SQLCAT's Guide to: High Availability and Disaster Recovery

Microsoft SQLCAT Team

Guide & Reference

Microsoft

SQLCAT's Guide to:

High Availability and Disaster Recovery

Microsoft SQLCAT Team

Summary: This ebook is a collection of some of the more popular technical content that was available on the old SQLCAT.COM site. It covers SQL Server technology ranging from SQL Server 2005 to SQL Server 2012. However, this is not all the content that was available on SQLCAT.COM. To see additional content from that site you can follow the SQLCAT blog which will point to additional content. For more comprehensive content on SQL Server, see the MSDN library.

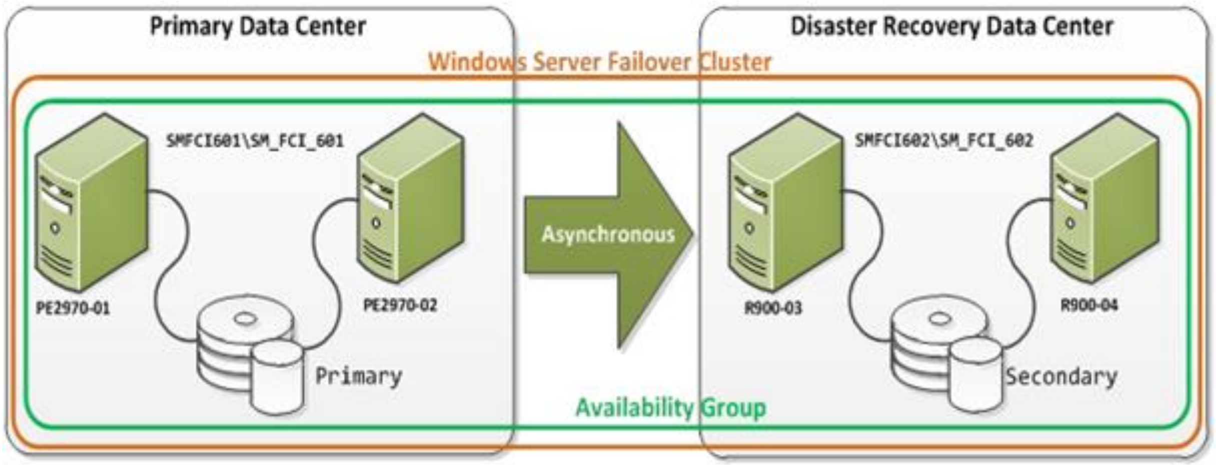
Category: Guide & Reference
Applies to: SQL Server High Availability and Disaster Recovery
Source: SQLCAT Blog
E-book publication date: September 2013

Contents

Section 1: SQL Server Failover Cluster

Impact of Adding a Node to a Windows Cluster on the Possible Owners property of existing SQL Server Failover Cluster Instances

It is common to have more than one SQL Server Failover Cluster Instance (FCI) within a single Windows Server Failover Cluster (WSFC). It is also fairly common that not all nodes in the WSFC are the possible owners of all the FCIs within that WSFC. One such scenario is when you have a FCI + AG solution where multiple FCIs reside within a single WSFC, and an availability group is created across the FCIs, as shown in the Figure 1 below:



(AlwaysOn Architecture Guide: Building a High Availability and Disaster Recovery Solution by Using Failover Cluster Instances and Availability Groups),

Figure 1: An FCI+AG solution across two data centers (FCI being used for HA and AG being used for DR)

In this deployment,

- Possible owners of SMFCI601\SM_FCI_601 = {PE2970-01, PE2970-02}.
- Possible owners of SMFCI602\SM_FCI_602 = {R900-03, R900-04}.

High level steps to build such a solution are:

1. **Step 1**: Build the primary site failover cluster instance SMFCI601\SM_FCI_601 on nodes PE2970-01 and PE2970-02.
2. **Step 2**: Add the DR site nodes R900-03 and R900-04, to the same Windows cluster, and then create the secondary failover cluster instance SMFCI602\SM_FCI_602 on R900-03 and R900-04.

3. **Step 3**: Create the AG across the two instances SMFCI601\SM_FCI_601 and SMFCI602\SM_FCI_602.

After building the primary site failover cluster instance (Step 1 above, shown in Figure 2 below), you will see the possible owners of the resources for the failover cluster instance resource group SMFCI601\SM_FCI_601 as shown in the Figure 3.

