery scenarios; instead of performing a full re-initialization of the Subscriber database, you can initialize a Subscriber from an LSN. This means that the Distribution Agent will apply transactions (after the supplied LSN), from the distribution database to the Subscriber, as long as the distribution retention period has been set appropriately.

Option (d) - A Subscriber can also be initialized from a SQL Server backup. For more information, see "Initializing a Transactional Subscription without a Snapshot" in SQL Server Books Online (http://msdn.microsoft.com/en-us/library/ms151705.aspx). This approach restores the complete dataset on the Subscriber and does not require an outage for the initialization, as long as the distribution database stores the in-flight transactions that were taken after the backup. Post-configuration administrative procedures may be required to remove unwanted objects and data on the Subscriber database.

Option (e) - It is also possible to initialize the Subscriber using a copy of the data. The database can be provisioned using any mechanism that will copy the Publisher schema and data to the Subscriber, such as a manual network file copy of the data and log files, a native SQL Server restore, or alternatively, an array-based restore. While this approach does require an outage during the initialization process, an array-based restore is particularly beneficial for very large databases (VLDBs) in either production or benchmark environments where the setup time can be minimized by using the VDI mechanisms. This technique was used to initialize the Subscriber from a HDS-array based snapshot, which is the focus of this article, and is discussed in more detail in the rest of this article. For more information about VDI, see "SQL Server 2005 Virtual Backup Device Interface (VDI) Specification" (http://www.microsoft.com/downloads/details.aspx?familyid=416f8a51-65a3-4e8e-a4c8-adfe15e850fc&displaylang=en

The table below summarizes the pros and cons of the transactional replication snapshot techniques.

Initialization technique	Online	Pros	Cons	Use when
Concurrent snapshot processing	Yes	Available in Workgroup, Standard, and Enterprise editions, allows online processing, generates a snapshot only for published objects.	Duration of Subscriber initialization is impacted by size of database, network, and so on. DML statements can be expensive.	Online processing is a requirement, storage space is limited on the Subscriber.
Database snapshot	Yes	No locking on the Publisher database, in comparison to concurrent snapshot processing.	Enterprise Edition only. Requires sufficient storage for the snapshot sparse files.	Online processing is a requirement, concurrency is essential, sufficient storage exists for the sparse file.

Initialize from LSN	Yes	Full snapshot not required, allows online processing.	SQL Server 2008 Enterprise only, must be able to determine correct LSN.	Only consider if a full snapshot is not feasible and not necessary. May also be suitable for environments that use database mirroring and transactional replication.
SQL Server Backup (initialize with backup)	Yes	Full snapshot not required, allows online processing.	Restores a complete copy of the database on the Subscriber (for example, requires more storage space) and persists copies of all objects unless removed.	Data volume is manageable and offline initialization is not an option. Speed of backup and restore is preferable to use of the **BCP** utility to copy data.
Copy of data / array-based restore	No	Rapid backup/restore of large data volumes, replication snapshot is not required.	Offline processing, must stop transactional activity, requires a complete copy of the data on the Subscriber.	Volume of data is in the terabyte range and backup and restore times are a priority.

Pros and Cons for Transactional Replication Initialization Techniques

Setup Procedure – Initializing the Subscriber from an Array-Based Snapshot

The procedure to initialize the Subscriber from a copy of the data is similar to using the native SQL Server restore mechanism. However, the array-based approach is more useful for large databases, because hardware implementations of the VDI freeze and thaw mechanism allow large amounts of data to be copied quickly and in a transactionally consistent manner, thereby reducing the restore time.

High-Level Steps

The following procedure was used to initialize the Subscriber during a high-end benchmark with a multi-terabyte database. The backup and restore of the database was performed using the HDS array capabilities. This includes the use of HDS Split Second, which is a command-line utility developed by Hitachi Consultancy Services. We also tested a full database backup operation with SQL Server 2005. However, this was simply to benchmark the operation for a 17-terabyte database.

It is important to note that application activity must be paused during this operation, because any change in data at the Publisher will result in data inconsistency. This inconsistency will be raised as an alert in Replication Monitor. One can use **tablediff,** a command line utility that returns detailed information about the differences between two tables. It can also generate a Transact-SQL script to bring a subscription into convergence with data at the Publisher. This utility can be used to correct data inconsistencies; however, it is recommended that precautionary steps be put in place to ensure that the application or any other activity cannot write to the Publisher or Subscriber databases during this procedure. These steps are discussed later. Alternatives to this approach that allow transactional activity during the setup procedure include concurrent snapshot processing, initialization from backup, and initialization from a database snapshot. Generating a replication snapshot for this volume of data would take many hours, so we elected to initialize the Subscriber using an array-based restore in order to minimize the movement of data.

Initializing the Subscriber

The following steps were used to initialize the Subscriber from the array-based restore. The majority of the time was consumed by the backup and restore procedures using HDS Split Second. This was approximately two hours. The other activities were DBA operations to modify and validate the replication metadata.

1. Pause application activity.

2. Disable the application login(s) using ALTER LOGIN <login> DISABLE.

3. Ensure the database is in RESTRICTED_USER WITH ROLLBACK IMMEDIATE mode to clear user connections that may still be in the database. Please note that RESTRICTED_USER does not prevent access to applications run under the context of **sysadmin**, **dbcreater**, or **sysadmin**.

4. Back up the primary (Publisher) database using a storage array-based mechanism.

5. Restore the primary (Publisher) database on a separate instance using the storage array-based mechanism. This database will become the Subscriber.

6. Recover the Subscriber database and check the SQL Server error log to verify completion of the recovery operation.

7. Set the RESTRICTED_USER mode on the subscription database to prevent any user activity. This is simply an additional precautionary step to ensure that applications or users cannot access the database. Please refer to step 3 above, because the same caveat applies.

8. Enable the NOT FOR REPLICATION value on the Subscriber tables (and any other objects that will be published for replication). The NOT FOR REPLICATION option enables you to specify which database objects are treated differently if a replication agent performs a transactional operation. For example, the identity column value is not incremented if

a replication agent performs an insert operation. For more information, see "Controlling Constraints, Identities, and Triggers with NOT FOR REPLICATION" in SQL Server 2005 Books Online (http://msdn.microsoft.com/en-us/library/ms152529(SQL.90).aspx) or SQL Server 2008 Books Online (http://msdn.microsoft.com/en-us/library/ms152529.aspx).

9. Create the publication(s) on the Publisher.

10. Drop any redundant columns on the subscription database using ALTER TABLE DROP COLUMN. For our tests, binary large object (BLOB) columns were dropped, because we did not replicate columns of this data type (primarily to reduce the storage requirement on the Subscriber). We did not drop any other objects, because there were scenarios where we wanted to be able to view or query copies of nonpublished tables on the Subscriber.

11. Optional: Reclaim storage space on the Subscriber by using DBCC CLEANTABLE in the subscription database and by specifying a batch size to reduce the impact on the transaction log. In previous smaller volume tests, we also reclaimed the BLOB storage space, using DBCC CLEANTABLE with a batch size of 100,000. If a batch size is not specified, DBCC CLEANTABLE processes the whole table in one transaction and the table is exclusively locked during the operation. This can require considerable transaction log space for very large tables.

12. Create the subscriptions using the *replication support only* sync_type of sp_addsubscription. This is a key component of the process, because it indicates that the Subscriber already has a copy of the schema and data and does not require initialization with a replication snapshot.

13. Perform DBA checks. For this exercise, we used Transact-SQL scripts to ensure that the number of objects marked for publication was consistent with the number of objects marked for replication at the Subscriber. For example:

```
-- Count the number of columns with the NOT FOR REPLICATION option set to 1.

-- Execute this on the Publisher and Subscriber to ensure the count is consistent.

-- Investigate further if there is a difference.

USE <Publisher or Subscriber>

GO

SELECT COUNT(*) NOT_FOR_REPL_IDENT_Tables

FROM sys.identity_columns

WHERE is_not_for_replication = 1

AND OBJECTPROPERTY(OBJECT_ID, 'IsMSShipped') = 0;
```

Similar statements can also be executed for foreign keys, triggers, and constraints that may require this property to be set.

14. Enable the publication and subscription databases for MULTI_USER activity.

15. Enable the application login(s); for example, use ALTER LOGIN <login> ENABLE.

It is advisable to launch Replication Monitor (Sqlmonitor.exe) to verify that the publications are healthy and that the Log Reader and Distribution agents are not reporting any problems. Similarly, posting a tracer token will also provide a good idea of latency between the databases.

Restoring the Databases for Repeated Benchmark Tests – Initializing the Subscriber from an Array-Based Snapshot

There may be situations where the transactional replication databases need to be restored to a previous point in time, for example, in a benchmark environment to re-establish the baseline for repeated tests. This eliminates the need to re-create the replication objects and also synchronise the Publisher and Subscriber, which can be time-consuming for large databases.

The procedure is similar to that shown above, with the added advantage that SQL Server transactional replication has already been configured and is functioning correctly.

The following steps were used to initialize the transactional replication Publisher, Distributor, and Subscriber from the array-based restore prior to each test cycle.

1. Pause application activity.

2. Disable the application login(s) by using ALTER LOGIN <login> DISABLE; on both the publication and subscription databases.

3. Ensure that the Publisher database is in RESTRICTED_USER WITH ROLLBACK IMMEDIATE mode to clear user connections that may still be in the database. Connections under the context of **sysadmin**, **dbcreator**, or **db_owner** will still be allowed.

4. Stop the SQL Server Agent services on both the Publisher and Subscriber, or disable the Log Reader and Distribution agent jobs to ensure that the array-based backup and restore mechanisms have exclusive access to the database.

5. Set RESTRICTED_USER mode on the subscription database to prevent any user activity.

6. Use the HDS array-based restore mechanisms to:

    a. Restore the Publisher database.

    b. Restore the distribution database.

    c. Restore the Subscriber database.

Note: For large OLTP databases with high transaction volume, you may observe intermittent I/O in System Monitor or Performance Monitor (perfmon.exe) after the restore and recovery. This is typically due to the GHOST_CLEANUP process, which is removing records that have been marked for deletion. This process can be observed in the sys.dm_exec_requests catalog view.

7. Start the SQL Agent Service on the Publisher, Distributor, and Subscriber.

8. Enable the Publisher and Subscriber databases for MULTI_USER activity.

9. Enable the application login(s); for example, use ALTER LOGIN <login> ENABLE.

As with the previous procedure, it is recommended that Replication Monitor be launched to verify the health of the publications. The Log Reader and Distribution agents should also be launched before the application logins are enabled.

The restore of the Publisher, Distributor, and Subscriber databases can be performed in parallel using the HDS array-based restore mechanism. This approach was used to re-establish the baseline for repeated benchmark tests.

Observations and Data Points

A number of observations were documented during the initialization of the Subscriber during the benchmark. This relates to both the initial setup as described above and also to repeated tests conducted during the benchmark. Figure 1.0 below provides a graphical illustration to accompany the following data points. Numbers denote phases in figure 1.0.

1. The data was first loaded into the primary OLTP database without using transactional replication.

2. During this data load phase, the data was shadowed to a second set of volumes. Owing to the volume of data, this was deemed the most appropriate mechanism to provision the Subscriber database due to the volume of data. Loading data with transactional replication enabled was not appropriate, because the operation is fully logged regardless of the database recovery model.

3. After the data load had completed, the volumes were dismounted from the Publisher and mounted on the Subscriber. Transactional replication was then configured using the 'replication support only' sync_type parameter to avoid the need for a replication snapshot.

4. The 17-terabyte Subscriber database was then paired with a set of volumes as a backup. This process took approximately 17 hours.

A full backup of the 17-terabyte OLTP database using the native SQL Server 2005 backup mechanism took approximately four hours using eight streams. Please note that this was conducted on SQL Server 2005 Enterprise Edition. SQL Server 2008 may offer additional gains due to the native backup compression. For the native SQL Server 2005 backup procedure, the following Transact-SQL script was executed.

```
USE [master]

GO

DECLARE

@DateStr nvarchar(50),
```

```
@Stmt nvarchar(1000),

@Filepath nvarchar(50),

@Quote nvarchar(1),

@Databasename nvarchar(50)

SET @Quote = char(39)

SET @Filepath = 'DISK=' + @Quote + '<directory>'

SET @Databasename = '<database>'

SELECT @DateStr = N'_' +

CONVERT(nchar(8),

getdate(), 112) +

N'_' +

RIGHT(N'0' + rtrim(CONVERT

(nchar(2), datepart(hh, getdate()))), 2) +

RIGHT(N'0' + rtrim(CONVERT

(nchar(2), datepart(mi, getdate()))), 2) +

RIGHT(N'0' + rtrim(CONVERT

(nchar(2), datepart(ss, getdate()))), 2)

SELECT @Stmt =

'BACKUP DATABASE ' + @databasename + ' TO ' +

@Filepath + @databasename + @DateStr + '_1.BAK' + @Quote + ', ' +

@Filepath + @databasename + @DateStr + '_2.BAK' + @Quote + ', ' +

@Filepath + @databasename + @DateStr + '_3.BAK' + @Quote + ', ' +

@Filepath + @databasename + @DateStr + '_4.BAK' + @Quote + ', ' +

@Filepath + @databasename + @DateStr + '_5.BAK' + @Quote + ', ' +
```

```
@Filepath + @databasename + @DateStr + '_6.BAK' + @Quote + ', ' +

@Filepath + @databasename + @DateStr + '_7.BAK' + @Quote + ', ' +

@Filepath + @databasename + @DateStr + '_8.BAK' + @Quote +

' WITH NOFORMAT, NOINIT, SKIP, NOREWIND, NOUNLOAD, STATS = 5'

EXEC (@Stmt)
```

You should modify the @Filepath and @Databasename variables if script reuse is intended.

5. New gold (baseline) backups were occasionally required due to schema changes like indexes or partitioning for performance reasons. This HDS Split Second backup took approximately 30 minutes to complete; this was also a function of how much data had changed since the last restore, because the data had changed during the test cycle. While this backup process typically occurs almost instantaneously in the background, we opted to wait for the completion signal from the array before commencing a new test. SQL Server 2005 differential database backup timings were comparable and took approximately the same amount of time.

6. Following a test run, a restore of the Publisher, Distributor, and Subscriber databases took approximately two hours. This was also a symptom of the data that had changed during the test. However, approximately one hour and thirty minutes of this duration was due to the import and export of the VERITAS volumes, which was sequential in nature.

The Publisher and Subscriber databases were restored in parallel, thereby reducing the restore time for the repeated benchmark tests. The Distributor could be restored in parallel; however, we opted to conduct this step separately, because we wanted more control over the procedure.

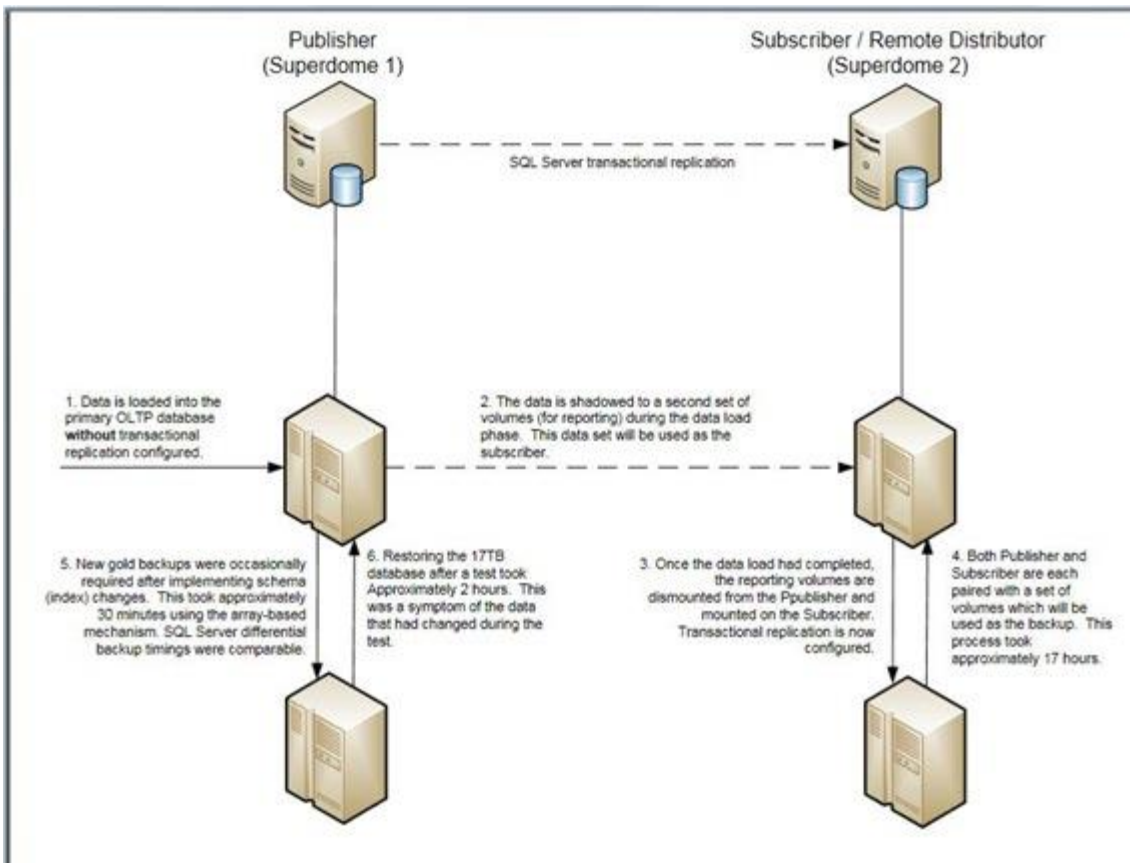


Figure 1.0. Benchmark Data Points

As previously mentioned, before commencing a test, ensure that the GHOST_CLEANUP process has completed on both the Publisher and Subscriber databases (following a restore), because it may affect I/O measurements taken during the test.

Summary

Initializing a transactional replication Subscriber from an array-based restore is beneficial when the data volume is very large and the restore and configuration time need to be minimized. Though this can still be achieved using any other restore or data copy mechanism, the benefit of an array-based VDI freeze and thaw, which allows large amounts of data to be copied quickly and in a transactionally consistent manner, significantly reduces backup and restore time.

Upgrading Replication from SQL Server 2000 32-Bit to SQL Server 2008 64-Bit without re-initialization

Summary

In this article we summarized the experiences we gathered during the planning and upgrade of a customer's project. The customer is one of the largest retail shops in its area. Their architecture is made up of a main server, which is located at headquarters and hosts several hundred publications containing over a thousand articles. Regional stores are located all over the country acting as subscribers for the publications hosted on the main server. Based on the business needs and infrastructure limitations the customer set the following goals for the project:

Install all SQL Server instances on the Windows Server® 2003 x64 platform.
Upgrade the publisher and distributor from the 32-bit edition of SQL Server 2000 to the 64-bit edition of SQL Server 2008. (Upgrading subscribers was not a goal of this phase of the project.)
Avoid replication re-initialization.

This paper describes how to upgrade replication members such as Publishers and Distributors from the 32-bit edition of SQL Server 2000 to the 64-bit edition of SQL Server 2008. Both clustered and nonclustered environments are discussed.