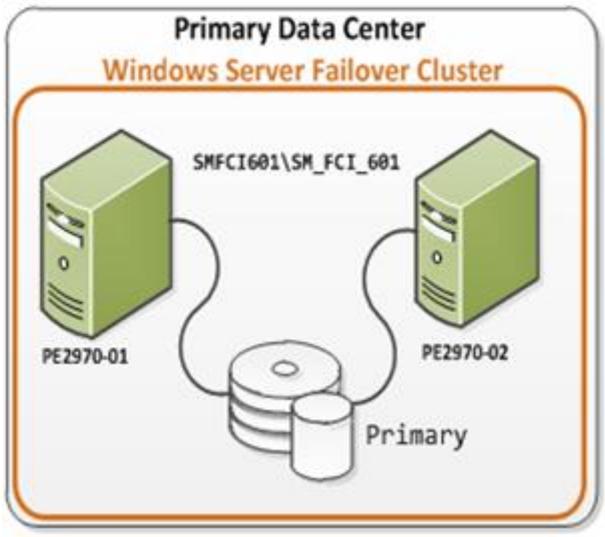
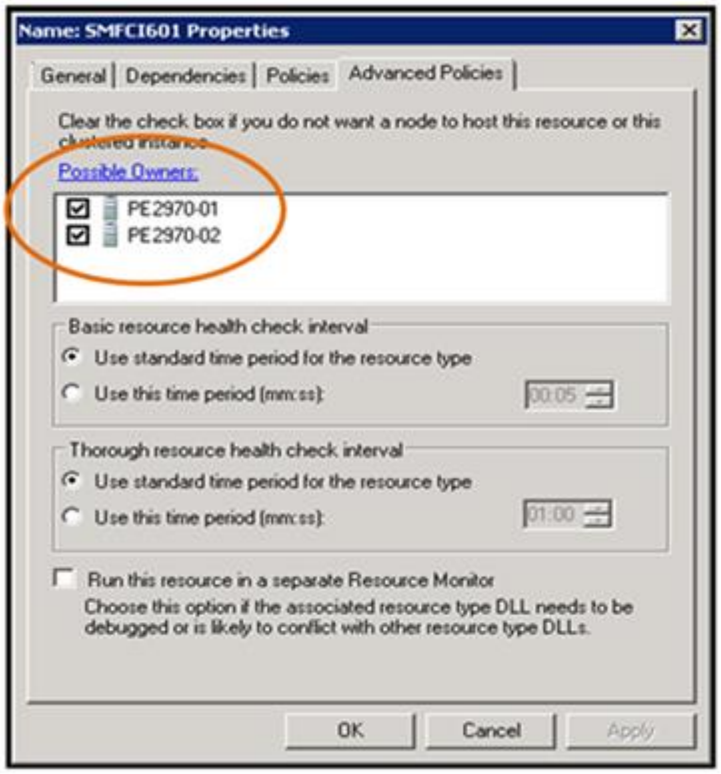


Figure 2: Topology after Completing Step 1 (2-node FCI created at the primary data center)


Name: SMFCI601 Properties

General | Dependencies | Policies | Advanced Policies

Clear the check box if you do not want a node to host this resource or this clustered instance.

Possible Owners:

☑ PE2970-01
☑ PE2970-02

Basic resource health check interval
(●) Use standard time period for the resource type
( ) Use this time period (mm:ss): 00:05

Thorough resource health check interval
(●) Use standard time period for the resource type
( ) Use this time period (mm:ss): 01:00

☐ Run this resource in a separate Resource Monitor
Choose this option if the associated resource type DLL needs to be debugged or is likely to conflict with other resource type DLLs.

OK | Cancel | Apply

Figure 3: Possible owners of the FCI network name resource after Completing Step 1 (2-node FCI created at the primary data center)

This is correct, and as expected.

During Step 2, the two nodes from the DR site (R900-03 and R900-04) are first added to the WSFC (Figure 4). Once these two nodes are added to the WSFC, you will see that the newly added nodes have been unexpectedly added as possible owners (Figure 5) for all the resources of the previously existing FCI (SMFCI601\SM_FCI_601).

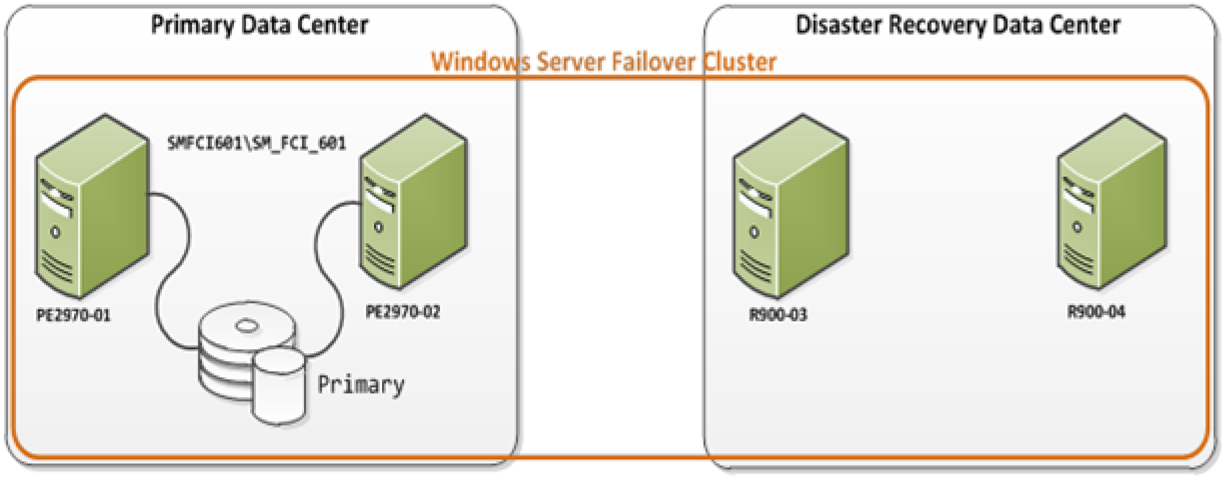
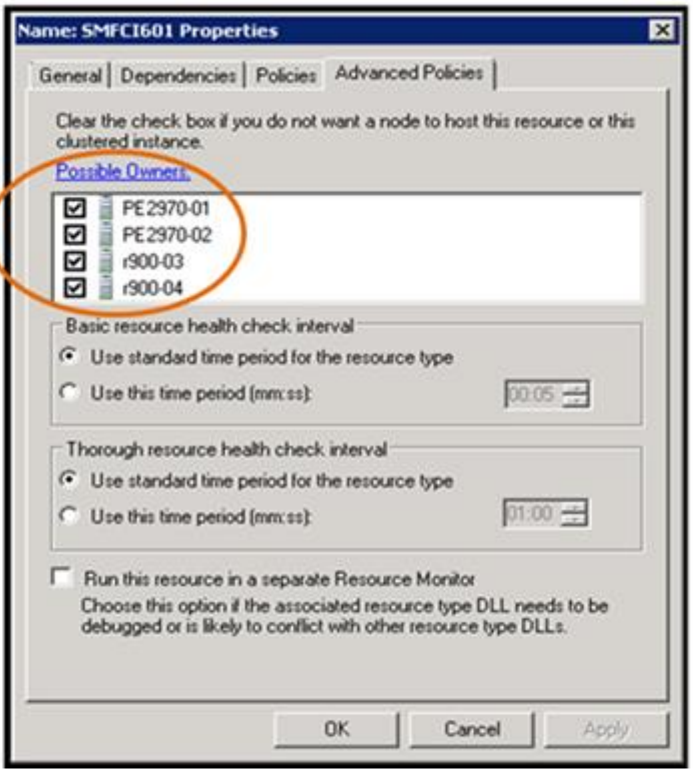