Section 7: Service Broker

SQL Server Service Broker: Maintaining Identity Uniqueness Across Database Copies

A Deployment Method: Copying Databases

When dealing with many large customers, they often develop new and interesting ways of using technology and deploying it. One such case concerning service broker is with large, scale-out deployments consisting of hundreds of servers where the same database will be copied and deployed many times over. This database will contain all of this particular customer's service broker standards and custom built pieces for how broker will operate in their environment. As a package this database can be easily copied and deployed across many instances and servers through 1) "copy & attach," or 2) backup & restore. The caveat for this approach is that each of these databases will contain the same service broker identity. Therefore, we need to reset it. For more details on **Managing Service Broker Identities** see Books Online (BOL):

http://msdn.microsoft.com/en-us/library/ms166057(SQL.100).aspx

We will get to the relevance of having an actual globally unique service broker identity shortly. However, it's important to recognize the database properties that are modified when a database is attached or restored. For our discussion relevant to service broker, I'll point out that the following values are set to 0 when the database is attached or restored:

- is_broker_enabled
- is_honor_broker_priority_on
- is_trustworthy_on

You can view database properties by querying **sys.databases**, see BOL:

http://msdn.microsoft.com/en-us/library/ms178534(SQL.100).aspx

```
SELECT
    name
    ,is_broker_enabled
    ,is_honor_broker_priority_on
    ,is_trustworthy_on
    ,service_broker_guid
FROM sys.databases
```

	name	is_broker_enabled	is_honor_broker_priority_on	is_trustworthy_on	service_broker_guid
1	master	0	0	0	00000000-0000-0000-0000-000000000000
2	tempdb	1	0	0	A12D68E0-649C-46B0-9749-78A30BCC895C
3	model	0	0	0	00000000-0000-0000-0000-000000000000
4	msdb	1	0	1	E9ED53AB-F3C3-4526-8E7D-3C5D10C82915
5	MySSBDB	0	0	0	B4951972-D346-4BCC-B5DE-C819185DCAD3
6	MySSBDBCopy1	0	0	0	B4951972-D346-4BCC-B5DE-C819185DCAD3
7	MySSBDBCopy2	0	0	0	B4951972-D346-4BCC-B5DE-C819185DCAD3

Just after detaching, making a couple copies and attaching the 3 databases, you'll notice that all of my databases have the same service_broker_guid. There is no way to know the purpose, function or environment in which the database is attached or restored. This is why service broker is disabled by default. Now that the database has been duplicated, we need to reset the service broker identity as follows:

```
ALTER DATABASE [MySSBDBCopy1] SET NEW_BROKER
ALTER DATABASE [MySSBDBCopy2] SET NEW_BROKER
GO
SELECT
    name
    ,is_broker_enabled
    ,is_honor_broker_priority_on
    ,is_trustworthy_on
    ,service_broker_guid
FROM sys.databases
```

	name	is_broker_enabled	is_honor_broker_priority_on	is_trustworthy_on	service_broker_guid
1	master	0	0	0	00000000-0000-0000-0000-000000000000
2	tempdb	1	0	0	A12D68E0-649C-46B0-9749-78A30BCC895C
3	model	0	0	0	00000000-0000-0000-0000-000000000000
4	msdb	1	0	1	E9ED53AB-F3C3-4526-8E7D-3C5D10C82915
5	MySSBDB	0	0	0	B4951972-D346-4BCC-B5DE-C819185DCAD3
6	MySSBDBCopy1	1	0	0	4E27CA3D-0550-4D82-926F-F199091E5BA7
7	MySSBDBCopy2	1	0	0	72073C17-AA9B-4E40-8DF2-15F6DB97D2BB

Giving the database a new service broker identity will enable service broker (is_broker_enabled=1) and will remove all service broker messages and conversations in the database with sending end dialog messages. Again, in our scenario since we are deploying a "standards" database we will assume there are no messages or conversations in the database.

Service Broker Diagnostic Utility

Having 1 or more databases with the same service broker guid may not seem problematic. However, service broker does assume that this guid is globally unique across "time and space" independent of the location, instance and server. When messages are sent and received i.e. the initiator and target are the same database, having multiple identical guids may not pose a problem. Unless you'd like to use the new service broker diagnostic tool called the **ssbdiagnose Utility**. See BOL for more detail: http://msdn.microsoft.com/en-us/library/bb934450(SQL.100).asp. SSBDiagnose is new in SQL Server 2008 however it can be used in SQL Server 2005 only environments or mixed. The utility will error when used against an instance where there are 2 or more databases having the same service broker guid. Yes, it is a problem even if the database being diagnosed has a guid that's not duplicated. The Service Broker Diagnostic Utility will generate output similar to:

Microsoft SQL Server 10.0.1442.23
Service Broker Diagnostic Utility
D 29997 MYSERVER\INST2 SSBReceiver1 Service Broker GUID is identical to that of database SSBReceiver on server MYSERVER\INST2
D 29997 MYSERVER\INST2 SSBReceiver2 Service Broker GUID is identical to that of database SSBReceiver

on server MYSERVER\INST2
D 29905 MYSERVER\INST2 SSBReceiver Service Broker is not enabled in the database
3 Errors, 0 Warnings

To summarize, when making a copy of a database establish a new service broker identity. This will avoid any potential future conflicts and will guarantee that your service broker identity will truly be a G.U.I.D.!

Section 8: Troubleshooting

Diagnosing Transaction Log Performance Issues and Limits of the Log Manager

Overview

For transactional workloads I/O performance of the writes to the SQL Server transaction log is critical to both throughput and application response time. This document discusses briefly how to determine if I/O to the transaction log file is a performance bottleneck and how to determine if this is storage related; a limitation is due to log manager itself or a combination of the two. Concepts and topics described in this paper apply mainly to SQL Server 2005 and SQL Server 2008.

Monitoring Transaction Log Performance

To determine if I/O performance of transaction log writes is a problem there are several tools which can help quickly isolate bottlenecks related to the log writes. These are:

1. SQL Server Dynamic Management Views (DMV's).

a. **sys.dm_os_wait_stats**: This DMV exposes a number of wait types documented here. The wait type most relevant the current discussion is WRITELOG. WRITELOG waits represent the time waiting on a log I/O to complete after a COMMIT has been issued by the transaction. When observed these should be an indication I/O performance and characteristics of the log writes should be pursued.

b. **sys.dm_io_pending_io_requests:** This DMV exposes outstanding I/O requests at the individual I/O level. Documentation for this DMV can be found here. In scenarios where the SQL Server transaction log file is not on a dedicated volume this DMV can be used to track the number of outstanding I/O's at the file level. If the transaction log is on a dedicated logical volume this information can be obtained using Performance Monitor counters. More details on both are given below.

2. Window Performance Monitor "SQL Server:Databases" Object. This performance monitor object contains several counters specific to performance of a transaction log for a specific database. In many cases these can provide more detailed information about log performance as the granularity is at the log level regardless of the logical storage configuration. The specific counters are:

a. Log Bytes Flushed/sec

b. Log Flushes/sec - (i.e. I/O operation to flush a log record to the transaction log)

c. Log Flush Wait Time

3. Windows Performance Monitor "Logical or Physical Disk" Objects. There are five counters that should be considered in any I/O analysis:

a. Current Disk Queue Length

b. Avg. Disk/sec Read, Avg. Disk/Write

c. Avg. Disk Bytes/Read, Avg. Disk Bytes/Write

d. Disk Reads/sec, Disk Writes/sec

e. Disk Read Bytes/sec, Disk Write Bytes/sec

The key counters to monitor with respect to log performance are the Current Disk Queue Length and Avg. Disk/sec Write.

These are the primary tools which are used to monitor I/O performance of the transaction log and diagnose any bottlenecks related to transaction log throughput. When troubleshooting any performance issue it is critical to look first at the complete overall system performance before focusing on any individual item. For the purpose of this discussion we will focus on diagnosing transaction log performance.

How Do I Determine if I Have a Log Bottleneck?

The quickest way to determine if you think performance issues are related to transaction log performance is to monitor sys.dm_os_wait_stats for high waits on WRITELOG. It is important to understand that this counter is cumulative since SQL Server was last restarted so monitoring deltas of these values over specific time periods is necessary to provide meaningful details.

There are two primary questions to answer when investigating a performance issue which is suspected to be log related.

1. Is the performance of the I/O subsystem adequate to provide healthy response times to I/O issued against the log?

2. Am I hitting any limits imposed by SQL Server related to transaction log behavior?

In the majority of experiences observed related to I/O performance issues, improperly sized or poorly configured storage is the primary contributor to I/O issues. This can be made worse by queries which are not tuned and issue more I/O than necessary affecting everyone using the I/O subsystem. In addition, there are other factors that will have some impact on log including things such as transactional replication, log backups, mirroring, storage level replication etc...

With respect to #1 our recommendation for response time on the log device should be in the range of <1ms to 5ms. It is important to keep I/O response time on log devices as low as possible which is the primary reason we recommend isolating logs from data files on separate physical spindles. Writes to the transaction log are sequential in nature and benefit by being isolated on to separate physical devices. In today's modern storage environments there are many considerations which may make this impractical so in the absence of the ability to do this focus on keeping response times in the healthy range.

Limits of the Log Manager

Within the SQL Server engine there are a couple limits related to the amount of I/O that can be "in-flight" at any given time; "in-flight" meaning log data for which the Log Manager has issued a write and not yet received an

acknowledgement that the write has completed. Once these limits are reached the Log Manager will wait for outstanding I/O's to be acknowledged before issuing any more I/O to the log. These are hard limits and cannot be adjusted by a DBA. The limits imposed by the log manager are based on conscious design decisions founded in providing a balance between data integrity and performance.

There are two specific limits, both of which are per database.

1. **Amount of "outstanding log I/O" Limit.**

a. SQL Server 2008: limit of 3840K at any given time

b. Prior to SQL Server 2008: limit of 480K at any given time

c. Prior to SQL Server 2005 SP1: based on the number of outstanding requests (noted below)

2. **Amount of Outstanding I/O limit.**

a. SQL Server 2005 SP1 or later (including SQL Server 2008):

i. 64-bit: Limit of 32 outstanding I/O's

ii. 32-bit: Limit of 8 outstanding I/O's

b. Prior to SQL Server 2005 SP1: Limit of 8 outstanding I/O's (32-bit or 64-bit)

Either of the above limits being reached will cause the suspension of the Log I/O until acknowledgments are received. Below are two examples which illustrate the above and provide guidance on how to determine how isolate whether or not these limits are being reached.

Example 1

This example illustrates the limit of 32 outstanding I/O's on the 64-bit edition of SQL Server 2008. The Current Disk Queue Length counter is represented by the black line in the graph. The workload was executing approximately 14,000 inserts per second with each insert being a single transaction. The workload was generated using a multi-threaded client. The disk response time in this example is averaging 2ms. High wait times on WRITELOG were observed as this workload throughput level was approached suggesting investigation of disk throughput on the log device. This led us to investigate if the 32 outstanding I/O limit was the cause.

In Figure 1 below we can see the limit of 32 outstanding I/O's clearly being reached. We can also determine it is the limit of outstanding I/O's vs. the limit of 480K "outstanding log I/O" based on the following calculations (averages over this period of time as observed via Performance Monitor).

· (9.6MB Log Bytes Flushed/sec)/(5000 Log Flushes/sec) = ~1.9K per Flush

· (32 Outstanding I/O's)*(1.9K per Flush) = ~60K "in-flight" at any given time (far below the limit)

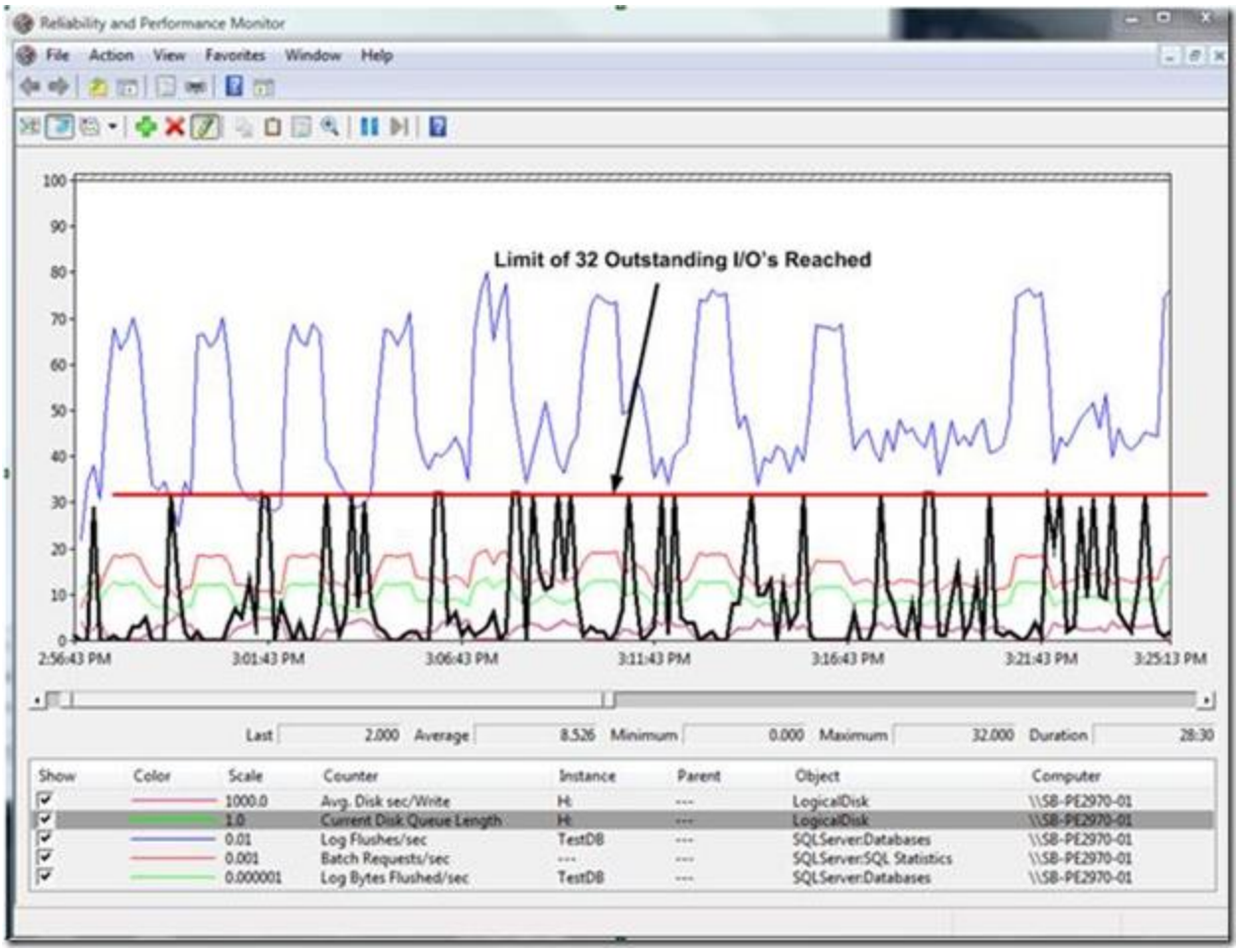


Figure 1: Performance Monitor graph illustrating the limit of 32 outstanding I/O's

In this particular scenario the Current Disk Queue Length could be utilized to diagnose the bottleneck because the log resided on its own logical volume (in this case H:\) . As discussed above, if the log did not reside on a dedicated logical volume, an alternative approach to diagnosing the limit being encountered would have been to use the sys.dm_io_pending_io_requests DMV discussed previously.

The following is a sample query that could be used to monitor a particular log file. This query needs to be run in the context of the database being monitored. It is worth noting that the results returned by this query are transient in nature and polling at short intervals would likely be needed for the data to be useful much in the same way performance monitor samples the disk queue.

```
select vfs.database_id, df.name, df.physical_name

,vfs.file_id, ior.io_pending

,ior.io_handle, vfs.file_handle

from sys.dm_io_pending_io_requests ior
```

inner join sys.dm_io_virtual_file_stats (DB_ID(), NULL) vfs on (vfs.file_handle = ior.io_handle)

inner join sys.database_files df on (df.file_id = vfs.file_id)

where df.name = '[***Logical Log File Name***]'

For this example, the best strategy to resolve or optimize the transaction log performance would be to isolate the transaction log file on a storage device which was not impacted by the checkpoint operations. In our storage configuration this was not possible and shared components (specifically array controllers) were the primary source of the problem.

Example 2

This example is a little more complicated but illustrates the limit of 480K "in-flight" data running SQL Server 2005 SP2 (64-bit edition). This was observed during a benchmark of a high throughput OLTP application supporting approximately 15,000 concurrent users. Characteristics of this application were a bit different in that the log flush sizes were much larger due to logging of larger transactions (including BLOB data). Again, high waits on the WRITELOG wait type were observed pointing in the direction of investigating I/O performance of the log device.

One important item to note when considering this data is that the log behavior over this period of time is spiky in nature and it is during the spikes the limits are encountered. If we look at averages over the problem time period we see the following

· (4,858,000 Log Bytes Flushed/sec)/(124 Log Flushes/sec) = 39,177 bytes per flush

· 5.4 ms per log flush but there are spikes. This is on the edge of health latency for the log.

· Current Disk Queue Length shows spikes and when Current Disk Queue Length * Bytes per flush is close to 480KB then we will start to see the Log Manager throttle the writes (observed as higher wait times on WRITELOG).

Figure 2 illustrates this although it is not as obvious to see as the pervious example. The correlation to watch for is (Average Bytes per Flush) * (Current Disk Queue Length) near the 480K limit.

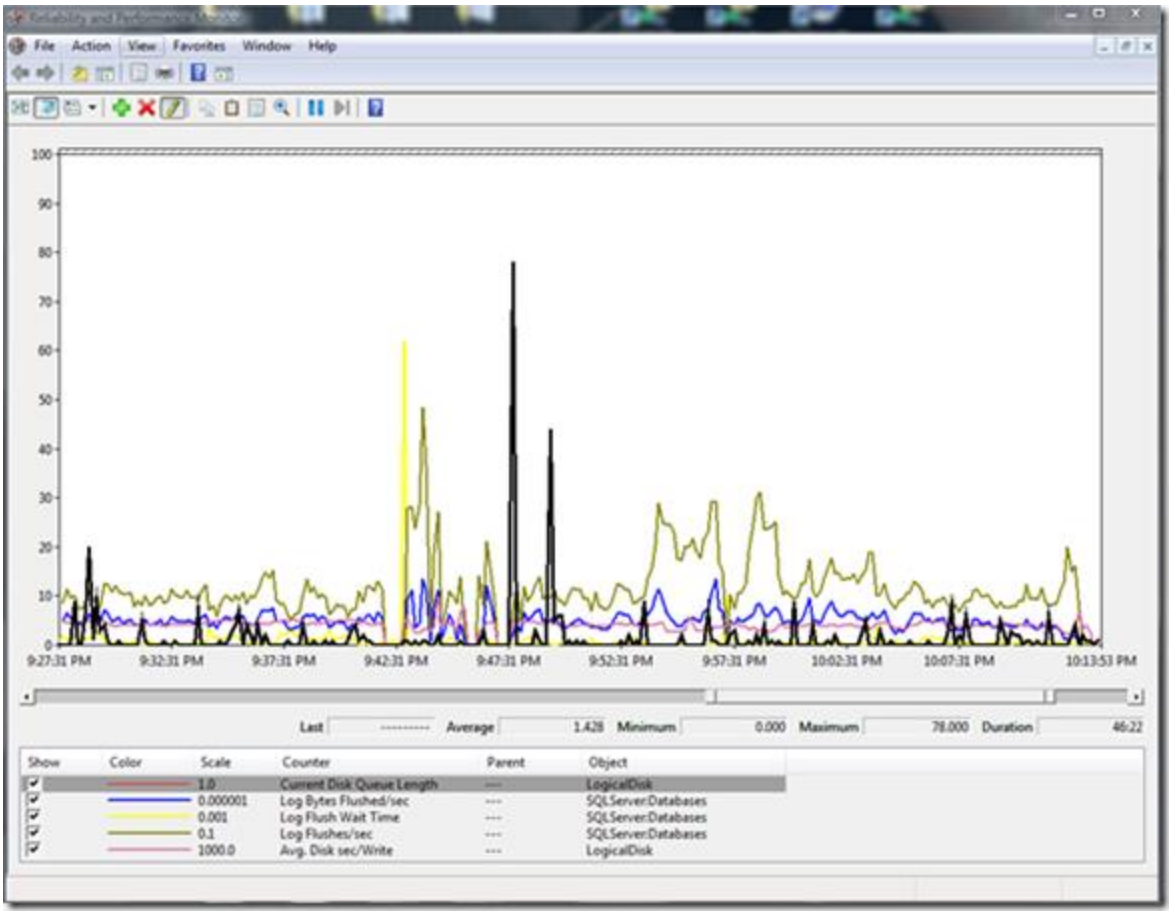


Figure 2: Performance Monitor graph illustrating the limit of 480K outstanding log I/O data

A strategy for resolving this problem would be to work on the storage configuration to bring the latency on the log writes to a lower average. In this particular scenario, log latency was impacted by the fact that synchronous storage replication was occurring at the block level from the primary storage array to a second array. Disabling storage level replication reduces the log latency from the 5ms range to the range of <1ms however disaster recovery requirements for zero data loss made this not an option for the production environment.

It is worth noting that this was observed on SQL Server 2005 SP2 and migrating to SQL Server 2008 resolved the bottleneck since the limit in SQL 2008 is 8 times that of SQL Server 2005.

In either example reducing log latency is critical to increasing the transactional throughput. In very high throughput scenarios at the extreme end we have observed customers approaching this by alternative methods including 1) utilizing minimal logging capabilities of SQL Server when possible and 2) scaling out the transaction log through partitioning data into multiple databases.

Summary

The above provides information on limits imposed by the SQL Server Log Manager. These limits are based on conscious design decisions founded in providing a balance between data integrity and performance. When troubleshooting performance issues related to log performance, always consider optimizing the storage configuration

to provide the best response time possible to the log device. This is critical to transactional performance in OLTP workloads.

Eliminating Deadlocks Caused By Foreign Keys with Large Transactions

Executive Summary

Validating foreign keys requires additional work for data modification operations and can lead to deadlocks in certain scenarios. There are, however, ways you can work around this contention and eliminate the possibility of deadlocks while checking referential integrity.

In this technical note, we describe a recent test lab in which we were able to observe how SQL Server behaves at scale with tables that use foreign keys. We discuss what we learned about how SQL Server locking and access strategies can change as the data size changes and we describe the effects these changes can have. We also describe how the deadlocking was resolved in our case by preventing different transactions from requesting locks on the same rows or pages, forcing a direct access of the rows and forcing all locks to be taken at the lowest possible granularity. This allowed for very high levels of concurrent imports on tables that use foreign keys.

Introduction

In a typical parent-child modification operation, one table is modified before the related table or tables are modified. In the case of inserts, the parent table is modified first; when the child table receives the corresponding insert, a verification step is initiated. In the case of deletes, the child table is removed first; when the parent table is deleted, a verification step is carried out to ensure that no child records exist for the parent record. It is in the verification steps that deadlocking can occur.

When Microsoft SQL Server operates with large INSERT … SELECT combo statements or with large DELETE statements, SQL Server might choose verification methods that increase the chances of deadlocking. Disabling page locks and lock escalation on certain objects involved in these transactions and specifying OPTION (LOOP JOIN) might be necessary to prevent deadlocking in systems that have multiple concurrent large transactions on tables that use foreign keys.

In a recent test lab, we were able to observe how SQL Server behaves at scale with tables that use foreign keys and we tested these locking and access strategies that can prevent deadlocking. In our case, 24 or 48 processes imported data concurrently. Each of these processes performed a bulk insert into a local temporary table where the data was scrubbed and converted. Each process then used an INSERT … SELECT combo statement to move the scrubbed data into the permanent tables.

Our system performed well with low volumes of data in the tables and with low volumes of data in the import transactions. However, severe deadlocking surfaced that effectively prevented concurrent imports as the tables grew and as the size of the data imported from the temporary staging tables expanded. We observed that this deadlocking occurred even though different import processes dealt strictly with different sets of data that were loaded into the same tables.

To resolve deadlocking, we typically resulted had to rollback most of the concurrent transactions, a major setback when processing these large amounts of data. Moreover, retrying the rolled-back transactions often resulted in additional deadlocks.

Validation of Foreign Keys

In our benchmark, foreign keys were a critical part of the database design. Dropping foreign keys was not considered a viable option. Everyone understood that foreign keys would require additional work for data manipulation operations, but this was deemed to be an equitable tradeoff for maintaining the data integrity, regardless of the source of the data manipulation operation.

SQL Server places the validation of foreign keys into the query plan after the data is inserted into the target table. You see the validation of foreign keys as a join to the table against which the data must be validated.

Figure 1 shows a query plan with foreign key validation.

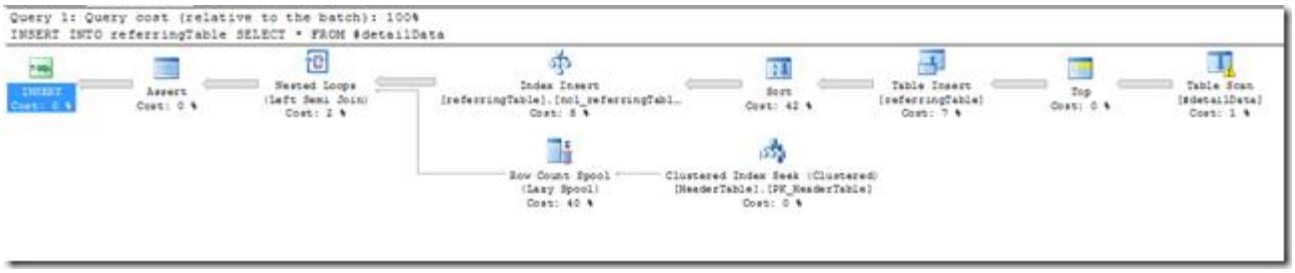


Figure 1: Execution plan showing SQL Server using a loop join to verify referential integrity

In Figure 1, an INSERT … SELECT statement is used to insert data into *dbo.referringTable* from *#detailData*. The table *dbo.referringTable* has a foreign key that references *dbo.HeaderTable*.

Starting from the upper right, you can see the flow of the data as SQL Server first scans the *#detailData* table for the data to be inserted and then inserts the data into *dbo.referringTable*. After inserting the data into the base table, SQL Server sorts the data so that the rows can also be inserted into the non-clustered index *ncl_referringTable*. This sort makes the insert into that non-clustered index most efficient, but note that an estimated 42% of the total query cost is spent on sorting the data in the order of the non-clustered index.

After the data is inserted into *dbo.referringTable* and all of its associated indexes, SQL Server verifies that referential integrity is maintained by using a loop join operation to perform a left semi-join to *dbo.HeaderTable*. Note that the row count spool operation that is used to support this join accounts for an estimated 40% of the total query cost. More importantly, the verification does not take place until after the data is inserted into the target table (*dbo.referringTable*, in this case). It is only at this point that SQL Server can verify whether the data inserted into the target table violates the foreign key constraint, so it is at this time that SQL Server determines whether the transaction can continue or if it must be rolled back.

Note SQL Server acquires shared locks when validating foreign keys, even if the transaction is using read committed snapshot (read committed using row versioning) or snapshot isolation level. Be mindful of this when examining deadlock graphs from transactions when these transaction isolation levels are used. If you see shared locks, check to see whether the locks are taken on an object that is referenced by a foreign key.

Contention Issues as Transactions Grow

As the amount of data grows, the query plan is likely to change, typically when statistics get updated (automatically or manually) or when the SQL Server Query engine determines that a significant change in table size necessitates a different query plan. Additionally, when an index is created, the statistics are updated with fullscan, also causing a potential change in the query plan.

For large datasets, when the clustered index is added (whether the index is added before or after the data is inserted) can determine whether SQL Server uses a nested loop join or a merge join during the foreign key validation.

In the query plan shown in Figure 1, the temporary table from which the data to be inserted is selected has no clustered index key. The statistics have probably been updated, but only by auto-create and auto-update statistics. This means that the table might contain significantly more data than the statistics indicate. Therefore, we added a clustered index to the *#detailData* temporary table after data was added in an effort to improve join efficiencies (resulting in up-to-date statistics on *#detailData's* new index).

Figure 2 shows the query plan for the same INSERT … SELECT statement.

Figure 2: Execution plan showing SQL Server using a merge join to validate referential integrity

Figure 2 and Figure 1 show query plans for the same query, but because SQL Server recognizes that a larger dataset is being inserted in Figure 2, SQL Server replaces the loop join to validate referential integrity in Figure 1 with a merge join in Figure 2. This change eliminates the row count spool operation, but the most significant effect it has, from a concurrency standpoint, is that the clustered index seek on *HeaderTable.PK_HeaderTable* is replaced by a clustered index scan on *HeaderTable.PK_HeaderTable*.

Understanding the potential deadlocking

The new query plan in Figure 2 sets up potential deadlocks in transactions. Consider the following progression with two transactions, T1 and T2:

1. T1 opens a transaction and inserts a few rows into *HeaderTable*. Because only a few rows are inserted, an intent exclusive (IX) lock is taken on *HeaderTable* and on the pages into which the rows will be inserted. Exclusive key (X) locks are taken on the rows that are actually inserted.

2. T2 opens a transaction and inserts a few rows into *HeaderTable*. Because only a few rows are inserted, an IX lock is taken on *HeaderTable* and on the pages into which the rows will be inserted. Exclusive key locks are taken on the rows that are actually inserted. No blocking has occurred at this point.

3. T1 begins inserting data into *ReferringTable*. Locks are taken as appropriate on this table.

4. T2 begins inserting data into *ReferringTable*. Locks are taken as appropriate on this table. Lock escalation is prevented because multiple transactions concurrently hold locks on this table.

5. When T1 data is inserted into *ReferringTable*, T1 begins the scan of *HeaderTable.PK_HeaderTable* to verify referential integrity. Before the scan is complete, T1 is blocked in its attempt to read the rows that have been inserted into *HeaderTable* by T2.

6. When T2 data is inserted into *ReferringTable*, T2 begins the scan of *HeaderTable.PK_HeaderTable* to verify referential integrity. Before the scan is complete, T2 is blocked in its attempt to read the rows that have been inserted into *HeaderTable* by T1. This is now a deadlock.

Blocking occurs on *HeaderTable* when referential integrity is validated because shared locks are taken for this validation, even if the transaction itself is using read committed snapshot isolation or snapshot isolation levels.

We also encountered deadlocking because, in some instances, a row locking strategy was chosen for inserting rows, but a page locking strategy was chosen for validation of referential integrity. In this case, T1 and T2 could deadlock when trying to validate referential integrity because a shared lock on the page was requested by one transaction to validate the referential integrity, but the other transaction had an IX lock on the page because of X locks on rows inserted into that page.

Eliminating the Contention

To work around the contention, it is necessary to prevent different transactions from requesting locks on the same rows or pages. The general idea is to force a direct access of the rows and to force all locks to be taken at the lowest possible granularity.

In our benchmark, we used the following steps:

1. Ensure that different transactions process different data.

2. Disable lock escalation on *HeaderTable*.

3. Disable page locking on indexes on *HeaderTable*.

4. Hint the INSERT … SELECT query with OPTION (LOOP JOIN).

Ensure that different transactions process different data

In the case of our benchmark, there were 24 or 48 processes running concurrently to import data, and each processed a different dataset. For inserts, data must first be inserted into *HeaderTable* and then the details that reference those header records are inserted into *detailTable*. Mutually exclusive datasets were an integral part of the process; if the datasets were not mutually exclusive, there would have been primary key violations when inserting data into the *HeaderTable*. Note that it might be possible for multiple processes that all refer to the same row in the *HeaderTable* to insert data into *detailTable*; however, the processes must address the situation to prevent accessing the same rows.

Disable lock escalation on HeaderTable

If locks are allowed to escalate on *HeaderTable*, different types of deadlocking might occur. In our case, lock escalation on *HeaderTable* would result in severe contention, preventing the required level of concurrency. Because we also planned to disable page locking, we would have forced inserts into *HeaderTable* to take row locks, and single statements could reach the escalation threshold much faster than if the statement were able to take page locks. For the sake of concurrency, therefore, we disabled lock escalation.

Lock escalation can be enabled or disabled on a table by using the ALTER TABLE syntax. To disable lock escalation on *HeaderTable*, execute the following query:

```
ALTER TABLE HeaderTable SET (LOCK_ESCALATION = DISABLE)
```

Note Disabling lock escalation is not appropriate for all scenarios; in some cases, disabling lock escalation can cause out-of-memory errors in SQL Server. We recommend disabling lock escalation only for dealing with specific contention issues and only after extensive testing. In some cases, lock escalation can be disabled temporarily for processes that are known to have contention issues.

Note that you can determine whether lock escalation is allowed or disabled on a table by querying sys.tables.

Disable page locking on indexes on HeaderTable

You can disable page locking on indexes to keep different granularities of locks from causing deadlocks. Disabling page locking on indexes eliminates the possibility of contention that is caused by the inserts holding key locks (which implies IX locks are held on the containing pages) and by the validation of referential integrity taking shared page locks. If page locking is disaabled, the process of checking foreign keys takes row locks, even if a scan is performed.

To disable page locking on *HeaderTable.PK_HeaderTable*, you can execute the following query:

```
ALTER INDEX PK_HeaderTable ON HeaderTable SET (ALLOW_PAGE_LOCKS = OFF)
```

Note Disabling page locking is not appropriate for all scenarios; in some cases, disabling page locking can cause additional memory contention or can lock escalation, which can increase lock contention. We recommend disabling page locking on an index only for dealing with specific contention issues and only after extensive testing.

Note that you can determine whether page locking and row locking are enabled by querying sys.indexes.

Hint the INSERT … SELECT query with OPTION (LOOP JOIN)

If a scan is performed to support a merge or hash join, every row in that index or table must be read. If another transaction holds a lock on a row, the scan cannot complete until the lock on that row is released. This sets up the possibility of deadlock. To eliminate this possibility, you can force an access strategy that avoids touching rows locked by other transactions. We can use the OPTION (LOOP JOIN) when we know that the query must validate foreign keys to prevent the deadlock.

If an index exists to support a seek on the inner table, SQL Server can navigate from the root of that index directly to the leaf page where the row that needs to be accessed exists when a loop join is used. No other rows need be

checked. This eliminates the possibility of a query being blocked by locks on rows in the table that are locked by other transactions. (We eliminate the scan by hinting for a loop join to be used, and this prevents the blocking and deadlocking.)

With normal joins, for example, you can use the INNER LOOP JOIN or the LEFT OUTER LOOP JOIN syntax in the FROM clause of the query to specify a join type. However, the table that is referenced by a foreign key is not explicitly stated in the query. Because SQL Server uses a join to validate foreign keys, you can still bring about the chosen join strategy by using the OPTION clause in the query.

In the query plans shown in Figures 1 and 2, the query text was as follows:

```
INSERT INTO referringTable SELECT * FROM #detailData
```

This query allowed SQL Server to choose the join type that was used to validate the referential integrity. In the two query plans, SQL Server chose two different join strategies. To force SQL Server to use a loop join to validate the referential integrity, the query should read as follows:

```
INSERT INTO referringTable SELECT * FROM #detailData

OPTION (LOOP JOIN)
```

When looking at the resulting query plan, you will notice that the overall estimated subtree cost in the execution plan is higher with the OPTION (LOOP JOIN) hint than it was when SQL Server chose the scan and merge join. This cost difference is the reason SQL Server chose the scan and merge join. However, in our case, the contention prevented concurrency and therefore limited the overall throughput. It was worthwhile to choose a query that had a higher estimated cost to gain the concurrency and to allow for a higher level of overall throughput. You can determine whether this is true in your environment only by testing.

Challenges to using the OPTION (LOOP JOIN) hint

Using the OPTION (LOOP JOIN) hint causes SQL Server to use a loop join to perform all joins in the query, not just joins that are used to validate foreign keys. If SQL Server previously found another join strategy to be more efficient on other joins, it will now use a loop join operator to perform those joins as well.

In some cases, this can cause a cost explosion. You cannot hint those other joins specifically to use a different join type, because this will result in an error informing you that you have conflicting join hints. If you must use the OPTION (LOOP JOIN) hint, consider the following tips:

· Look at the execution plan on the other joins. Ensure that the inner table (the table that appears on the bottom on the execution plan) has an index to support the loop join without scanning. For example, consider the following query:

```
SELECT l.col1, l.col2, r.col3, r.col4

FROM leftTable l

JOIN rightTable r ON l.col1 = r.col1 AND l.col2 = r.col2
```

OPTION (LOOP JOIN)

If the execution plan shows that rightTable is the inner table, ensure that rightTable has an index on (col1, col2) or on (col2, col1), whichever is appropriate in the situation, to avoid scanning on the inner table. A loop join that has a scan on the inner table is very inefficient and can result in high CPU utilization or other resource contention.

· Loop joins for which the seek on the inner table results in a range scan can also be very inefficient and might result in high CPU utilization or other resource contention. Try to avoid situations in which many rows can be returned from the inner table on any one join value combination.

As with any possible solution, hinting a loop join results in a tradeoff. In this case, we are trading available CPU resources for better concurrency. Be sure to test this solution thoroughly in your environment to verify that the tradeoff is worthwhile.

Summary

Using foreign keys results in additional work for data manipulation operations in which referential integrity is enforced. The validation of the referential integrity is verified with a join. In cases in which the data being inserted is small enough, SQL Server defaults to the use of a loop join. In cases in which the dataset being inserted is known to be large, SQL Server defaults to the use of another join strategy that necessitates a scan of the referenced table. Scanning can result in deadlocking with concurrent insert operations.

In some cases, transactions may use a key locking strategy when inserting into the referenced table and then use a page locking strategy when verifying referential integrity. This different granularity can also result in deadlocking.

SQL Server always uses shared locks when validating referential integrity. This is true even if the transactions are using read committed snapshot (read committed using row versioning) or snapshot isolation levels.

To eliminate the deadlocking when checking referential integrity, you can disable lock escalation on the referenced table, disable page locking on the referenced table, and hint OPTION (LOOP JOIN) on transactions that operate on mutually exclusive datasets. This forces SQL Server to lock the minimal amount of data and to use a more direct seek operation to access the referenced rows while checking referential integrity.

Resolving scheduler contention for concurrent BULK INSERT

Background information

As you may be aware, SQL Server, through SQLOS, implements its own scheduling mechanism on top of the Windows operating system. This is done to spend the maximum amount of CPU time in user mode by using yielding instead of preemptive scheduling. Also, SQLOS can exercise very fine control over the threads by providing an abstraction on top of Windows threads.

Central to the SQLOS scheduling mechanism is the scheduler object. A SQLOS scheduler is an abstraction of a CPU or, in the case of multi-core machines, a CPU-core. Schedulers are grouped into nodes; a node corresponds either to the hardware NUMA nodes on the host machine or to the soft NUMA configuration of SQL Server. For example, an 8 CPU dual core machine with 2 hardware NUMA nodes and no soft NUMA configured has 2 nodes with 8 schedulers in each node.