Figure 4: Two nodes from the DR site added to the WSFC


Name: SMFCI601 Properties

General | Dependencies | Policies | Advanced Policies

Clear the check box if you do not want a node to host this resource or this clustered instance.

Possible Owners:

- [x] PE2970-01
- [x] PE2970-02
- [x] r900-03
- [x] r900-04

Basic resource health check interval

- (●) Use standard time period for the resource type
- ( ) Use this time period (mm:ss): 00:05

Thorough resource health check interval

- (●) Use standard time period for the resource type
- ( ) Use this time period (mm:ss): 01:00

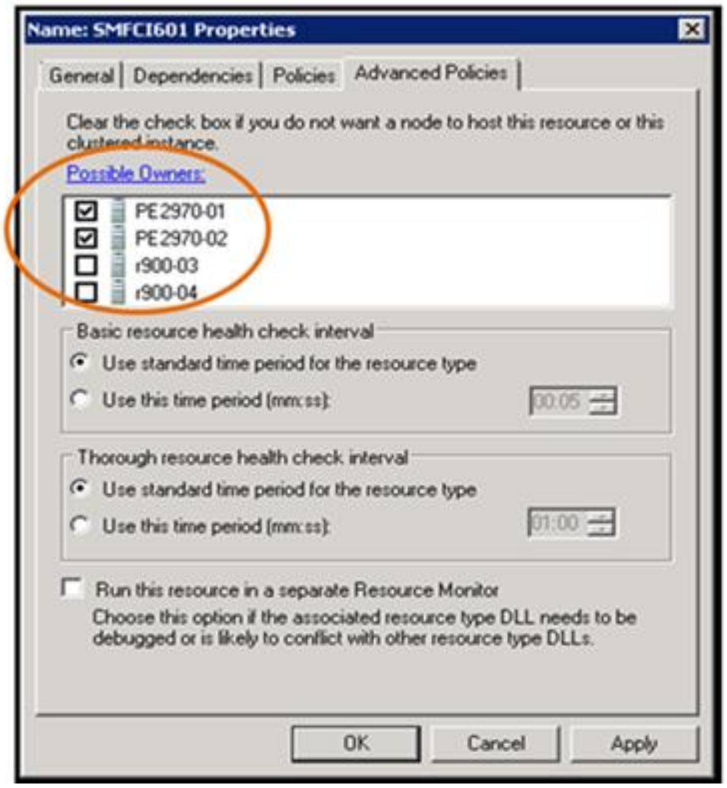
- [ ] Run this resource in a separate Resource Monitor

Choose this option if the associated resource type DLL needs to be debugged or is likely to conflict with other resource type DLLs.

Figure 5: Possible owners list of the FCI network name resource of the primary FCI, after adding two nodes from the DR site to the WSFC

This is neither intended nor desired. R900-03 and R900-04 can never be the possible owners of any resources of SMFCI601\SM_FCI_601. So, why does this happen? It is the way SQL Server FCI setup works with Windows clustering. When a node is added to the WSFC, the node is added as a possible owner for all existing FCIs. This works well in some scenarios, but doesn't work well in the scenario we are discussing in this article.

The possible owners list must be manually modified each time you add a node to the WSFC. Figure 6 shows corrected possible owners list for one resource (the FCI network name on the primary).


Name: SMFCI601 Properties

General | Dependencies | Policies | Advanced Policies

Clear the check box if you do not want a node to host this resource or this clustered instance.

Possible Owners:

- [✓] PE2970-01
- [✓] PE2970-02
- [ ] r900-03
- [ ] r900-04

Basic resource health check interval

[✓ Use standard time period for the resource type] Use this time period (mm:ss): 00:05

Thorough resource health check interval

[✓ Use standard time period for the resource type] Use this time period (mm:ss): 01:00

[ ] Run this resource in a separate Resource Monitor
Choose this option if the associated resource type DLL needs to be debugged or is likely to conflict with other resource type DLLs.

Figure 6: Possible owners list of the FCI network name resource of the primary FCI, after manually correcting the list

Repeat this for all resources (other than disks, the disk resources don't demonstrate this behavior), such as:

- FCI network name
- FCI IP address
- SQL instance
- SQL Agent
- Any other resources part of the resource group must be checked for correct possible owners.

And repeat this for each FCI in your topology. Note that the possible owners list for the resources in each FCI will differ from other FCIs.

The possible owners list must be appropriately modified before proceeding to create the availability group across the FCIs (Step 3 listed above), otherwise AG creation will fail, because there will be overlap between the possible owners of the two instances hosting the AG replicas.

This should not be confused with the possible owners list for the availability group resource. You should not alter the possible owners list for the availability group (DO NOT use Windows Failover Cluster Manager to perform Availability Group Failover).

Six Failover Clustering Benefits Realized from Migrating to SQL Server 2008

SQL Server 2008 failover clustering introduces several supportability, reliability, and availability improvements. The following list details the more significant and immediate benefits of making the move to SQL Server 2008 Failover Clustering.