You can view the SQLOS schedulers and nodes by using the DMV sys.dm_os_schedulers. Notice that there are some extra, special schedulers in this DMV; these are for SQL Server internal use. Also, you will see the dedicated admin connection (DAC) scheduler here (scheduler_id = 255). The schedulers that do the actual query execution work are marked as VISIBLE ONLINE in the status column.

At connection time, the user's session is assigned to a specific node. SQL Server uses a round-robin assignment mechanism to assign the connection to the node. Once a session is on a specific node, it will not move from it for the duration of the connection.

Whenever a session sends a batch request or RPC to the server, a scheduler is assigned to handle the full execution of the request. In SQLOS, this call is known as a task. The assignment of a scheduler is done by identifying the least busy scheduler within the session's node, although the scheduler that was used for a prior task on the same connection is somewhat favored. The assigned scheduler will be used for the duration of the task—it is not possible for SQLOS to do a scheduler switch inside a task.

BULK INSERT commands

There are cases when SQL Server assignment of schedulers is less than optimal. Remember that the scheduler is an SQLOS internal abstraction of a CPU Core. So, if two long-running, CPU-bound queries end up on the same scheduler, they will compete for the same CPU resources. At the same time, another idle scheduler may have CPU time available. Ideally, the idle schedulers would take some of the load from the scheduler that is executing two long-running queries. But remember, the scheduler switch cannot happen in the middle of a task.

Multiple BULK INSERT tasks fit this scenario precisely: a bulk insert, assuming good I/O, is CPU bound and typically runs for a long time. If you execute more than one of them concurrently, you can end up in a situation where two bulk inserts run on the same scheduler, while another scheduler is idle.

If the above situation occurs, your wait stats will show signal waits, even though the system is not actually under CPU pressure. If two CPU-bound queries share a scheduler, each one will periodically yield to the other. These yields show up in sys.dm_os_wait_stats as waits for SOS_SCHEDULER_YIELD.

Summing up, if you see the following pattern on your server:

Total CPU load less than 100%

More than one long-running, CPU-bound query is executing on the same scheduler

Your queries are individually CPU bound, but you are still not able to push CPU load to 100%

Many signal waits in sys.dm_os_wait_stats

Many waits for SOS_SCHEDULER_YIELD in sys.dm_os_wait_stats

… You may have less than optimal scheduler distribution.

The solution: terminate and reconnect

In a well-planned, long-running BULK INSERT workload you may be able to do better than the SQL Server scheduler assignment method. You can actually leverage the fact that a task stays on the same scheduler during its run. The DMV sys.dm_exec_requests contains information about all sessions, including the scheduler that is currently executing your connection. Using this view, you can check to see if the current scheduler is busy. If it is, you can have your client application retry the connection.

Using this “terminate and reconnect” trick we were able to increase the throughput of a CPU-bound batch run by more than 25% on a large 64-core computer. Before we applied this trick, we might have 16 cores almost idle and still we observed SOS_SCHEDULER_YIELD waits. Your mileage may vary depending on the type of work you do and how “lucky” you get with the assignment of schedulers.

The following code snippet can be used to build a "terminate and reconnect" wrapper:

```
CREATE PROCEDURE Batch.Wrapper_DoWork
AS /* Get my scheduler */

DECLARE @my_scheduler_id INT
```

```
SELECT @my_scheduler_id = scheduler_id
FROM sys.dm_exec_requestsWHERE session_id = @@SPID

/* Check if someone else is doing long running work on my scheduler */
IF EXISTS (SELECT *
  FROM sys.dm_exec_requests
  WHERE scheduler_id <> @my_scheduler_id
    AND command LIKE 'BULK%' /* replace with your specific check for long running
query*/
  )
BEGIN
    RETURN 0 /* Failed to get a non busy scheduler, let client try again */
END
ELSE BEGIN
    /* Do long running query here */
    RETURN 1
END
```

Remember that your client application must perform a reconnect if the wrapper returns 0.

The above stored procedure is subject to some race conditions. You can end up in a situation where the scheduler check shows that your scheduler is not busy but in the meantime, before you start your long-running query, another query connects to your scheduler and starts its long running work.

One way to avoid this race condition is to add a semaphore to your batch control system. This semaphore can then be used to ensure exclusive access to a scheduler.

An example of this implementation is:

```
/* Create table to act as semaphore */
CREATE TABLE Batch.ClaimedSchedulers(
  scheduler_id INT PRIMARY KEY
  , session_id INT
)

CREATE PROCEDURE Batch.DoWork_WithClaim
AS

/* Get my scheduler */
DECLARE @my_scheduler_id INT
SELECT @my_scheduler_id = scheduler_id
FROM sys.dm_exec_requests
WHERE session_id = @@SPID

DECLARE @RC INT /* hold the row count of inserted data */

/* Claim the scheduler as my own */
INSERT INTO Batch.ClaimedSchedulers (scheduler_id, session_id)
SELECT @my_scheduler_id, @@SPID
FROM sys.dm_os_schedulers s
LEFT JOIN Batch.ClaimedSchedulers cs WITH (TABLOCKX)  /* ensure serialization */
  ON cs.scheduler_id = s.scheduler_id
```

```
WHERE cs.scheduler_id IS NULL
  AND s.status = 'VISIBLE ONLINE'
  AND s.scheduler_id = @my_scheduler_id
SET @RC = @@ROWCOUNT/* Did my scheduler claim fail? (no row will be inserted) */

IF @RC = 0 BEGIN
  RETURN 0 /* Could not get a non busy scheduler, let client try again */
END
ELSE BEGIN
  /* DO LONG RUNNING QUERY HERE */

  /* Release my scheduler again */
  DELETE FROM Batch.ClaimedSchedulers
  WHERE session_id = @@SPID

  RETURN 1
END
```

Response Time Analysis using Extended Events

This tool demonstrates response time analysis at the session or statement level including waitstats using the new Extended Events infrastructure in SQL Server 2008. This tool is based on the simple principle:

Response time = service time + wait time

This tool allows you to drill down on the time spent in serving the user requests and the time spent in waiting for resources.

Download the application and documentation from http://sqlcat.codeplex.com/Wiki/View.aspx?title=ExtendedEventsWaitstats. Follow the User Guide to install and use the tool. The download also contains the source code for the project.

Memory Error Recovery in SQL Server 2012

SQL Server 2012 has many hidden gems, one of them is the capability of recovering from memory corruption error. We will tell you how it works in this article.

Contents

SQL Server is able to recover from memory corruption when hardware support is available. Platforms that support hardware memory scrubber can send notification to applications when memory corruption is detected. SQL Server responds to these notifications and attempts to repair the memory. Clean database pages in buffer pool are restored by reading the page again from disk. This new feature helps SQL Server to remain running even when there are hardware memory errors.

Hardware errors and taxonomy

Please see the Hardware Errors and Error Sources on MSDN for an overview of hardware errors and their definitions.

Soft error vs. hard error

A soft error is an error in a signal or datum which is wrong. After a soft error is observed , there is no implication that the system is any less reliable than before. If detected, a soft error may be corrected by rewriting correct data in place of erroneous data. An example of a soft error is a single bit flip.

Unlike soft error, a hard error is an error that occurs because of a physical hardware issue such as a defect, a mistake in design or construction, or a broken component. These errors require that the hardware causing the error be replaced. Rewriting the data does not correct the error.

Corrected error vs. uncorrected error

A corrected error is a hardware error condition that has been corrected by the hardware or the firmware by the time that the operating system is notified about the presence of the error condition.

An uncorrected error is a hardware error condition that cannot be corrected by the hardware or the firmware. Uncorrected errors are classified as either fatal or nonfatal.

Fatal error vs. non-fatal error

A fatal hardware error is an uncorrected or uncontained error condition that is determined to be unrecoverable by the hardware. When a fatal uncorrected error occurs, the operating system generates a bug check to contain the error.

A nonfatal hardware error is an uncorrected error condition from which the operating system can attempt recovery by trying to correct the error. If the operating system cannot correct the error, it generates a bug check to contain the error.

What is memory scrubbing?

Memory scrubbing is the process of detecting and correcting bit errors in computer memory by using error-detecting codes like ECC. Memory scrubbing can detect and correct soft, correctable errors. For certain soft, uncorrected non-fatal errors, SQL Server captures of this information and checks whether the corrupted memory is part of a clean database page that is in Buffer Pool. If that is the case, this page is tossed and the memory is de-allocated. If corruption is in another region of memory that can not be repaired, only logging is done to notify of the event and no action is taken.

How To: Find if this feature is available

If the memory error recovery feature is available, you will be able to see the following by looking at SQL errorlog.

```
Machine supports memory error recovery. SQL memory protection is enabled to recover
from memory corruption.
```

How To: Detect that a page has been repaired

If the memory corruption is associated with a clean page in Buffer Pool, SQL Server is able to recover from it and the following message will be logged in the errorlog.

```
SQL Server has detected hardware memory corruption in database '%ls', file ID: %u,
page ID; %u, memory address: 0x%x and has successfully recovered the page.
```

Also you can monitor the detection and repair of the memory corruption with the following Extended Events (XEvents):

- bad_memory_detected: corrupted memory has been detected and reported to SQL. The memory may or may not belong to a database page.

- bad_memory_fixed: memory corruption has been detected and fixed. Fields of then event contain more details regarding the exact page affected

How to: Detect uncorrected hardware memory corruption

SQL Server is able to recover from memory corruption of clean pages in buffer pool. It cannot recover from memory errors associated with dirty pages or outside Buffer Pool. Upon detection of such uncorrectable hardware errors, the following entry can be observed from errorlog.

```
Uncorrectable hardware memory corruption detected. Your system may become unstable.
Please check the Windows event log for more details.
```

How To: Monitor the system

The user can monitor the detection of memory corruption using **sp_server_diagnostics**. The *system* component state will be set to *warning* upon detection of a memory corruption. In addition, the *data* column logs information about count of bad page detected, count of bad page fixed as well as the virtual address of the bad page last encountered.

Section 9: SQL Top 10

Storage Top 10 Best Practices

Proper configuration of IO subsystems is critical to the optimal performance and operation of SQL Server systems. Below are some of the most common best practices that the SQL Server team recommends with respect to storage configuration for SQL Server.

1 - Understand the IO characteristics of SQL Server and the specific IO requirements / characteristics of your application.

In order to be successful in designing and deploying storage for your SQL Server application, you need to have an understanding of your application's IO characteristics and a basic understanding of SQL Server IO patterns. Performance monitor is the best place to capture this information for an existing application. Some of the questions you should ask yourself here are:

- What is the read vs. write ratio of the application?
- What are the typical IO rates (IO per second, MB/s & size of the IOs)? Monitor the perfmon counters:

    1. Average read bytes/sec, average write bytes/sec
    2. Reads/sec, writes/sec
    3. Disk read bytes/sec, disk write bytes/sec
    4. Average disk sec/read, average disk sec/write
    5. Average disk queue length

- How much IO is sequential in nature, and how much IO is random in nature? Is this primarily an OLTP application or a Relational Data Warehouse application?

To understand the core characteristics of SQL Server IO, refer to SQL Server 2000 I/O Basics.

2 - More / faster spindles are better for performance

- Ensure that you have an adequate number of spindles to support your IO requirements with an acceptable latency.
- Use filegroups for administration requirements such as backup / restore, partial database availability, etc.
- Use data files to "stripe" the database across your specific IO configuration (physical disks, LUNs, etc.).

3 - Try not to "over" optimize the design of the storage; simpler designs generally offer good performance and more flexibility.

- Unless you understand the application very well avoid trying to over optimize the IO by selectively placing objects on separate spindles.
- Make sure to give thought to the growth strategy up front. As your data size grows, how will you manage growth of data files / LUNs / RAID groups? It is much better to design for this up front than to rebalance data files or LUN(s) later in a production deployment.

4 - Validate configurations prior to deployment

- Do basic throughput testing of the IO subsystem prior to deploying SQL Server. Make sure these tests are able to achieve your IO requirements with an acceptable latency. SQLIO is one such tool which can be used for this. A document is included with the tool with basics of testing an IO subsystem. Download the SQLIO Disk Subsystem Benchmark Tool.
- Understand that the of purpose running the SQLIO tests is not to simulate SQL Server's exact IO characteristics but rather to test maximum throughput achievable by the IO subsystem for common SQL Server IO types.
- IOMETER can be used as an alternative to SQLIO.

5 - Always place log files on RAID 1+0 (or RAID 1) disks. This provides:

- Better protection from hardware failure, and
- Better write performance.
  Note: In general RAID 1+0 will provide better throughput for write-intensive applications. The amount of performance gained will vary based on the HW vendor's RAID implementations. Most common alternative to RAID 1+0 is RAID 5. Generally, RAID 1+0 provides better write performance than any other RAID level providing data protection, including RAID 5.

6 - Isolate log from data at the physical disk level

- When this is not possible (e.g., consolidated SQL environments) consider I/O characteristics and group similar I/O characteristics (i.e. all logs) on common spindles.
- Combining heterogeneous workloads (workloads with very different IO and latency characteristics) can have negative effects on overall performance (e.g., placing Exchange and SQL data on the same physical spindles).

7 - Consider configuration of TEMPDB database

- Make sure to move TEMPDB to adequate storage and pre-size after installing SQL Server.
- Performance may benefit if TEMPDB is placed on RAID 1+0 (dependent on TEMPDB usage).
- For the TEMPDB database, create 1 data file per CPU, as described in #8 below.

8 - Lining up the number of data files with CPU's has scalability advantages for allocation intensive workloads.

- It is recommended to have .25 to 1 data files (per filegroup) for each CPU on the host server.
- This is especially true for TEMPDB where the recommendation is 1 data file per CPU.
- Dual core counts as 2 CPUs; logical procs (hyperthreading) do not.

9 - Don't overlook some of SQL Server basics

- Data files should be of equal size – SQL Server uses a proportional fill algorithm that favors allocations in files with more free space.
- Pre-size data and log files.
- Do not rely on AUTOGROW, instead manage the growth of these files manually. You may leave AUTOGROW ON for safety reasons, but you should proactively manage the growth of the data files.

10 - Don't overlook storage configuration bases

- Use up-to-date HBA drivers recommended by the storage vendor
- Utilize storage vendor specific drivers from the HBA manufactures website
- Tune HBA driver settings as needed for your IO volumes. In general driver specific settings should come from the storage vendor. However we have found that Queue Depth defaults are usually not deep enough to support SQL Server IO volumes.
- Ensure that the storage array firmware is up to the latest recommended level.
- Use multipath software to achieve balancing across HBA's and LUN's and ensure this is functioning properly
- Simplifies configuration & offers advantages for availability
- Microsoft Multipath I/O (MPIO): Vendors build Device Specific Modules (DSM) on top of Driver Development Kit provided by Microsoft.

Top 10 Hidden Gems in SQL Server 2005

By Cihan Biyikoglu

Technical Reviewers: Lindsey Allen, Peter Scharlock, Burzin Patel, Eric Hanson, Mark Souza, Sanjay Mishra, Michael Thomassy

SQL Server 2005 has hundreds of new and improved components. Some of these improvements get a lot of the spotlight. However there is another set that are the hidden gems that help us improve performance, availability or greatly simplify some challenging scenarios. This paper lists the top 10 such features in SQL Server 2005 that we have discovered through the implementation with some of our top customers and partners.

The order in the list does not have much significance except the specific instances we used them and the impact we saw. I will use a practical analogy; I started with the utility-knife size features that can help make life very easy at the right moment and build up to chain-saw size features that can help you implement a full scenario.

TableDiff.exe

Table Difference tool allows you to discover and reconcile differences between a source and destination table or a view. Tablediff Utility can report differences on schema and data. The most popular feature of tablediff is the fact that it can generate a script that you can run on the destination that will reconcile differences between the tables. TableDiff.exe takes 2 sets of input;

- Connectivity - Provide source and destination objects and connectivity information.
- Compare Options - Select one of the compare options
- Compare schemas: Regular or Strict
- Compare using Rowcounts, Hashes or Column comparisons
- Generate difference scripts with I/U/D statements to synchronize destination to the source.

TableDiff was intended for replication but can easily apply to any scenario where you need to compare data and schema.

You can find more information about command line utilities and the Tablediff Utility in Books Online for SQL Server 2005.

Triggers for Logon Events (New in Service Pack 2)

- With SP2, triggers can now fire on Logon events as well as DML or DDL events.
- Logon triggers can help complement auditing and compliance. For example, logon events can be used for enforcing rules on connections (for example limiting connection through a specific username or limiting connections through a username to a specific time periods) or simply for tracking and recording general connection activity. Just like in any trigger, ROLLBACK cancels the operation that is in execution. In the case of logon event that means canceling the connection establishment. Logon events do not fire when the server is started in the minimal configuration mode or when a connection is established through dedicated admin connection (DAC).
- The following code snippet provides an example of a logon trigger that records the information about the client connection.

```
CREATE TRIGGER connection_limit_trigger
          ON ALL SERVER FOR LOGON
          AS
          BEGIN
          INSERT INTO logon_info_tbl SELECT EVENTDATA()
          END;
```

You can find more information about this feature in updated Books Online for SQL Server Services Pack 2 un the heading “Logon Triggers”.

Boosting performance with persisted-computed-columns (pcc).

- Btree Indexes provide great compromise for tuning queries vs redundant storage of data and added cost of modifying data (insert/update/delete). A less known capability for tuning in SQL Server 2005 is persisted computed columns (PCC). Computed columns can help you shift the runtime computation cost to data modification phase. The computed column is stored with the rest of the row and is transparently utilized when the expression on the computed columns and the query matches. You can also build indexes on the PCC’s to speed up filtrations and range scans on the expression.
- The following sample can demonstrate the benefits of a persisted computed column applied to a complex expression. The same TSQL query run against the following table schema with and without the DayType column will demonstrate the effect of the transparent expression matching with persisted computed columns. The output from the sys.dm_exec_query_stats DMV also shows the difference in the IO and CPU characteristics of the query.

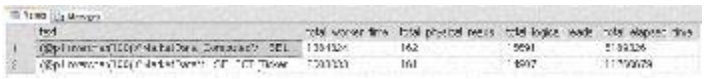
Query

```
SELECT [Ticker] ,[Date] , [DayHigh] ,[DayLow] ,[DayOpen] ,[Volume]
,[DayClose] ,[DayAdjustedClose],
CASE
WHEN volume > 200000000 and dayhigh-daylow /daylow > .05 THEN 'heavy
volatility'
WHEN volume > 100000000 and dayhigh-daylow /daylow > .03 THEN 'volatile'
WHEN volume > 50000000 and dayhigh-daylow /daylow > .01 THEN 'fair'
ELSE 'light'
END as [DayType]
FROM dbo.MarketData
WHERE
CASE
WHEN volume > 200000000 and dayhigh-daylow /daylow > .05 THEN 'heavy
volatility'
WHEN volume > 100000000 and dayhigh-daylow /daylow > .03 THEN 'volatile'
WHEN volume > 50000000 and dayhigh-daylow /daylow > .01 THEN 'fair'
ELSE 'light'
END = 'heavy volatility'
```

Table Schema

```
CREATE TABLE [dbo].[MarketData](
[ID] [bigint] IDENTITY(1,1) NOT NULL,
[Ticker] [nvarchar](5) NOT NULL,
[Date] [datetime] NOT NULL,
[DayHigh] [decimal](38, 6) NOT NULL,
[DayLow] [decimal](38, 6) NOT NULL,
[DayOpen] [decimal](38, 6) NOT NULL,
[Volume] [bigint] NOT NULL,
[DayClose] [decimal](38, 6) NOT NULL,
[DayAdjustedClose] [decimal](38, 6) NOT NULL,
-- PERSISTED COMPUTED COLUMN --
[DayType] AS (
CASE
WHEN volume > 200000000 and dayhigh-daylow /daylow > .05 THEN 'heavy
volatility'
WHEN volume > 100000000 and dayhigh-daylow /daylow > .03 THEN 'volatile'
WHEN volume > 50000000 and dayhigh-daylow /daylow > .01 THEN 'fair'
ELSE 'light'
END) PERSISTED NOT NULL
) ON [PRIMARY]
```

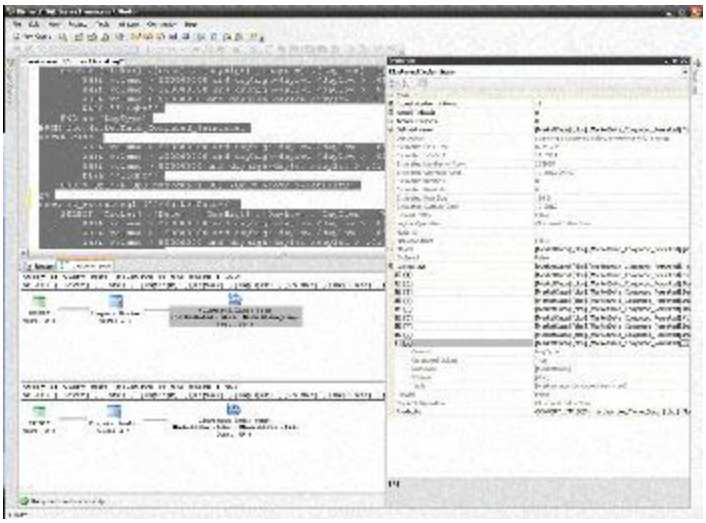
Output From The Sys.Dm_Exec_Query_Stats Dynamic Management View (DMV)



See full-sized image.

In the above picture, the output from sys.dm_exec_query_stats dynamic management view shows the difference in CPU and IO statistics between the same query hitting MarketData_Computed and MarketData tables. Line 1 represents the query run against the table with the persisted computed column. Line 2 is the table without the persisted computed column. With the complex expression pre-calculated in the DayType column, total worker time and overall elapsed time is lower compared to the table without the DayType persisted computed column.

Another way to verify that the persisted computed column is utilized, is to use the execution plan and look at the scan or the seek operator for the table with the computed column and check the output list, which should contain the column. In the example below you can see the DayType, the name for the PCC, in the output list under #9.



See full-sized image.

4 DEFAULT_SCHEMA setting in sys.database_principles

- SQL Server provides great flexibility with name resolution. However name resolution comes at a cost and can get noticeably expensive in adhoc workloads that do not fully qualify object references. SQL Server 2005 allows a new setting of DEFEAULT_SCHEMA for each database principle (also known as "user") which can eliminate this overhead without changing your TSQL code. Here is an example:

In SQL Server 2005, the following query when executed by user1 that has a DEFAULT_SCHEMA of 'dbo' will directly resolve to dbo.tab1, instead of the extra search for user1.tab1.

```
SELECT * FROM tab1
```

Whereas the same query will search for 'user1.tab1' in SQL Server 2000 and if that does not exist it will resolve to 'dbo.tab1'.

- This setting can be especially useful for databases upgraded from SQL Server 2000 to SQL Server 2005. To preserve the original behavior, databases upgraded from SQL Server 2000 will get the username as the DEFAULT_SCHEMA for each database principle. That means, in a database upgraded from a previous version to SQL Server 2005, 'user1' will get a DEFAULT_SCHEMA values of 'user1'. To take advantage of the performance benefits, administrators can set the DEFAULT_SCHEMA through ALTER USER command and change it to the schema that most of the of the objects reside. Be aware this may break queries that may be utilizing objects in other schemas than the one set in the DEFAULT_SCHEMA setting and has not qualified the object names.
- DEFULT_SCHEMA is documented in Book Online under the "CREATE USER (Transact-SQL)" heading.

5 Forced Parameterization

- Parameterization allows SQL Server to take advantage of query plan reuse and avoid compilation and optimization overheads on subsequent executions of similar queries. However there are many applications out there that, for one reason or another, still suffer from ad-hoc query compilation overhead. For those

cases with high number of query compilation and where lowering CPU utilization and response time is critical for your workload, force parameterization can help.

- Force parameterization forces most queries to be parameterized and cached for reuse in subsequent submissions. Forced parameterization will remove the literal values and replaces them with parameters. This minimizes the compilation overhead for queries that are the same except the literal values in the query text. Forced parameterization is typically enabled at the database level. However it is also possible to hint FORCED PARAMETERIZATION on individual queries.
- In a number of cases, we have witnessed improvements in performance up to 30% due to forced parameterization. However forced parameterization can cause inappropriate plan sharing in cases where a single execution plan does not make sense. For those cases, you can utilize features like plan guides or query hints.
- You can find more information on Forced Parameterization in Books Online.

Vardecimal Storage Format

- In Service Pack 2, SQL Server 2005 adds a new storage format for numeric and decimal datatypes called vardecimal. Vardecimal is a variable-length representation for decimal types that can save unused bytes in every instance of the row. The biggest amount of savings come from cases where the decimal definition is large (like decimal(38,6)) but the values stored are small (like a value of 0.0) or there is a large number of repeated values or data is sparsely populated.
- SQL Server 2005 also includes a stored procedure that can estimate the savings before you enable the new storage format.

```
master.dbo.sp_estimate_rowsize_reduction_for_vardecimal 'tablename'
```

- To enable vardecimal storage format, you need to first allow vardecimal storage on the database;

```
exec sys.sp_db_vardecimal_storage_format N'databasename', N'ON'
```

- Once the database option is enabled, you can then turn on vardecimal storage at a table level using the following procedure;

```
exec sp_tableoption 'tablename', 'vardecimal storage format', 1
```

- Vardecimal storage format presents an overhead due to the complexity inherent in variable length data processing. However in IO bound workloads, savings on IO bandwidth due to efficient storage can far exceed this processing overhead.
- If you would like more information on this topic, updated SQL Server 2005 Books Online for Service Pack 2 contains extensive information on the new vardecimal format.

Indexing made easier with SQL Server 2005

- The new Dynamic Management Views have improved monitoring and trouble shooting greatly. A few of the dynamic management views (DMVs) deserve special attention.
- Through sys.dm_index_usage_stats you can find out how much maintenance and traversal you have for each index. Indexes with high maintenance numbers and low traversal numbers can be considered as good candidates for dropping.

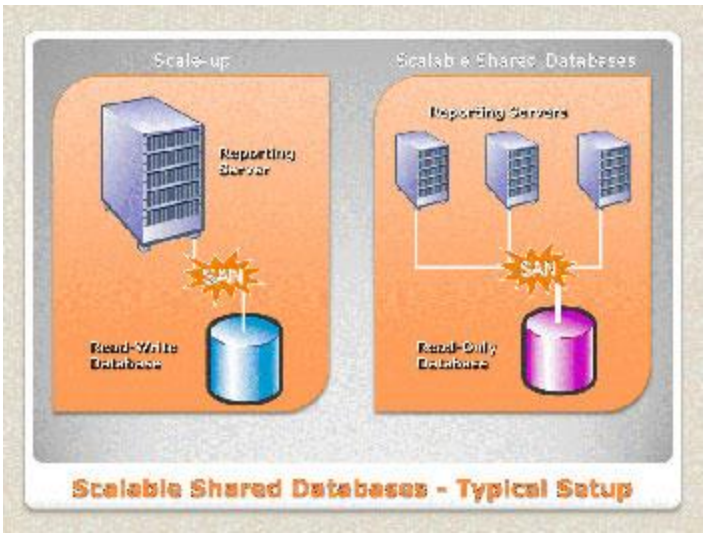
- Through sys.dm_db_missing_index_* collection of DMVs, you can get recommendations on what new indexes could benefit the queries running on your server. The recommendations come with a estimate on how much improvement you can expect from the new index.
- If you'd like to automate creation and dropping of indexes, SQL Server Query Optimization Team has blogged about how to automate index recommendations into actions: http://blogs.msdn.com/queryoptteam/archive/2006/06/01/613516.aspx
-

Figuring out the most popular queries in seconds

- Another great DMV that can help save you a lot of work is sys.dm_exec_query_stats. In previous version of SQL Server to find out the highest impact queries on CPU or IO in system, you had to walk through a long set of analyses steps including getting aggregated information out of the data you collected from profiler.
- With sys.dm_exec_query_stats, you can figure out many combinations of query analyses by a single query. Here are some of the examples;
- Find queries suffering most from blocking – (total_elapsed_time – total_worker_time)
- Find queries with most CPU cycles – (total_worker_time)
- Find queries with most IO cycles – (total_physical_reads + total_logical_reads + total_logical_writes)
- Find most frequently executed queries – (execution_count)
- You can find more information on how to use dynamic management views for performance troubleshooting in the "SQL Server 2005 Waits and Queues" whitepaper located at: http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/performance_tuning_waits_queues.mspx

Scalable Shared Databases

- Scalable Shared Databases provide an alternative scale out mechanism for Read-Only environments. Through Scalable Shared Databases one can mount the same physical drives on commodity machines and allow multiple instances of SQL Server 2005 to work off of the same set of data files. The setup does not require duplicate storage for every instance of SQL Server and allows additional processing power through multiple SQL Server instances that have their own local resources like cpu, memory, tempdb and potentially other local databases. However this type of setup does limit the IO bandwidth since all instances point to the physical set of files.



See full-sized image.

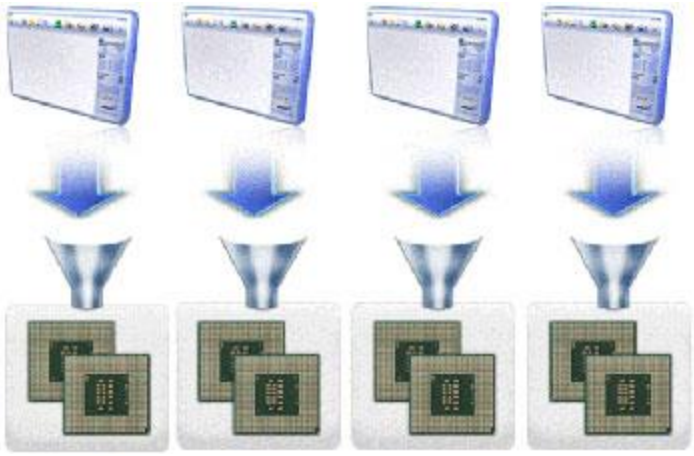
Book Online for SQL server 2005 contains details on Scalable Shared Databases.

Steps to setup:

http://support.microsoft.com/kb/910378

10 Soft-NUMA

- Highly concurrent workloads hit a contention point around global state they maintain at some point. That point in many cases happen to be '8'. One way around this contention has been to eliminate the global state and create hierarchies. NUMA architectures allow us to eliminate the contention around global resources by moving main resources closer to each other and forming nodes. SQL Server 2005 recognizes the NUMA architecture and self manages allocation of resources to adhere and take advantage of the hardware NUMA setup at the time of startup. By aligning with the HW setup SQL Server partitions its internal management to improve throughput.
- Some workloads benefit greatly from the partitioning concept, especially mixed workloads that have to run varying characteristics of data access concurrently (example: OLTP and Reporting). Soft-NUMA allows partitioning configuration to be extended into the software level and defined either on top of a NUMA enabled environment to further divide the hardware partitions into smaller chunks or on a machine that does not utilize NUMA concepts to enable partitioning for the configuration. By configuring partitions through Soft-NUMA, the administrator can control the allocation of schedulers and memory managers for each node and can configure specific TCP/IP ports for the nodes. Then, clients can be configured to connect using the specific ports to access specific partitions.



See full-sized image.

- Soft-NUMA topic is extensively covered in Book Online. You can also read more about the details of Soft-NUMA at Slava Oks's Weblog at http://blogs.msdn.com/slavao/

Top 10 Hidden Gems in SQL 2008 R2

With all the new features in SQL 2008 R2, here are the major ones getting all the press:

PowerPivot
Parallel Data Warehouse
Application and Multi-Server Management
StreamInsight
256 core support

There is so much written on the ones above, I wanted to concentrate on talking about other new features in SQL 2008 R2. So, in no particular order:

1. **SMB support** – SMB stands for Server Message Block and this protocol is now officially supported by SQL Server 2008 R2 and beyond. This improvement has formalized the support status of placing SQL database files on SMB network file shares. From Kevin Farlee, the owner of this feature in SQL Server: "This presents a better-together story with the work that Windows has done in Windows7/Server 2008 R2 to make the Windows SMB stack far more performant and resilient than in the past. It is also a recognition that with the increasing acceptance of iSCSI, customers are viewing Ethernet as a viable way to connect to their storage. Finally, it gives customers in consolidation environments a very simple to manage method for moving databases between servers without investing in a large SAN infrastructure."
2. **Increased Performance** – there are very nice performance improvements, especially with the combination of Windows 2008 R2 and SQL 2008 R2. The actual TPC-E measurements on have been audited and published.
3. **SYSPREP** – Finally! We can now create Sysprep versions of SQL Server environments, starting with SQL Server 2008 R2, but only for the relational engine. My favorite thing about this piece is that it even works with HyperV images containing SQL Server 2008 R2.
4. **Report Builder 3.0 and Reporting Services** – Too many great new features to talk about in a blog and the development team already has a great blog. But my favorite is the Report Part feature where you can take an existing report and designate report items and data regions to save and reuse in other reports. This can amount to a huge time savings for developing new reports. Other customers tell me they like the improved Sharepoint integration and the performance improvements in Sharepoint. But that is not all, there is Bing map support, spark lines, and shared data sets.
5. **Master Data Services** – for data consistency across heterogeneous applications. BOL link.
6. **SSIS - Bulk Inserts with ADO.NET provider** are now possible, which is extremely nice because it used to do it a row at a time. Now, if you check the box to "Use Bulk Insert when possible" then you can see vastly improved performance when it kicks in.
7. **Setup** – integrated Sharepoint mode setup is vastly improved for both Reporting Services and Analysis Services. See the link for Powerpivot for Sharepoint to get the instructions.
8. **Excel 2010** – new additions for databases: slicers, data cleansing, AJAX data feeds, Odata feeds and named set improvements. To create a Named Set in Excel, once you've created a PivotTable against an OLAP source go to the Options tab under PivotTable Tools, and select "Fields, Items, & Sets" à "Manage Sets" à "New…" à "Create Set Using MDX…". Another of my new favorites within PowerPivot in Excel is the new Data Cleansing ribbon. This allows the users to do their own clean

up, which will be essential when they combine data from disparate sources. And you can also get an OData feed from a Reporting Services report.

9. **Database Compression** - now supports Unicode. If you have Unicode date types, like nchar and nvarchar, but the data contained within is normally single byte character sets, you will see significant space savings.
10. **PHP 5 Driver** – Version 1.1 of the PHP 5 driver has a list of new capabilities, allowing access to SQL Server 2005 and SQL Server 2008.

New and changed Editions (more details and pricing)

- Data Center Edition – needed for machine with more than 8 physical CPU sockets plus other improvements needed for the top SQL Server projects.
- Parallel Data Warehouse Edition – Massively Parallel Processing (MPP) Edition of SQL Server targeted at data warehouses in the 10's to 100's of terabytes. It is an appliance where you order the hardware and software together and it comes preinstalled and preconfigured. The minimum installation is one rack so no, you cannot install it on your laptop to play with it.
- Standard Edition – now has the capability to do backup compression.

Top 10 SQL Server 2008 Features for the Database Administrator (DBA)

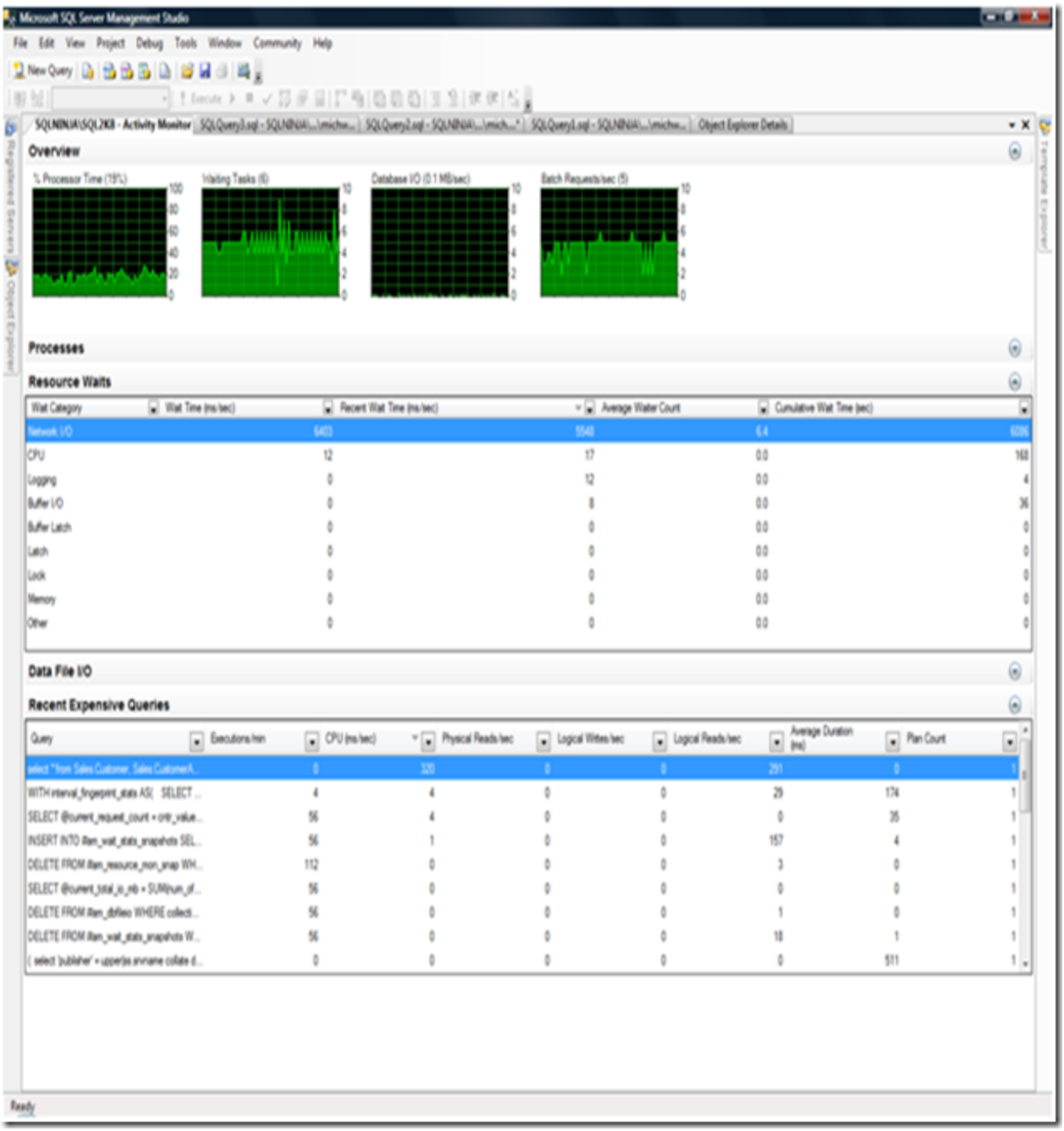
Microsoft SQL Server 2008 provides a number of enhancements and new functionality, building on previous versions. Administration, database maintenance, manageability, availability, security, and performance, among others, all fall into the roles and responsibilities of the database administrator. This article provides the top ten new features of SQL Server 2008 (referenced in alphabetical order) that can help DBAs fulfill their responsibilities. In addition to a brief description of each feature, we include how this feature can help and some important use considerations.