1 - Reliable Setup

The installation process for SQL Server 2008 Failover Clusters has changed significantly. Essentially, you have two options for installation, **integrated** install or **advanced/enterprise** install. Integrated install involves the installation of a single-node SQL Server 2008 failover cluster instance. If you want the instance to be able to failover to other nodes, you follow a separate “add node” install for each node.

The advanced/enterprise install differs from the integrated install in that you prepare each node with SQL binaries and services, and then select the active node that owns the SQL Server shared disk, and then bring the SQL Server instance online in a separate step. The advanced/enterprise install is intended for third party enterprise deployment solutions (yet to hit the market), or adding the ability to prepare each node prior to configuring the Windows Cluster. You may decide to use the advance option if you prefer, but overall the integrated install option will provide less steps and will allow you to make the SQL Server instance available sooner.

From a “number of steps” perspective, the integrated install option requires less effort. For example, a two node cluster integrated install would require one “Install” step on the first node, and then an “Add node” step on the second node. An advanced/enterprise install would require a “prepare” operation on each node (two steps), followed with a third step, “completing” the SQL Server instance and bringing it online.

At first blush, this seems like more work for the DBA, so where is the benefit to this new process? Unlike with SQL Server 2005 failover clusters, SQL Server 2008 Failover Cluster installs do *not* involve remote-node operations. This translates to more discrete install steps on your part, but helps reduce several installation and patching problems that occur due to remote node permission issues, remote offline services, terminal services connections, and other communication issues that can leave you with a partial or failed installation. By moving to a SQL Server 2008 Failover Cluster, the reliability of your install will increase significantly by eliminating the several remote-node variables that once hampered a solid install of a SQL Server failover cluster.

2 - Improved availability with rolling upgrades

Prior to SQL Server 2008, installing a service pack or cumulative update could require a several-minute outage for the SQL Server instance. This is due to the fact that in order to update a SQL Server instance to the latest Service Pack or Cumulative update, SQL Server services were stopped until the upgrade was completed. With SQL Server 2008 failover clustering, your outage period can be significantly reduced if you follow the proper “rolling update” process. Specifically, you can avoid prolonged outages of a SQL Server instance by applying Service Packs or Cumulative Updates against the *passive* nodes of a failover cluster. After applying the patches to the passive nodes, you can then failover the SQL Server instance to a newly upgraded node. Upon failover, the SQL Server instance is then upgraded. You can then proceed with updating the formerly active (now passive) nodes.

In my own testing of a two-node cluster hosting a single SQL Server 2008 failover cluster instance, I started off my patching process by installing a Cumulative Update on the passive node of the cluster. While this Cumulative Update was installed, the SQL Server instance remained up. After applying the cumulative update, I failed over the instance of SQL Server 2008 to the newly upgraded node, and then applied the cumulative update to the new passive node. The total down time for the upgrade was 15 seconds, which was the amount of time it took to fail over the SQL Server instance to the newly upgraded node.

3 - Availability with adding or removing nodes

As with SQL Server 2005, adding a new node for a SQL Server failover cluster instance or removing a node does not require an outage of the SQL Server instance. Like all cluster setup actions, AddNode needs to be run on the node to be added, as opposed to on the active node for 2005. This results in increased reliability since 2008 AddNode does not rely on remote task scheduling and execution. The only user inputs to the 2008 AddNode are: instance selection, service account passwords on the UI (service account names and passwords on the command line), Error and Usage Reporting options. All feature selection is retrieved from the existing instance where the current node is being added.

Also, in my own testing when adding a new node to a SQL Server failover cluster, I received the following notification during install:

"The current node TX147913-3 is at patch level [10.0.1600.22], which is lower than that of active node TX147913-2: patch level [10.0.1763.0]. After completing setup, you must download and apply the latest SQL Server 2008 service pack and/or patch and bring all nodes to the same version and patch level."

This helpful warning let me know that I needed to update the newly added SQL Server failover cluster to match the existing and already-upgraded SQL Server failover cluster node. Patching the newly added passive node did not require a restart of the SQL Server 2008 failover cluster.

4 - Service SIDs instead of Domain Groups on Windows Server 2008

A pain point for many DBAs was the introduced requirement in SQL Server 2005 Failover Clustering for using domain groups for SQL Server services. These domain groups were used to manage the permissions of the SQL Server service accounts; however they required that each domain group already contained the service accounts as members prior to install. Changing the domain group for a clustered service, although possible, was not a trivial procedure (see KB 915846, "Best practices that you can use to set up domain groups and solutions to problems that may occur when you set up a domain group when you install a SQL Server 2005 failover cluster").

If you are creating a new SQL Server 2008 failover cluster on Windows Server 2008, you can now bypass the use of domain groups by designating Service SIDs during the install. Service SID functionality was introduced in Windows Vista and Windows Server 2008, and allows the provisioning of ACLs to server resources and permissions directly to a Windows service. On the "Cluster Security Policy" dialog during install of a SQL Server failover cluster, you still have the option to use domain groups, however selecting "Use service SIDS" is the recommended choice for SQL Server 2008 on Windows Server 2008 and allows you to bypass provisioning of domain groups and associated service account membership additions prior to installation.

5 - Windows server 2008 Integration improvements

In addition to Service SIDs, running SQL Server 2008 on Windows Server 2008 provides other significant benefits. For example, Windows Server 2008 clustering removes the requirement for having all hardware in a cluster solution be listed in the Hardware Compatibility List (HCL). Finding and validating your exact cluster solution in the HCL was often a difficult task. For Windows Server 2008, you *no longer* need to validate your exact solution in the HCL. Instead, your Windows Server 2008 logo cluster solution must pass validation using the Windows Server 2008 Cluster Validation Tool. Prior to configuration of your cluster, you can use this tool to scan the server nodes and storage you plan on using for your cluster solution. The tool checks for any issues that may impact support of a Failover Cluster. Any blocking issues across the hardware, network components and configurations, storage resources, and Operating System configurations will be identified in a final report and will allow you to address issues prior to deployment.