1 - Activity Monitor

When troubleshooting a performance issue or monitoring a server in real time, it is common for the DBA to execute a number of scripts or check a number of sources to collect general information about what processes are executing and where the problem may be. SQL Server 2008 Activity Monitor consolidates this information by detailing running and recently executed processes, graphically. The display gives the DBA a high-level view and the ability to drill down on processes and view wait statistics to help understand and resolve problems.

To open up Activity Monitor, just right-click on the registered server name in Object Explorer and then click **Activity Monitor,** or utilize the standard toolbar icon in SQL Server Management Studio. Activity Monitor provides the DBA with an overview section producing output similar to Windows Task Manager and drilldown components to look at specific processes, resource waits, data file I/Os, and recent expensive queries, as noted in Figure 1.

Figure 1: Display of SQL Server 2008 Activity Monitor view from Management Studio



NOTE: There is a refresh interval setting accessed by right-clicking on the Activity Monitor. Setting this value to a low threshold, under 10 seconds, in a high-volume production system can impact overall system performance.

DBAs can also use Activity Monitor to perform the following tasks:

- Pause and resume Activity Monitor with a simple right-click. This can help the DBA to “save” a particular point-in-time for further investigation without it being refreshed or overwritten. However, be careful, because if you manually refresh, expand, or collapse a section, the data will be refreshed.

- Right-click a line item to display the full query text or graphical execution plan via Recent Expensive Queries.
- Execute a Profiler trace or kill a process from the Processes view. Profiler events include *RPC:Completed*, *SQL:BatchStarting*, and *SQL:BatchCompleted* events, and *Audit Login* and *Audit Logout*.

Activity Monitor also provides the ability to monitor activity on any SQL Server 2005 instance, local or remote, registered in SQL Server Management Studio.

2- [SQL Server] Audit

Having the ability to monitor and log events, such as who is accessing objects, what changes occurred, and what time changes occurred, can help the DBA to meet compliance standards for regulatory or organizational security requirements. Gaining insight into the events occurring within their environment can also help the DBA in creating a risk mitigation plan to keep the environment secure.

Within SQL Server 2008 (Enterprise and Developer editions only), SQL Server Audit provides automation that allows the DBA and others to enable, store, and view audits on various server and database components. The feature allows for auditing at a granularity of the server and/or database level.

There are server-level audit action groups, such as:

- FAILED_LOGIN_GROUP, which tracks failed logins.
- BACKUP_RESTORE_GROUP, which shows when a database was backed up or restored.
- DATABASE_CHANGE_GROUP, which audits when a database is created, altered, or dropped.

Database-level audit action groups include:

- DATABASE_OBJECT_ACCESS_GROUP, which is raised whenever a CREATE, ALTER, or DROP statement is executed on database objects.
- DATABASE_OBJECT_PERMISSION_CHANGE_GROUP, which is raised when GRANT, REVOKE, or DENY is utilized for database objects.

There are also audit actions, such as SELECT, DELETE, or EXECUTE. For more information, including a full list of the audit groups and actions, see SQL Server Audit Action Groups and Actions.

Audit results can be sent to a file or event log (Windows Security or System) for viewing. Audit information is created utilizing *Extended Events*, another new SQL Server 2008 feature.

By using SQL Server 2008 audits, the DBA can now answer questions that were previously very difficult to retroactively determine, such as “Who dropped this index?”, “When was the stored procedure modified?”, “What changed which might not be allowing this user to access this table?”, or even “Who ran SELECT or UPDATE statements against the **[dbo.Payroll]** table?”

For more information about using SQL Server Audit and some examples of implementation, see the SQL Server 2008 Compliance Guide.

3 - Backup Compression

This feature has long been a popular request of DBAs for SQL Server. The wait is finally over, and just in time! Many factors, including increased data retention periods and the need to physically store more data have contributed to the recent explosion in database size. Backing up a large database can require a significant time window to be allotted to backup operations and a large amount of disk space allocated for use by the backup file(s).

With SQL Server 2008 backup compression, the backup file is compressed as it is written out, thereby requiring less storage, less disk I/O, and less time. In lab tests conducted with real customer data, we observed in many cases a reduction in the backup file size between 70% and 85%. Testing also revealed around a 45% reduction in the backup and restore time. It is important to note that the additional processing results in higher processor utilization. To help segregate the CPU intensive backup and minimize its effect on other processes, one might consider utilizing another feature mentioned in this paper, *Resource Governor*.

The compression is achieved by specifying the WITH COMPRESSION clause in the BACKUP command (for more information, see SQL Server Books Online) or by selecting it in the **Options** page in the **Back Up Database** dialog box. To prevent having to modify all existing backup scripts, there is also a global setting to enable compressing all backups taken on a server instance by default. (This setting is accessed by using the **Database Settings** page of the **Server Properties** dialog box or by running **sp_configure** with **backup compression default** set to 1.) While the compression option on the backup command needs to be explicitly specified, the restore command automatically detects that a backup is compressed and decompresses it during the restore operation.

Backup compression is a very useful feature that can help the DBA save space and time. For more information about tuning backup compression, see the technical note on Tuning the Performance of Backup Compression in SQL Server 2008. NOTE: Creating compressed backups is only supported in SQL Server 2008 Enterprise and Developer editions; however, every SQL Server 2008 edition allows for a compressed backup to be restored.

4 - Central Management Servers

DBAs are frequently responsible for managing not one but many SQL Server instances in their environment. Having the ability to centralize the management and administration of a number of SQL Server instances from a single source can allow the DBA to save significant time and effort. The Central Management Servers implementation, which is accessed via the Registered Servers component in SQL Server Management Studio, allows the DBA to perform a number of administrative tasks on SQL Servers within the environment, from a single management console.

Central Management Servers allow the DBA to register a group of servers and apply functionality to the servers, as a group, such as:

- Multiserver query execution: A script can now be executed from one source, across multiple SQL Servers, and be returned to that source, without the need to distinctly log into every server. This can be extremely helpful in cases where data from tables on two or more SQL Servers needs to be viewed or compared without the execution of a distributed query. Also, as long as the syntax is supported in earlier server versions, a query executed from the Query Editor in SQL Server 2008 can run against SQL Server 2005 and SQL Server 2000

instances as well. For more information, see the SQL Server Manageability Team Blog, specifically Multiple Server Query Execution in SQL Server 2008 .

- Import and evaluate policies across servers: As part of *Policy-Based Management* (another new SQL Server 2008 feature discussed in this article), SQL Server 2008 provides the ability to import policy files into particular Central Management Server Groups and allows policies to be evaluated across all of the servers registered in the group
- Control Services and bring up SQL Server Configuration Manager: Central Management Servers help provide a central place where DBAs can view service status and even change status for the services, assuming they have the appropriate permissions
- Import and export the registered servers: Servers within Central Management Servers can be exported and imported for use between DBAs or different SQL Server Management Studio instance installations. This is an alternative to DBAs importing or exporting into their own local groupings within SQL Server Management Studio.

Be aware that permissions are enforced via Windows authentication, so a user might have different rights and permissions depending on the server registered within the Central Management Server group. For more information, see Administering Multiple Servers Using Central Management Servers and a Kimberly Tripp blog: *SQL Server 2008 Central Management Servers-have you seen these?*

5 - Data Collector and Management Data Warehouse

Performance tuning and troubleshooting are a time-consuming tasks that can require in-depth SQL Server skills and an understanding of database internals. Windows System monitor (Perfmon), SQL Server Profiler, and dynamic management views (DMVs) helped with some of this, but they were often intrusive, laborious to use, or the dispersed data collection methods were cumbersome to easily summarize and interpret.

To provide actionable performance insight, SQL Server 2008 delivers a fully extensible performance data collection and warehouse tool also known as the data collector. The tool includes several out-of-the-box data collection agents, a centralized data repository for storing performance data called management data warehouse, and several precanned reports to present the captured data. The data collector is a scalable tool that can collect and assimilate data from multiple sources such as dynamic management views , Perfmon, Transact-SQL queries, by using a fully customizable data collection frequency. The data collector can be extended to collect data for any measurable attribute of an application.

Another helpful feature of the management data warehouse is that it can be installed on any SQL Server and then collect data from one or more SQL Server instances within the environment. This can help minimize the performance impact on production systems and improve the scalability in terms of monitoring and collecting data from a number of servers. In lab testing we observed around a 4% reduction in throughput when running the agents and the management data warehouse on a server running at capacity (via an OLTP workload). The impact can vary based on the collection interval (as the test was over an extended workload with 15-minute-pulls into the warehouse), and it can be exacerbated during intervals of data collection. Finally, some capacity should be considered, because the DCExec.exe process will take up some memory and processor resources, and writes to the management data warehouse will increase the I/O workload and space allocation needed where the data and log files are located.

The diagram (Figure 2) below depicts a typical data collector report.

Figure 2: Display of SQL Server 2008 Data Collector Report

This report shows SQL Server processing over the period of time data was collected. Events such as waits, CPU, I/O, memory usage, and expensive query statistics are collected and displayed. A DBA can also drill down into the reports to focus on a particular query or operation to further investigate, detect, and resolve performance problems. This data collection, storage, and reporting can allow the DBA to establish proactive monitoring of the SQL Server(s) in the

environment and go back over time to understand and assess changes to performance over the time period monitored. The data collector and management data warehouse feature is supported in all editions (except SQL Server Express) of SQL Server 2008.

6 - Data Compression

The ability to easily manage a database can greatly enhance the opportunity for DBAs to accomplish their regular task lists. As table, index, and file sizes grow and very large databases (VLDBs) become commonplace, the management of data and unwieldy file sizes has become a growing pain point. Also, with more data being queried, the need for large amounts of memory or the necessity to do physical I/O can place a larger burden on DBAs and their organizations. Many times this results in DBAs and organizations securing servers with more memory and/or I/O bandwidth or having to pay a performance penalty.

Data compression, introduced in SQL Server 2008, provides a resolution to help address these problems. Using this feature, a DBA can selectively compress any table, table partition, or index, resulting in a smaller on-disk footprint, smaller memory working-set size, and reduced I/O. The act of compression and decompression will impact CPU; however, this impact is in many cases offset by the gains in I/O savings. Configurations that are bottlenecked on I/O can also see an increase in performance due to compression.

In some lab tests, enabling data compression resulted in a 50-80% saving in disk space. The space savings did vary significantly with minimal savings on data that did not contain many repeating values or where the values required all the bytes allocated by the specified data type. There were also workloads that did not show any gains in performance. However, on data that contained a lot of numeric data and many repeating values, we saw significant space savings and observed performance increases from a few percentage points up to 40-60% on some sample query workloads.

SQL Server 2008 supports two types of compressions: row compression, which compresses the individual columns of a table, and page compression, which compresses data pages using row, prefix, and dictionary compression. The amount of compression achieved is highly dependent on the data types and data contained in the database. In general we have observed that using row compression results in lower overhead on the application throughput but saves less space. Page compression, on the other hand, has a higher impact on application throughput and processor utilization, but it results in much larger space savings. Page compression is a superset of row compression, implying that an object or partition of an object that is compressed using page compression also has row compression applied to it. Also, SQL Server 2008 does support the **vardecimal** storage format of SQL Server 2005 SP2. However, because this storage format is a subset of row compression, it is a depreciated feature and will be removed from future product versions.

Both row and page compression can be applied to a table or index in an online mode that is without any interruption to the application availability. However, a single partition of a partitioned table cannot be compressed or uncompressed online. In our testing we found that using a hybrid approach, where only the largest few tables were compressed, resulted in the best performance in terms of saving significant disk space while having a minimal negative impact on performance. Because there are disk space requirements, similar to what would be needed to create or rebuild an index, care should be taken in implementing compression as well. We also found that compressing the smallest objects first, from the list of objects you desire to compress, minimized the need for additional disk space during the compression process.

Data compression can be implemented via Transact-SQL or the Data Compression Wizard. To determine how compressing an object will affect its size, you can use the **sp_estimate_data_compression_savings** system stored procedure or the Data Compression Wizard to calculate the estimated space savings. Database compression is only supported in SQL Server 2008 Enterprise and Developer editions. It is implemented entirely within the database and does not require any application modification.

For more information about using compression, see Creating Compressed Tables and Indexes.

7 - Policy-Based Management

In a number of business scenarios, there is a need to maintain certain configurations or enforce policies either within a specific SQL Server, or many times across a group of SQL Servers. A DBA or organization may require a particular naming convention to be implemented on all user tables or stored procedures that are created, or a required configuration change to be defined across a number of servers in the same manner.

Policy-Based Management (PBM) provides DBAs with a wide variety of options in managing their environment. Policies can be created and checked for compliance. If a target (such as a SQL Server database engine, a database, a table, or an index) is out of compliance, the administrator can automatically reconfigure it to be in compliance. There are also a number of evaluation modes (of which many are automated) that can help the DBA check for policy compliance, log and notify when a policy violation occurs, and even roll back the change to keep in compliance with the policy. For more information about evaluation modes and how they are mapped to facets (a PBM term also discussed in the blog), see the SQL Server Policy-Based Management blog.

The policies can be exported and imported as .xml files for evaluation and implementation across multiple server instances. Also, in SQL Server Management Studio and the Registered Servers view, policies can be evaluated across multiple servers if they are registered under a local server group or a Central Management Server group.

Not all of the functionality of Policy-Based Management can be implemented on earlier versions of SQL Server. However, the policy *reporting* feature can be utilized on SQL Server 2005 and SQL Server 2000. For more information about administering servers by using Policy-Based Management, see Administering Servers by Using Policy-Based Management in SQL Server Books Online. For more information about the technology itself, including examples, see the SQL Server 2008 Compliance Guide.

8 - Predictable Performance and Concurrency

A significant problem many DBAs face is trying to support SQL Servers with ever-changing workloads, and achieving some level of predictable performance (or minimizing variance in plans and performance). Unexpected query performance, plan changes, and/or general performance issues can come about due to a number of factors, including increased application load running against SQL Server or version upgrades of the database itself. Getting predictable performance from queries or operations run against SQL Server can greatly enhance the DBAs ability to meet and maintain availability, performance, and/or business continuity goals (OLAs or SLAs).

SQL Server 2008 provides a few feature changes that can help provide more predictable performance. In SQL Server 2008, there exist some enhancements to the SQL Server 2005 plan guides (or *plan freezing*) and a new option to control lock escalation at a table level. Both of these enhancements can provide a more predictable and structured interaction between the application and the database.

First, plan guides:

SQL Server 2005 enabled greater query performance stability and predictability by providing a new feature called plan guides to enable specifying hints for queries that could not be modified directly in the application. For more information, see the Forcing Query Plans white paper. While a very powerful feature, the USE PLAN query hint only supported SELECT DML operations and were often cumbersome to use due to the sensitivity of the plan guides to the formatting.

SQL Server 2008 builds on the plan guides mechanism in two ways: It expands the support for the USE PLAN query hint to cover all DML statements (INSERT, UPDATE, DELETE, MERGE), and it introduces a new *plan freezing* feature that can be used to directly create a plan guide (freeze) any query plan that exists in the SQL Server plan cache, as in the following example.

```
sp_create_plan_guide_from_handle

@name = N'MyQueryPlan',

@plan_handle = @plan_handle,

@statement_start_offset = @offset;
```

A plan guide created by either means has a database scope and is stored in the **sys.plan_guides** table. Plan guides are only used to influence the query plan selection process of the optimizer and do not eliminate the need for the query to be compiled. A new function, *sys.fn_validate_plan_guide*, has also been introduced to validate existing SQL Server 2005 plan guides and ensure their compatibility with SQL Server 2008. Plan freezing is available in the SQL Server 2008 Standard, Enterprise, and Developer editions.

Next, lock escalation:

Lock escalation has often caused blocking and sometimes even deadlocking problems, which the DBA is forced to troubleshoot and resolve. Previous versions of SQL Server permitted controlling lock escalation (trace flags 1211 and 1224), but this was only possible at an instance-level granularity. While this helped some applications work-around the problem, it caused severe issues for others. Another problem with the SQL Server 2005 lock escalation algorithm was that locks on partitioned tables were directly escalated to the table level, rather than the partition level.

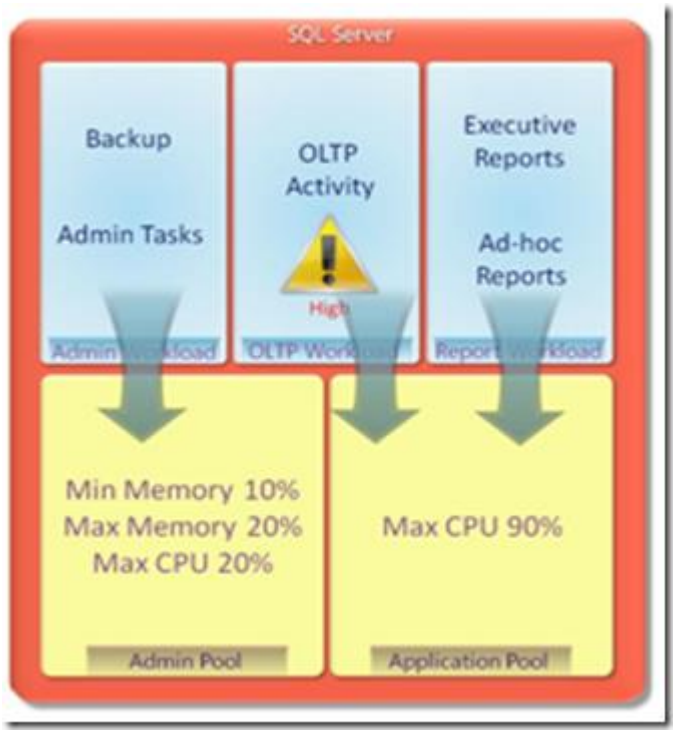
SQL Server 2008 offers a solution for both of these problems. A new option has been introduced to control lock escalation at a table level. By using an ALTER TABLE command, option locks can be specified to not escalate, or escalate to the partition level for partitioned tables. Both these enhancements help improve the scalability and performance without having negative side-effects on other objects in the instance. Lock escalation is specified at the database-object level and does not require any application change. It is supported in all editions of SQL Server 2008.

9 - Resource Governor

Maintaining a consistent level of service by preventing runaway queries and guaranteeing resources for mission-critical workloads has been a challenge. In the past there was no way of guaranteeing a certain amount of resources to a set of queries and prioritizing the access. All queries had equal access to all the available resources.

SQL Server 2008 introduces a new feature called Resource Governor, which helps address this issue by enabling users to differentiate workloads and allocate resources as they are requested. Resource Governor limits can easily be reconfigured in real time with minimal impact on the workloads that are executing. The allocation of the workload to a resource pool is configurable at the connection level, and the process is completely transparent to the application.