Windows Server 2008 Failover Clustering also added new quorum options, moving from a single-point-of-failure to a consensus-based quorum model. Windows Server 2008 Failover Clustering also offers iSCSI disk support, up to 16-node clusters, and ipv6 internet layer protocol support.

6 - ConfigurationFile.ini automatic generation

SQL Server 2008 Failover cluster allows for the use of a configuration file used in conjunction with a command line setup. For example – the following command line execution initiates an integrated install of a single-node failover cluster, referencing a configuration file with the required command line options:

```
Setup.exe /q /ACTION=InstallFailoverCluster /Configurationfile="C:\temp\ConfigurationFile.ini"
```

What’s more, performing a non-command line install of SQL Server 2008 automatically generates a ConfigurationFile.ini which is saved to the following directory:

```
<drive letter>:\Program Files\Microsoft SQL Server\100\Setup
Bootstrap\Log\<YYYYMMDD_HHMMSS\ConfigurationFile.ini.
```

Please note that as of this writing, ConfigurationFile.ini does not automatically include the FAILOVERCLUSTERIPADDRESSES setup option – however this is easy to add manually. For example:

```
FAILOVERCLUSTERIPADDRESSES="IPv4;172.29.10.160;Cluster Network 1;255.255.248.0"
```

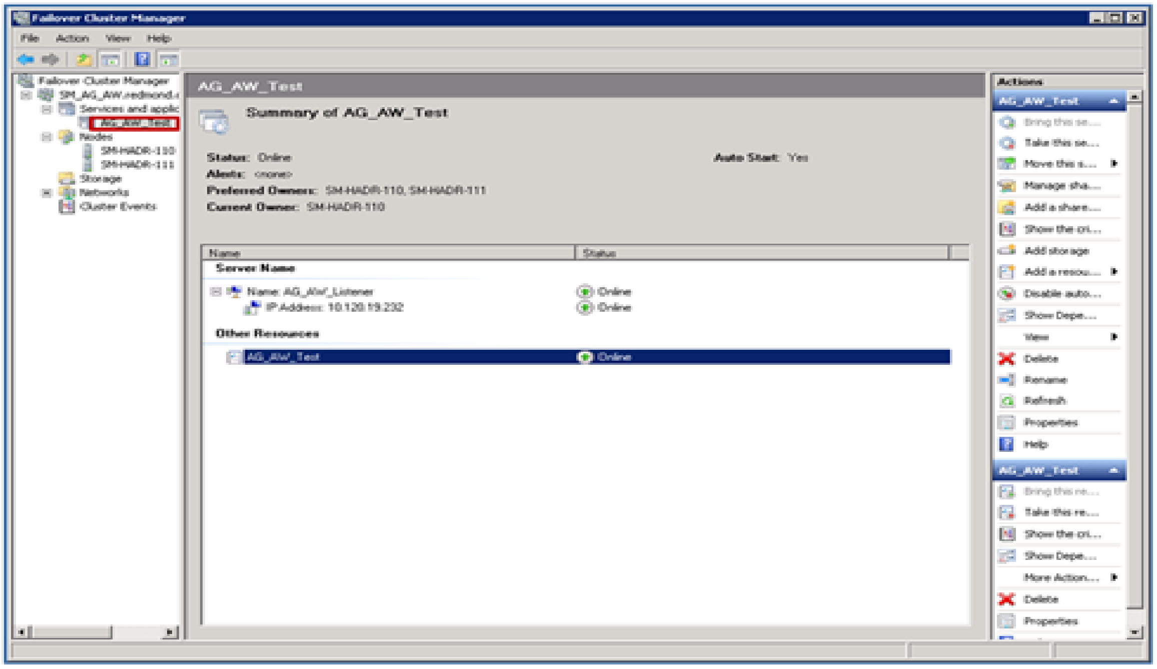
Using command-line setup in conjunction with a configuration file can help streamline your SQL Server 2008 failover cluster installation process, particularly for large enterprise environments.

Section 2: SQL Server Always On

DO NOT use Windows Failover Cluster Manager to perform Availability Group Failover

Author: Sanjay Mishra
Contributors: David P Smith (ServiceU)
Reviewers: Chuck Heinzelman, Mike Weiner, Prem Mehra, Kevin Cox, Jimmy May, Tim Wieman, Cephas Lin, Steve Lindell, Goden Yao

Windows Server Failover Cluster (WSFC) is the foundation for SQL Server 2012 AlwaysOn Availability Group functionality. The Availability Group (AG) is registered as a resource group within the Windows Server Failover Cluster. Figure 1 shows an availability group in the Windows Failover Cluster Manager (FCM) interface.



Status: Online
Alerts: <none>
Preferred Owners: SM-HADR-110, SM-HADR-111
Current Owner: SM-HADR-110
Auto Start: Yes

Name	Status
Server Name	
Name: AG_AW_Listener	Online
IP Address: 10.120.19.232	Online
Other Resources	
AG_AW_Test	Online

Figure 1: Availability Group service in the Failover Cluster Manager interface

Even though Availability Group (AG) is a resource group within the Windows Server Failover Cluster, DO NOT use the Failover Cluster Manager (FCM) to perform certain operations on the AG:

- DO NOT change the *preferred owners* and *possible owners* settings for the AG. When an AG is created, the *preferred owners* and *possible owners* settings for the AG are established based on the primary and secondary servers information provided to SQL Server. Whenever a failover happens, the *preferred owners* and *possible owners* settings for the AG are reset based on the new primary. This is automatically done for you by the AG, so do not try to manually configure these settings.

- DO NOT change the *preferred owners* and *possible owners* settings for the AG listener. Similar to the AG discussion above the AG Listener settings are handled automatically.
- DO NOT move the AG between nodes using the Windows Failover Cluster Manager. The FCM doesn't provide or have any awareness as to the synchronization status of the secondary replicas. Therefore, if the replica is not synchronized and the AG resource is failed over, the failover will then fail which can lead to extended downtime. The recommended ways to perform AG failover include SQL Server Management Studio and T-SQL statements.
- DO NOT add or remove resources in the AG resource group.

Note that the FCM does not prevent you from performing any of these operations. However, we recommend against executing these actions through FCM, as doing so may result in unintended outcomes, including unexpected downtime.

Database Mirroring Log Compression in SQL Server 2008 Improves Throughput

Author: Sanjay Mishra

Reviewers: Peter Byrne, Don Vilen, Kaloian Manassiev, Burzin Patel, Eric Jacobsen

Overview

Database mirroring works by transferring and applying a stream of database log records from the principal database to the mirror database. The log records are transferred over a network. For an application that generates lots of transaction log (as measured by the perfmon counter Log Bytes Flushed / sec), the bandwidth of the network can be a limiting factor. The efficiency of log transfer plays a significant role in achieving the best application throughput and performance in a database mirroring environment.

SQL Server 2008 introduces a new feature called "Database Mirroring Log Compression". With SQL Server 2008, the outgoing log stream from the principal to the mirror is compressed, thereby minimizing the network bandwidth used by database mirroring. In a network constrained for bandwidth, compressing the log stream helps improve the application performance and throughput.

In this study, we took a customer workload (an application at a stock exchange that captures stock quotes and orders), with a high log generation rate (12 MB/sec) and ran performance tests with a pre-release build of SQL Server 2008. The results indicate a significant performance benefit of log compression, especially on lower bandwidth networks.

What is Log Compression

Transferring transaction log records from the principal server to the mirror server over a network is central to database mirroring implementation. The more transaction log an application generates, the more log records need to be transferred from the principal to the mirror. Higher log generation rate puts a higher demand on the network bandwidth.

Synchronous database mirroring requires that the log records for each transaction be received and hardened by the mirror database and a confirmation message sent to the principal, before the transaction is committed. Therefore, the network plays a very important role in influencing the performance and throughput. As you can guess, the throughput of an application can reduce under a lower bandwidth network. Refer to the white paper Database Mirroring Best Practices and Performance Considerations for a discussion and performance results of applications under database mirroring in networks with varying bandwidth with SQL Server 2005.

Compressing the log stream means that a significantly less amount of network packets are sent from the principal to the mirror. Therefore more log records can be sent in a given time. This translates into improved throughput for the application.

Test Workload and Test Environment

The workload used for the tests described in this paper consists of a 20 GB database, and 20 concurrent active user connections.

The test hardware consisted of 2 Unisys x64 servers with 16 processors and 64 GB RAM each. The storage was EMC Clariion SAN. The network between two servers 1 Gbps.

The SQL Server 2008 prerelease build 10.00.1049.00 was used.

Synchronous database mirroring was used for these tests.

Improved Throughput

With SQL Server 2008, the log stream is compressed by default. To measure the impact of log compression, we used the trace flag 1462, which disables log compression. Disabling log compression is equivalent of SQL Server 2005 behavior.

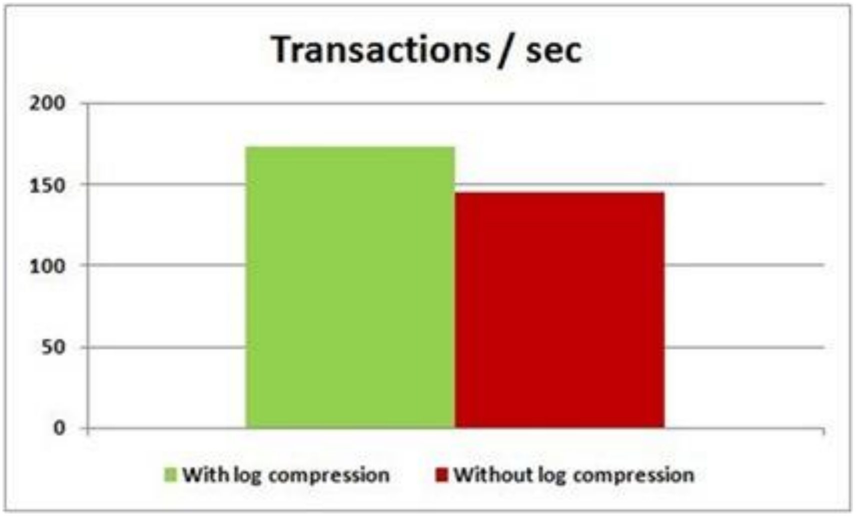


Figure 1: With log compression, database mirroring throughput is improved

As shown in Figure 1, the log compression results in higher throughput compared to disabling log compression.

Log Compression Helps Significantly on Lower Bandwidth Networks

The impact of log compression on throughput is much more pronounced on lower bandwidth networks. We used a network emulator software to simulate different network bandwidths between the principal and the mirror servers, and measured the application throughput with and without log compression. Figure 2 indicates a very significant improvement in throughput by having log compression.