The diagram below depicts the resource allocation process. In this scenario three workload pools (Admin Workload, OLTP Workload, and Report Workload) are configured, and the OLTP Workload pool is assigned a high priority. In parallel, two resource pools (Admin Pool and Application Pool) are configured with specific memory and processor (CPU) limits as shown. As a final step the Admin Workload is assigned to the Admin Pool and the OLTP and Report workloads are assigned to the Application Pool.



Below are some other points you need to consider when using Resource Governor.

- Resource Governor relies on login credentials, host name, or application name as a 'resource pool identifier', so using a single login for an application, depending on the number of clients per server, might make creating pools more difficult.

- Database-level object grouping, in which the resource governing is done based on the database objects being referenced, is not supported.
- Resource Governor only allows resource management within a single SQL Server instance. For managing multiple SQL Server instances or processes within a server from a single source, Windows System Resource Manager should be considered.
- Only processor and memory resources can be configured. I/O resources cannot be controlled.
- Dynamically switching workloads between resource pools once a connection is made is not possible.
- Resource Governor is only supported in SQL Server 2008 Enterprise and Developer editions and can only be used for the SQL Server database engine; SQL Server Analysis Services (SSAS), SQL Server Integration Services (SSIS), and SQL Server Reporting Services (SSRS) cannot be controlled.

10 - Transparent Data Encryption (TDE)

Security is one of the top concerns of many organizations. There are many different layers to securing one of the most important assets of an organization: its data. In most cases, organizations do well at securing their active data via the use of physical security, firewalls, and tightly controlled access policies. However, when physical medium such as the backup tape or disk on which the data resides is compromised, the above security measures are of no use, because a rouge user can simply restore the database and get full access to the data.

SQL Server 2008 offers a solution to this problem by way of transparent data encryption (TDE). TDE performs real-time I/O encryption and decryption of the data and log files by using a database encryption key (DEK). The DEK is a symmetric key secured by using a certificate stored in the master database of the server, or an asymmetric key protected by an Extensible Key Management (EKM) module.

TDE is designed to protect data 'at rest', which means the data stored in the .mdf, .ndf, and .ldf files cannot be viewed using a hex editor or other means. However, data that is not at rest, such as the results of a SELECT statement in SQL Server Management Studio, will continue to be visible to users who have rights to view the table. Also, because TDE is implemented at the database level, the database can leverage indexes and keys for query optimization. TDE should not be confused with column-level encryption, which is a separate feature that allows encryption of data even when it is not at rest.

Encrypting a database is a one-time process that can be initiated via a Transact-SQL command or SQL Server Management Studio, and it is executed as a background thread. You can monitor the encryption or decryption status using the *sys.dm_database_encryption_keys* dynamic management view. In a lab test we conducted, we were able to encrypt a 100 GB database using the AES_128 encryption algorithm in about an hour. While the overhead of using TDE is largely dictated by the application workload, in some of the testing conducted that overhead was measured to be less than 5%. One potential performance impact to be aware of is this: If any database within the instance does have TDE applied, the **tempDB** system database is also encrypted. Finally, of note when combining features:

- When backup compression is used to compress an encrypted database, the size of the compressed backup is larger than if the database were not encrypted, because encrypted data does not compress well.
- Encrypting the database does not affect data compression (row or page).

TDE enables organizations to meet the demands of regulatory compliance and overall concern for data privacy. TDE is only supported in the SQL Server 2008 Enterprise and Developer editions and can be enabled without changing

existing applications. For more information, see Database Encryption in SQL Server 2008 Enterprise Edition or the SQL Server 2008 Compliance Guide discussion on *Using Transparent Data Encryption*.

In conclusion, SQL Server 2008 offers features, enhancements, and functionality to help improve the Database Administrator experience. While a Top 10 list was provided above, there are many more features included within SQL Server 2008 that help improve the experience for DBA and other users alike. For a Top 10 feature set for other SQL Server focus areas, see the other SQL Server 2008 Top 10 articles on this site. For a full list of features and detailed descriptions, see SQL Server Books Online and the SQL Server 2008 Overview Web site.

Top 10 SQL Server 2008 Features for ISV Applications

Microsoft® SQL Server® 2008 has hundreds of new and improved features, many of which are specifically designed for large scale independent software vendor (ISV) applications, which need to leverage the power of the underlying database while keeping their code database agnostic. This article presents details of the top 10 features that we believe are most applicable to such applications based on our work with strategic ISV partners. Along with the description of each feature, the main pain-points the feature helps resolve and some of the important limitations that need to be considered are also presented. The features are grouped into two categories: ones that do not require any application change (features 1-8) and those that require some application code change (features 9-10). The features are not prioritized in any particular order.

1 - Data Compression

The disk I/O subsystem is the most common bottleneck for many database implementations. More disks are needed to reduce the read/write latencies; but this is expensive, especially on high-performing storage systems. At the same time, the need for storage space continues to increase due to rapid growth of the data, and so does the cost of managing databases (backup, restore, transfer, etc.).

Data compression introduced in SQL Server 2008 provides a resolution to address all these problems. Using this feature one can selectively compress any table, table partition, or index, resulting in a smaller on-disk footprint, smaller memory working-set size, and reduced I/O. Configurations that are bottlenecked on I/O may also see an increase in performance. In our lab test, enabling data compression for some ISV applications resulted in a 50-80% saving in disk space.

SQL Server supports two types of compressions: ROW compression, which compresses the individual columns of a table, and PAGE compression which compresses data pages using row, prefix, and dictionary compression. The compression results are highly dependent on the data types and data contained in the database; however, in general we've observed that using ROW compression results in lower overhead on the application throughput but saves less space. PAGE compression, on the other hand, has a higher impact on application throughput and processor utilization, but it results in much larger space savings. PAGE compression is a superset of ROW compression, implying that an object or partition of an object that is compressed using PAGE compression also has ROW compression applied to it. Compressed pages remain compressed in memory until rows/columns on the pages are accessed.

Both ROW and PAGE compression can be applied to a table or index in an online mode that is without any interruption to the application availability. However, partitions of a partitioned table cannot be compressed or uncompressed online. In our testing we found that using a hybrid approach where only the largest few tables were compressed resulted in the best overall performance, saving significant disk space while having a minimal negative impact on performance. We also found that compressing the smallest objects first minimized the need for additional disk space during the compression process.

To determine how compressing an object will affect its size you can use the **sp_estimate_data_compression_savings** system stored procedure. Database compression is

only supported in SQL Server 2008 Enterprise and Developer editions. It is fully controlled at the database level and does not require any application change.

2 - Backup Compression

The amount of data stored in databases has grown significantly in the last decade. resulting in larger database sizes. At the same time the demands for applications to be available 24x7 have forced the backup time-windows to shrink. In order to speed up the backup procedure, database backups are usually first streamed to fast disk-based storage and moved out to slower media later. Keeping such large disk-based backups online is expensive, and moving them around is time consuming.

With SQL Server 2008 backup compression, the backup file is compressed as it is written out, thereby requiring less storage, less disk I/O, and less time, and utilizing less network bandwidth for backups that are written out to a remote server. However, the additional processing results in higher processor utilization. In a lab test conducted with an ISV workload we observed a 40% reduction in the backup file size and a 43% reduction in the backup time.The compression is achieved by specifying the WITH COMPRESSION clause in the backup command (for more information, see SQL Server Books Online). To prevent having to modify all the existing backup scripts, there is also a global setting (using the **Database Settings** page of the **Server Properties** dialog box) to enable compression of all backups taken on that server instance by default; this eliminates the need to modify existing backup scripts. While the compression option on the backup command needs to be explicitly specified, the restore command automatically detects that a backup is compressed and decompresses it during the restore operation. Overall, backup compression is a very useful feature that does not require any change to the ISV application. For more information about tuning backup compression, see the technical note on Tuning the Performance of Backup Compression in SQL Server 2008.

Note: Creating compressed backups is only supported in SQL Server 2008 Enterprise and Developer editions; however, every SQL Server 2008 edition can restore a compressed backup.

3 - Transparent Data Encryption

In most cases, organizations do well at securing their active data via the use of firewalls, physical security, and tightly controlled access policies. However, when the physical media such as the backup tape or disk on which the data resides is compromised, the above security measures are of no use, since a rogue user can simply restore the database and get full access to the data.

SQL Server 2008 offers a solution to this problem by way of Transparent Data Encryption (TDE). TDE performs real-time I/O encryption and decryption of the data and log files using a database encryption key (DEK). The DEK is a symmetric key secured by using a certificate stored in the master database of the server, or an asymmetric key protected by an Extensible Key Management (EKM) module.TDE is designed to protect data 'at rest'; this means that the data stored in the .mdf, .ndf, and .ldf files cannot be viewed using a hex editor or some other such means. However, data that is not at rest, such as the results of a select statement in SQL Server Management Studio, continues to be visible to users who have rights to view the table. TDE should not be confused with column-level encryption, which is a separate feature that allows encryption of data even when it is not at rest.Encrypting a database is a one-time process that can be initiated via a Transact-SQL command and is executed as a background

thread. You can monitor the encryption/decryption status using the **sys.dm_database_encryption_keys** dynamic management view (DMV).

In a lab test we conducted we were able to encrypt a 100-gigabyte (GB) database using the AES_128 encryption algorithm in about one hour. While the overheads of using TDE are largely dictated by the application workload, in some of the testing we conducted, the overhead was measured to be less than 5%.

One point worth mentioning is when backup compression is used to compress an encrypted database, the size of the compressed backup is larger than if the database were not encrypted; this is because encrypted data does not compress well.

TDE enables organizations to meet the demands of regulatory compliance and overall concern for data privacy.

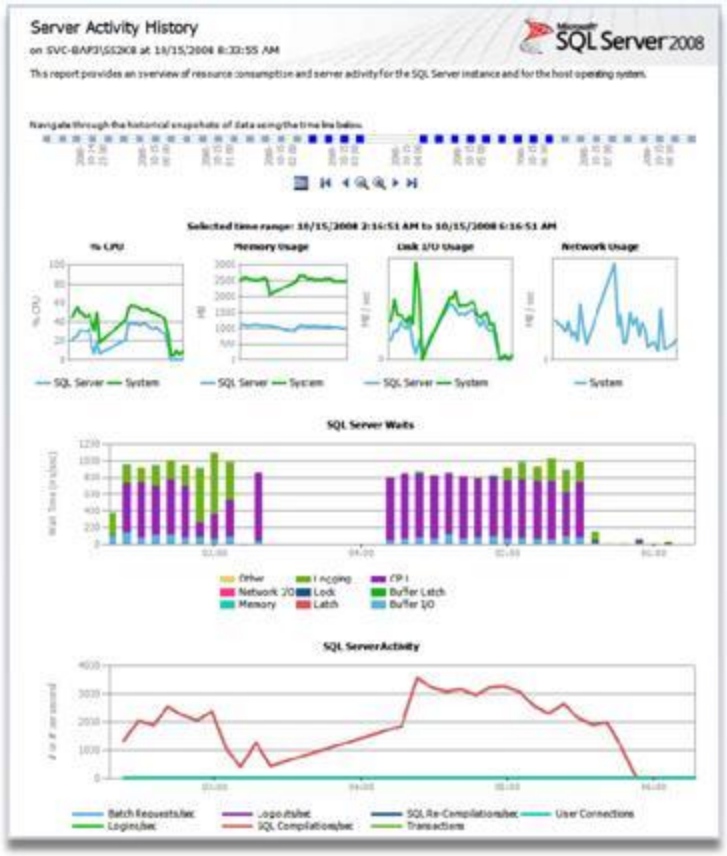
TDE is only supported in the SQL Server 2008 Enterprise and Developer editions, and it can be enabled without changing an existing application.

4 - Data Collector and Management Data Warehouse

Performance tuning and troubleshooting is a time-consuming task that usually requires deep SQL Server skills and an understanding of the database internals. Windows® System monitor (Perfmon), SQL Server Profiler, and dynamic management views helped with some of this, but they were often too intrusive or laborious to use, or the data was too difficult to interpret.

To provide actionable performance insights, SQL Server 2008 delivers a fully extensible performance data collection and warehouse tool also known as the Data Collector. The tool includes several out-of-the-box data collection agents, a centralized data repository for storing performance data called management data warehouse (MDW), and several precanned reports to present the captured data. The Data Collector is a scalable tool that can collect and assimilate data from multiple sources such as dynamic management views, Perfmon, Transact-SQL queries, etc., using a fully customizable data collection and assimilation frequency. The Data Collector can be extended to collect data for any measurable attribute of an application. For example, in our lab test we wrote a custom Data Collector agent job (40 lines of code) to measure the processing throughput of the workload.

The diagram below depicts a typical Data Collector report.



The Performance data collection and warehouse feature is supported in all editions of SQL Server 2008.

5 - Lock Escalation

Lock escalation has often caused blocking and sometimes even deadlocking problems for many ISV applications. Previous versions of SQL Server permitted controlling lock escalation (trace flags 1211 and 1224), but this was only possible at an instance-level granularity. While this helped some applications work around the problem, it caused severe issues for others. Another problem with the SQL Server 2005 lock escalation algorithm was that locks on partitioned tables were directly escalated to the table level, rather than the partition level.

SQL Server 2008 offers a solution for both these issues. A new option has been introduced to control lock escalation at a table level. If an ALTER TABLE command option is used, locks can be specified to not escalate, or to escalate to the partition level for partitioned tables. Both these enhancements help improve the scalability and performance without having negative side-effects on other objects in the instance. Lock escalation is specified at the database-object level and does not require any application change. It is supported in all editions of SQL Server 2008.

6 - Plan Freezing

SQL Server 2005 enabled greater query performance stability and predictability by providing a new feature called plan guides to enable specifying hints for queries that could not be modified directly in

the application (for more information, see the white paper Forcing Query Plans). While a very powerful feature, plan guides were often cumbersome to use due to the sensitivity of the plan guides to the formatting, and only supported SELECT DML operations when used in conjunction the USE PLAN query hint.

SQL Server 2008 builds on the plan guides mechanism in two ways: it expands the support for plan guides to cover all DML statements (INSERT, UPDATE, DELETE, MERGE), and introduces a new feature, *Plan Freezing,* that can be used to directly create a plan guide (freeze) for any query plan that exists in the SQL Server plan cache, for example:

```
sp_create_plan_guide_from_handle

@name =  N'MyQueryPlan',

@plan_handle = @plan_handle,

@statement_start_offset = @offset;
```

A plan guide created by either means have a database scope and are stored in the **sys.plan_guides** table. Plan guides are only used to influence the query plan selection process of the optimizer and do not eliminate the need for the query to be compiled. A new function **sys.fn_validate_plan_guide** has also been introduced to validate existing SQL Server 2005 plan guides and ensure their compatibility with SQL Server 2008. Plan freezing is available in the SQL Server 2008 Standard, Enterprise, and Developer editions.

7 - Optimize for Ad hoc Workloads Option

Applications that execute many single use ad hoc batches (e.g., nonparameterized workloads) can cause the plan cache to grow excessively large and result in reduced efficiency. SQL Server 2005 offered the Parameterization Forced database option to address such scenarios, but that sometimes resulted in adverse side-effects on workloads that had a large skew in the data and had queries that were very sensitive to the underlying data.

SQL Server 2008 introduces a new option, **optimize for ad hoc workloads**, which is used to improve the efficiency of the plan cache. When this option is set to 1, the SQL Server engine stores a small stub for the compiled ad hoc plan in the plan cache instead of the entire compiled plan, when a batch is compiled for the first time. The compiled plan stub is used to identify that the ad hoc batch has been compiled before but has only stored a compiled plan stub, so that when this batch is invoked again the database engine compiles the batch, removes the compiled plan stub from the plan cache, and replaces it with the full compiled plan.

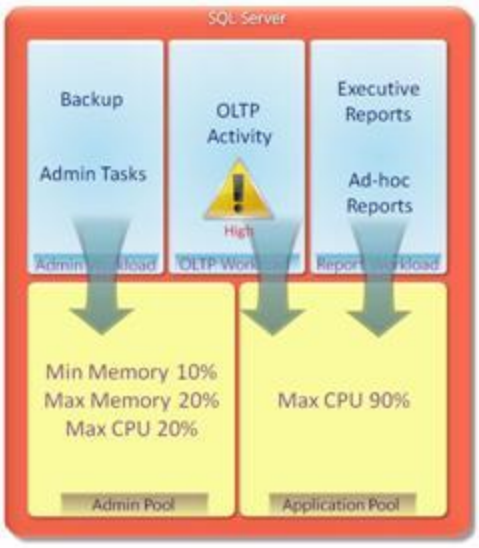
This mechanism helps to relieve memory pressure by not allowing the plan cache to become filled with large compiled plans that are not reused. Unlike the Forced Parameterization option, optimizing for ad hoc workloads does not parameterize the query plan and therefore does not result in saving any processor cycles by way of eliminating compilations. This option does not require any application change and is available in all editions of SQL Server 2008.

8 - Resource Governor

Maintaining a consistent level of service by preventing runaway queries and guaranteeing resources for mission-critical workloads has been a challenge for SQL Server. In the past there was no way of guaranteeing a certain amount of resources to a set of queries and prioritizing the access; all queries had equal access to all the available resources.

SQL Server 2008 introduces a new feature, Resource Governor, which helps address this issue by enabling users to differentiate workloads and allocate resources as they are requested. The Resource Governor limits can easily be reconfigured in real time with minimal impact on the workloads that are executing. The allocation of the workload to a resource pool is configurable at the connection level, and the process is completely transparent to the application.

The diagram below depicts the resource allocation process. In this scenario three workload pools (Admin workload, OLTP workload, and Report workload) are configured, and the OLTP workload pool is assigned a high priority. In parallel two resource pools (Admin pool and Application pool) are configured with specific memory and processor (CPU) limits as shown. As final steps, the Admin workload is assigned to the Admin pool, and the OLTP and Report workloads are assigned to the Application pool.



Below are some other points you need to consider when using resource governor:

- Since Resource Governor relies on login credentials, host name, or application name as a resource pool identifier, most ISV applications that use a single login to connect multiple application users to SQL Server will not be able to use Resource Governor without reworking the application. This rework would require the application to utilize one of the resource identifiers from within the application to help differentiate the workload.
- Database-level object grouping, in which the resource governing is done based on the database objects being referenced, is not supported.
- Resource Governor only allows resource management within a single SQL Server instance. For multiple instances. Windows System Resource Manager should be considered.

- Only processor and memory resources can be configured. I/O resource cannot be controlled.
- Dynamically switching workloads between resource pools once a connection is made is not possible.
- Resource Governor is only supported in SQL Server 2008 Enterprise and Developer editions and can only be used for the SQL Server database engine; SQL Server Analysis Services (SSAS), SQL Server Integration Services (SSIS), and SQL Server Reporting Services (SSRS) cannot be controlled.

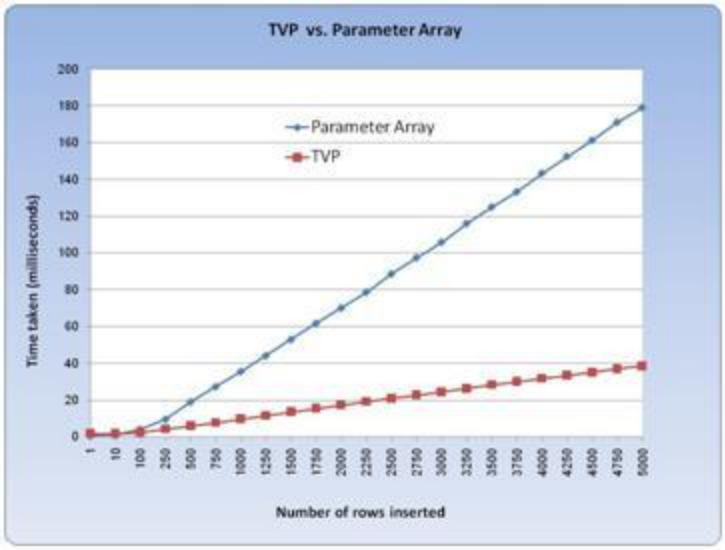
9 - Table-Valued Parameters

Often one of the biggest problems ISVs encountered while developing applications on earlier versions of SQL Server was the lack of an easy way to execute a set of UPDATE, DELETE, INSERT operations from a client as a single batch on the server. Executing the set of statements as singleton operations resulted in a round trip from the client to the server for each operation and could result in as much as a 3x slowdown in performance.

SQL Server 2008 introduces the table-valued parameter (TVP) feature, which helps resolve this problem. Using the new TVP data type, a client application can pass a potentially unlimited sized array of data directly from the client to the server in a single-batch operation. TVPs are first-class data types and are fully supported by the SQL Server tools and SQL Server 2008 client libraries (SNAC 10 or later). TVPs are read-only, implying that they can only be used to pass array-type data into SQL Server; they cannot be used to return array-type data.

The graph below plots the performance of executing a batch of insert statements using a parameter array (sequence of singleton operations) vs. executing the same batch using a TVP. For batches of 10 statements or less, parameter arrays perform better than TVPs. This is due to the one-time overhead associated with initiating the TVP, which outweighs the benefits of transferring and executing the inserts as a single batch on the server.

However, for batches larger than 10 statements, TVPs outperform parameter arrays, because the entire batch is transferred to the server and executed as a single operation. As can be seen in the graph for a batch of 250 inserts the amount of time taken to execute the batch is 2.5 times more when the operations are performed using a parameter array versus a TVP. The performance benefits scale almost linearly and when the size of the batch increases to 2,000 insert statements, executing the batch using a parameter array takes more than four times longer than using a TVP.



TVPs can also be used to perform other functions such as passing a large batch of parameters to a stored procedure. TVPs are supported in all editions of SQL Server 2008 and require the application to be modified.

10 - Filestream

In recent years there has been an increase in the amount of unstructured data (e-mail messages, documents, images, videos, etc.) created. This unstructured data is often stored outside the database, separate from its structured metadata. This separation can cause challenges and complexities in keeping the data consistent, managing the data, and performing backup/restores.

The new Filestream data type in SQL Server 2008 allows large unstructured data to be stored as files on the file system. Transact-SQL statements can be used to read, insert, update and manage the Filestream data, while Win32® file system interfaces can be used to provide streaming access to the data. Using the NTFS streaming APIs allows efficient performance of common file operations while providing all of the rich database services, including security and backup. In our lab tests we observed the biggest performance advantage of streaming access when the size of binary large objects (BLOBs) was greater than 256 kilobytes (KB). The Filestream feature is initially targeted to objects that do not need to be updated in place, as that is not yet supported.

Filestream is not automatically enabled when you install or upgrade SQL Server 2008. You need to enable it by using SQL Server Configuration Manager and SQL Server Management Studio. Filestream requires a special dedicated filegroup to be created to store the Filestream (**varbinary(max)**) data that has been qualified with the Filestream attribute. This filegroup points to an NTFS directory on a file system and is created similar to all the other filegroups. The Filestream feature is supported in all editions of SQL Server 2008, and it requires the application to be modified to leverage the Win32 APIs (if required) and to migrate the existing varbinary data.

SQL Server 2008 is a significant release that delivers many new features and key improvements, many of which have been designed specifically for ISV workloads and require zero or minimal application change.

This article presented an overview of only the top-10 features that are most applicable to ISV applications and help resolve key ISV problems that couldn't easily be addressed in the past. For more information, including a full list of features and detailed descriptions, see SQL Server Books Online and the SQL Server web site.

Top 10 Best Practices for Building a Large Scale Relational Data Warehouse

Building a large scale relational data warehouse is a complex task. This article describes some design techniques that can help in architecting an efficient large scale relational data warehouse with SQL Server. Most large scale data warehouses use table and index partitioning, and therefore, many of the recommendations here involve partitioning. Most of these tips are based on experiences building large data warehouses on SQL Server 2005.

1 - Consider partitioning large fact tables

- Consider partitioning fact tables that are 50 to 100GB or larger.
- Partitioning can provide manageability and often performance benefits.
    - Faster, more granular index maintenance.
    - More flexible backup / restore options.
    - Faster data loading and deleting
- Faster queries when restricted to a single partition..
- Typically partition the fact table on the date key.
    - Enables sliding window.
- Enables partition elimination.

2- Build clustered index on the date key of the fact table

- This supports efficient queries to populate cubes or retrieve a historical data slice.
- If you load data in a batch window then use the options ALLOW_ROW_LOCKS = OFF and ALLOW_PAGE_LOCKS = OFF for the clustered index on the fact table. This helps speed up table scan operations during query time and helps avoid excessive locking activity during large updates.
- Build nonclustered indexes for each foreign key. This helps 'pinpoint queries' to extract rows based on a selective dimension predicate.Use filegroups for administration requirements such as backup / restore, partial database availability, etc.

3 - Choose partition grain carefully

- Most customers use month, quarter, or year.
- For efficient deletes, you must delete one full partition at a time.
- It is faster to load a complete partition at a time.
    - Daily partitions for daily loads may be an attractive option.
    - However, keep in mind that a table can have a maximum of 1000 partitions.
- Partition grain affects query parallelism.
    - For SQL Server 2005:
        - Queries touching a single partition can parallelize up to MAXDOP (maximum degree of parallelism).
        - Queries touching multiple partitions use one thread per partition up to MAXDOP.
    - For SQL Server 2008:
        - Parallel threads up to MAXDOP are distributed proportionally to scan partitions, and multiple threads per partition may be used even when several partitions must be scanned.

- Avoid a partition design where only 2 or 3 partitions are touched by frequent queries, if you need MAXDOP parallelism (assuming MAXDOP =4 or larger).

4 - Design dimension tables appropriately

- Use integer surrogate keys for all dimensions, other than the Date dimension. Use the smallest possible integer for the dimension surrogate keys. This helps to keep fact table narrow.
- Use a meaningful date key of integer type derivable from the DATETIME data type (for example: 20060215).
    - Don't use a surrogate Key for the Date dimension
    - Easy to write queries that put a WHERE clause on this column, which will allow partition elimination of the fact table.
- Build a clustered index on the surrogate key for each dimension table, and build a non-clustered index on the Business Key (potentially combined with a row-effective-date) to support surrogate key lookups during loads.
- Build nonclustered indexes on other frequently searched dimension columns.
- Avoid partitioning dimension tables.
- Avoid enforcing foreign key relationships between the fact and the dimension tables, to allow faster data loads. You can create foreign key constraints with NOCHECK to document the relationships; but don't enforce them. Ensure data integrity though Transform Lookups, or perform the data integrity checks at the source of the data.

5 - Write effective queries for partition elimination

- Whenever possible, place a query predicate (WHERE condition) directly on the partitioning key (Date dimension key) of the fact table.

6 - Use Sliding Window technique to maintain data

- Maintain a rolling time window for online access to the fact tables. Load newest data, unload oldest data.
- Always keep empty partitions at both ends of the partition range to guarantee that the partition split (before loading new data) and partition merge (after unloading old data) do not incur any data movement.
- Avoid split or merge of populated partitions. Splitting or merging populated partitions can be extremely inefficient, as this may cause as much as 4 times more log generation, and also cause severe locking.
- Create the load staging table in the same filegroup as the partition you are loading.
- Create the unload staging table in the same filegroup as the partition you are deleteing.
- It is fastest to load newest full partition at one time, but only possible when partition size is equal to the data load frequency (for example, you have one partition per day, and you load data once per day).
- If the partition size doesn't match the data load frequency, incrementally load the latest partition.
- Various options for loading bulk data into a partitioned table are discussed in the whitepaper http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/loading_bulk_data_partitioned_table.mspx.
- Always unload one partition at a time.

7- Efficiently load the initial data

- Use SIMPLE or BULK LOGGED recovery model during the initial data load.
- Create the partitioned fact table with the Clustered index.

- Create non-indexed staging tables for each partition, and separate source data files for populating each partition.
- Populate the staging tables in parallel.
    - Use multiple BULK INSERT, BCP or SSIS tasks.
        - Create as many load scripts to run in parallel as there are CPUs, if there is no IO bottleneck. If IO bandwidth is limited, use fewer scripts in parallel.
        - Use 0 batch size in the load.
        - Use 0 commit size in the load.
        - Use TABLOCK.
        - Use BULK INSERT if the sources are flat files on the same server. Use BCP or SSIS if data is being pushed from remote machines.
- Build a clustered index on each staging table, then create appropriate CHECK constraints.
- SWITCH all partitions into the partitioned table.
- Build nonclustered indexes on the partitioned table.
- Possible to load 1 TB in under an hour on a 64-CPU server with a SAN capable of 14 GB/Sec throughput (non-indexed table). Refer to SQLCAT blog entry http://blogs.msdn.com/sqlcat/archive/2006/05/19/602142.aspx for details.

8 - Efficiently delete old data

- Use partition switching whenever possible.
- To delete millions of rows from nonpartitioned, indexed tables
    - Avoid DELETE FROM ...WHERE ...
        - Huge locking and logging issues
        - Long rollback if the delete is canceled
    - Usually faster to
        - INSERT the records to keep into a non-indexed table
        - Create index(es) on the table
        - Rename the new table to replace the original
- As an alternative, 'trickle' deletes using the following repeatedly in a loop

```
DELETE TOP (1000) ... ;

COMMIT
```

- Another alternative is to update the row to mark as deleted, then delete later during non critical time.

9 - Manage statistics manually

- Statistics on partitioned tables are maintained for the table as a whole.
- Manually update statistics on large fact tables after loading new data.
- Manually update statistics after rebuilding index on a partition.
- If you regularly update statistics after periodic loads, you may turn off autostats on that table.
- This is important for optimizing queries that may need to read only the newest data.
- Updating statistics on small dimension tables after incremental loads may also help performance. Use FULLSCAN option on update statistics on dimension tables for more accurate query plans.

10 - Consider efficient backup strategies

- Backing up the entire database may take significant amount of time for a very large database.
    - For example, backing up a 2 TB database to a 10-spindle RAID-5 disk on a SAN may take 2 hours (at the rate 275 MB/sec).
- Snapshot backup using SAN technology is a very good option.
- Reduce the volume of data to backup regularly.
    - The filegroups for the historical partitions can be marked as READ ONLY.
    - Perform a filegroup backup once when a filegroup becomes read-only.
    - Perform regular backups *only* on the read / write filegroups.
- Note that RESTOREs of the read-only filegroups cannot be performed in parallel.

Top 10 Best Practices for SQL Server Maintenance for SAP

SQL Server provides an excellent database platform for SAP applications. The following recommendations provide an outline of best practices for maintaining SQL Server database for an SAP implementation.

Perform a full database backup daily

- Technically there are no problems to backing up SAP databases online. This means that end users or nightly batch jobs can continue to use SAP applications without problems. SQL Server Backup consumes few CPU resources. However, SQL Server Backup does require I/O bandwidth because SQL Server will try to read every used extent to the backup device. Everything that is required for SAP (business data, metadata and ABAP applications etc) is included in one database named "<SID>". Sometimes the time needed to take a full backup (generally a few hours) might become a problem, especially in SQL Server 2000 where no transaction log backups can be made while an Online Database Backup was performed. SQL Server 2005 does not have this issue.

- To create faster online backups using SAN Technology, SQL Server offers interfaces for SAN vendors to perform a Snapshot Backup or to create clones of a SQL Server database. However, backing up terabytes of data every night may overload the backup infrastructure. Another possibility would be to do differential backups of the SAP database on a daily basis and do a full database backup on the weekend only.

Perform transaction log backup Every 10 to 30 minutes

- In case of a disaster happening on the production server, it is vital that the most recent status can be restored using online or differential database backups plus a series of transaction log backups which ideally cover as close as possible to the time of the disaster. For this purpose it is vital to perform transaction log backups on a regular basis. If you only create a transaction log backup every two hours, the in the case of a disaster, up to two hours of committed business transactions would not be able to be restored. Therefore it is vital to do transaction log backups often enough to reduce the risk of losing a large number of committed business transactions in case of a disaster. In many productive customer scenarios, a time frame of 10-30 minutes proved to be an acceptable frequency. However, in combination with SQL Server log shipping, you can create SQL Server transaction log backups even every two or five minutes. The finest granularity achievable is to perform SQL Server transaction log backups scheduled by SQL Agent every minute. Besides reducing the risk of losing business transactions, transaction log backups also truncate log data in the SQL Server transaction log, and reducing the possibility of the transaction log becoming full.

Back up system partition in case of configuration changes

- Back up the system partition after any configuration changes. Use Windows Server 2003 Automated System Recovery (ASR), or other tools such as Symantec Ghost or SAN boot to restore the system partitions.

Back up system databases in case of configuration changes

- Back up the system databases (master, msdb, model) after any configuration changes. In SQL Server 2005, the resource database does not need to be backed up because it does not experience any changes and is installed with the SQL Server 2005 installation.

Run DBCC CHECKDB periodically (ideally before the full database backup)

- Ideally, a consistency check using DBCC CHECKDB sould be run before performing an online database backup. However, please note that DBCC CHECKDB is a very time and resource consuming activity that puts heavy workload on SAP production systems, especially on databases over one terabyte. On commodity hardware with a good I/O subsystem, I/O throughputs in the range of 100-150 GB/h can be achieved. Given such I/O throughputs, and the fact that there are many SAP databases up to 10 terabytes or more, it is clear that running a DBCC CHECKDB on a production system is not always practical. Therefore, many people choose not to run DBCC CHECKDB. Although all components of hardware and software have become more reliable over the last decade, physical corruptions can still happen. One reason for physical corruptions is a catastrophic power outage without having battery backup for hardware components. Another reason could be physical damage to connections or hardware components. In massive cases there is no other way than to go back to a backup and restore the SAP database and then apply all the transaction logs up to the most recent. However, to detect physical inconsistencies at an early state, or to know that the backup method is reliable, or to minimize impact of physical corruptions, the following three major measures should be considered:

    - Consider running DBCC CHECKDB on a regular basis. This could be on a sandbox system that runs a restored image of the production environment. On such a system, time and resource consumption of DBCC CHECKDB would not be a concern and would not affect production users.

    - Test actually restoring the SAP database from an online or differential and transaction log backup. The fact that a backup is on tape does not necessarily mean that it is consistent on tape or that it can be read from tape. Tape hardware or tape cassettes may fail over the years, and you do not want to be in a position where you have tapes that cannot be read anymore. Having a backup in a vault does not say anything about the ability to be able to restore in case of a disaster. The backup must also be proven to be readable.

    - For databases with terabytes of volume, maintain a second copy of the database at the most recent status, using either log shipping or database mirroring. Both of these high-availability methods will de-couple hardware components and hence may provide a physical consistent image of the production database at a secondary site.

Evaluate security patches monthly (and install them if they are necessary)

- For most of SAP customers, availability is the most important requirement. Especially if they need to serve a single SAP instance globally, they don't want to stop and restart the SAP servers to apply security patches. Plus, some testing in these environments is definitely necessary before installing the security patches. Therefore one of realistic scenarios for SAP customers is carefully evaluating patches and reducing the frequency of patch installations, hopefully almost to zero. Filtering unnecessary packets, disabling unnecessary services, and so forth are good security measures.

- If you have real time anti-virus monitoring, it is recommended that you exclude the SQL Server database

files (including data files, transaction log files, tempdb and other system database files) from real time monitoring. If you perform backups to disks, exclude database backup files as well as transaction log backup files.

Evaluate update modules of hardware drivers and firmwares and install if necessary

- There have been critical issues due to bugs in hardware drivers and firmwares within the commodity servers. It is sometimes difficult to find this kind of issue within Microsoft, and furthermore hardware companies sometimes don't provide enough support services to commodity server customers. So it is a customer's responsibility to manage updates on drivers and firmwares regularly. Before updating the drivers on production commodity servers, thorough tests must be conducted on test and sandbox systems. Like nearly no other software component, a little flaw in a driver of an Host Bus Adapter (HBA) or SCSI card can be responsible for physical inconsistency within a database.

Update statistics on the largest tables weekly or monthly

- SQL Server provides two options to keep the statistics current: **auto create statistics** and **auto update statistics**. These two options are ON by default. SAP recommends keeping them ON. There may be some cases where auto update statistics may not be able to provide satisfactory performance. A specific case came up in SAP BW. The issue was resolved by the functionality in SAP BW that is documented in SAP OSS note #849062. Please keep in mind that auto update statistics is run only on tables with more than 500 rows. In some very specific cases of data developing into one direction, it is recommended to explicitly run update statistics on specific columns of the table on a scheduled basis. However, you should not perform a general manual update statistics. If performance problems are analyzed and the root cause is found in an index, or some column statistics not being recent enough, then the solution often is simply to have a certain column or index statistics updated on a more frequent basis.

Rebuild or defrag the most important indexes

- The impact of reorganizing tables and indexes on performance is highly dependent on the type of query that is executed and the I/O bandwidth that is available on the system. Simply going along measures like (1) Average page density < 80 percent or (2) Logical scan fragmentation > 40 percent as thresholds to start reorganizing are a waste of time and resources. Reasons are:
    - Some SAP queuing tables will always show up as being highly fragmented
    - A query reading a single row or a small number of rows which represents the majority of SAP queries do not benefit from reorganizing a table.
    - If there is enough I/O bandwidth and memory for SQL Server on the database server, the impact of table fragmentation might be limited.

- Many people never reorganize tables to speed up query performance. However, there are also people who reorganize tables to compress them after they archived SAP data. Not all the tables are organized or

sorted according to the archiving criteria. Hence it can happen that despite deleting 25 percent of a table, the table only decreased its volume by 10 percent. To maximize space reduction after archiving, you can run DBCC INDEXDEFRAG on the affected tables. DBCC INDEXDEFRAG will compress the data content on the pages of a table. DBCC INDEXDEFRAG treats every move of a bunch of rows to one page as a single transaction. Hence DBCC INDEXDEFRAG will result in many small transactions as opposed to creating an index which treats the entire index creation task as one large transaction. DBCC INDEXDEFRAG does not consume much CPU resources, but it does create significant I/O traffic. Therefore do not run too many DBCC INDEXDEFRAG commands in parallel. Completely reorganizing tables by re-creating their clustered indexes should not be done on large tables because this will generate huge amount of transaction log.

Use a health check monitoring tool for performance, availability, and so forth

- Unplanned downtime depends on how quickly system failures are notified to administrators and how soon they can start the recovery process. For availability, SAP administrators should be aware that automatic failover mechanism of Microsoft Clustering Services (MSCS) or database mirroring (DBM) is able to provide continuous availability of the system. However, a failover itself will cause rollback of open transactions on the database side which again will cause rollbacks on business transactions on the SAP side. The impact of these batch processes breaking and data not being available might be serious (for example, with a payroll calculation). Therefore monitoring the system and notification after failovers can be vital to having interrupted SAP Business processes restarted as quickly as possible.

The document SAP with Microsoft SQL Server 2005: Best Practices for High Availability, Performance, and Scalability (5.5 MB) describes best practices for tuning and configuring SAP on SQL Server 2005 in more detail.

-