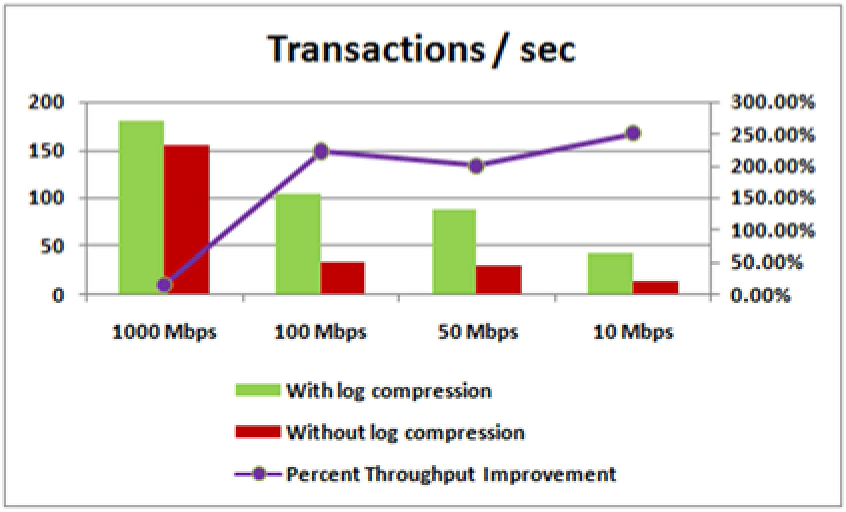


Figure 2: The improvement is more pronounced on lower bandwidth networks

As can be observed in Figure 2 (the line chart), the percent of throughput improvement obtained by log compression is usually higher for lower bandwidths. The throughput improvement experienced by an application is dependent upon the data processed by the application.

Log Compression Ratio

Log compression ratio indicates the factor by which the log stream has been compressed. You can divide the perfmon counter Log Bytes Sent/sec by the counter Log Compressed Bytes Sent/sec to get the compression ratio. Figure 3 illustrates the log compression ratio obtained in our test workload. The compression ratio depends upon the application and the data it processes. The compression ratio is not an externally configurable parameter; it is an inherent property of the data.

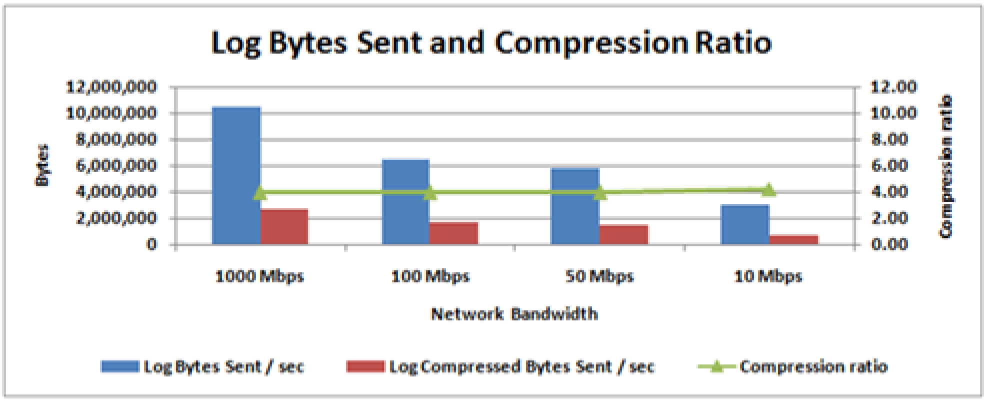


Figure 3: Log compression ratio

Cost of Log Compression

The outstanding benefits of log compression come with some processing cost. The log records are compressed on the principal before being sent to the mirror, where they are uncompressed before being applied to the mirror database. The extra tasks of compressing and uncompressing the log adds some processing overhead on the principal and mirror respectively, resulting in higher CPU usage. Figure 4 illustrates the CPU usage of the principal and mirror servers at various network bandwidths, with and without log compression.

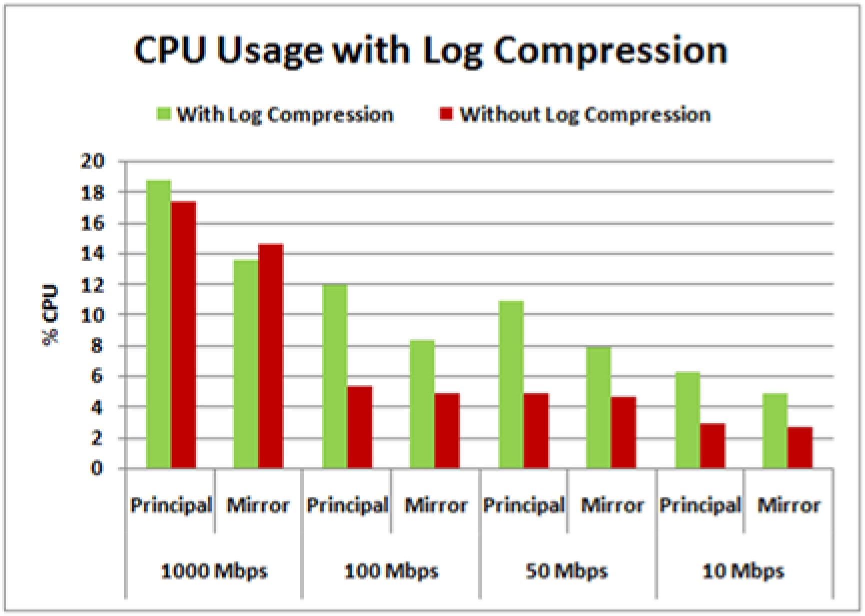


Figure 4: CPU Usage with log compression

As illustrated in Figure 4, the CPU usage on the principal and mirror servers goes up significantly with log compression, sometimes by as much as twice the CPU usage without log compression. One reason for increased CPU usage is the processing overhead introduced by compressing and uncompressing the log records. The other reason is that the servers are now processing a much larger number of transactions.

It is important to note the tradeoff between the increased CPU usage and the improved throughput of the application. If your application generates a significant amount of transaction log, you may notice reduced throughput and reduced CPU usage when you configure synchronous database mirroring in SQL Server 2005 (Refer to the white paper Database Mirroring Best Practices and Performance Considerations for some data points). The log compression feature introduced in SQL Server 2008 will get you back some of that throughput, and some of that CPU load.

Summary

SQL Server 2008 introduces a new feature that compresses the log stream sent from the principal to the mirror in a database mirroring configuration. Compressing the log stream results in improved application throughput, at the cost of increased CPU load. The improvement in application throughput as well the increase in CPU load are both dependent upon the workload. Some workloads may find that the throughput is constrained by CPU capacity. It is recommended that you test with appropriate workload to estimate the expected improvement in application throughput and increase in CPU load in your environment.

Comparing I/O characteristics in AlwaysOn Availability Groups and Database Mirroring

Microsoft SQL Server 2012 introduces AlwaysOn Availability Groups, which is a high availability and disaster recovery solution. The basic concept of AlwaysOn Availability Groups is similar to Database Mirroring, which was introduced in SQL Server 2005 SP1. AlwaysOn Availability Groups offers improvements over Database Mirroring; for example, with AlwaysOn Availability Groups, you can have multiple databases in one availability group, and you can have up to four secondary replicas for every primary replica. For more information about AlwaysOn Availability Groups, see Overview of AlwaysOn Availability Groups (http://technet.microsoft.com/en-us/library/ff877884(v=SQL.110).aspx).

AlwaysOn Availability Groups not only extends the functionality of Database Mirroring, it also provides performance enhancements. This paper provides some technical details on two of these enhancements:

- The I/O efficiency of flushing data pages on the secondary replica
- Throughput improvements by optimized I/O for transaction log file using log pool

The paper also discusses the results of tests that we performed to demonstrate these features and their effect on performance.

I/O Efficiency of Flushing Pages on the Secondary

One significant change in SQL Server 2012 is the improved I/O efficiency on the secondary replica (also known as the mirror database in Database Mirroring). Database Mirroring does a continual flush of the dirty pages, because to go from a 'restoring' state to an 'online' state (that is, from the mirror database to the principal database), all pages are required to be on disk. To shorten recovery time at failover, Database Mirroring writes dirty data pages to disk continuously. Because databases in AlwaysOn Availability Groups are online in both primary and secondary replicas (regardless of whether they are readable or not) , the flush is not required on a role change. Therefore AlwaysOn Availability Groups does not need to force page flushes except at checkpoints or buffer pool memory pressure. This change significantly reduces I/O for the secondary replica, when there are multiple updates on same page.

To compare AlwaysOn Availability Groups and Database Mirroring, I used one OLTP-type database and the same workload for AlwaysOn Availability Groups and Database

Mirroring. Each database had 12 data files (6 files on each file group) and one transaction log file. The total size was approximately 80 GB allocated and 35 GB used at the initial state. Both AlwaysOn Availability Groups and Database Mirroring were configured as synchronous replicas or mirrors. The workloads were mixed, with INSERTs, UPDATEs and READs. The concurrent connected user count was 1,000 users. On each node, data files were placed on drive H:(direct attached SSD drive), and transaction log files were on drive E:(SAN storage).

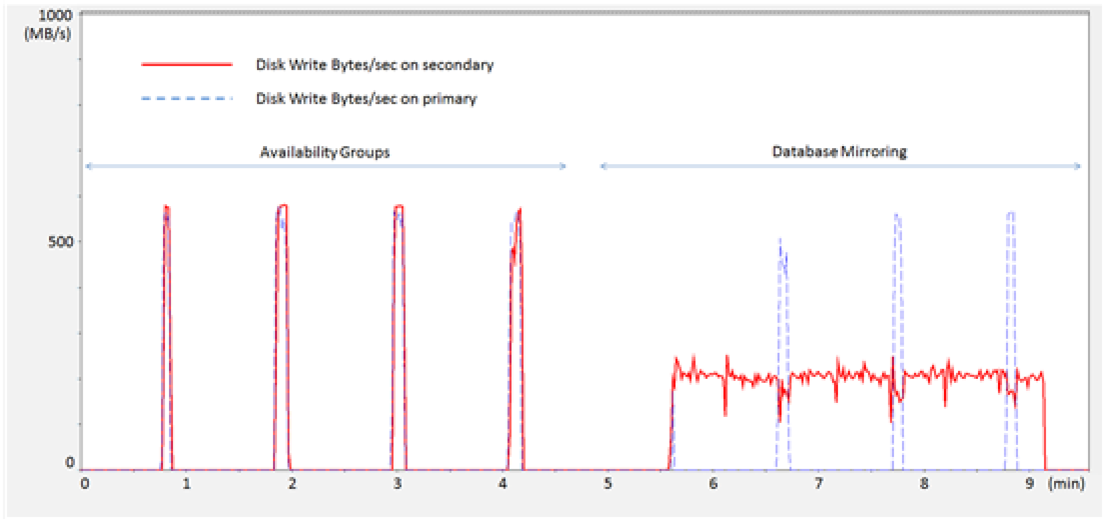


Figure 1: The Data Disk Write Byte /sec counter

In Figure 1, the red line shows the Disk Write Bytes/sec counter values on the secondary node data disk. It shows that there were approximately 500 MB per second of write I/O every minute for AlwaysOn Availability Groups; this write I/O occurred at the checkpoint. There were continuously 200 MB per second of write I/O for Database Mirroring. Note: The blue dotted line tracks the Disk Write Bytes/sec counter on the primary replica (in AlwaysOn Availability Groups) or principal database (in Database Mirroring), showing the checkpoint activity, which is a very similar pattern in both technologies. Average Disk Write Bytes/sec is around 63 MB per second for AlwaysOn Availability Groups and 205 MB per second for Database Mirroring during workload. These test results show that AlwaysOn Availability Groups reduces write I/O on the mirror significantly, compared to Database Mirroring.

Throughput improvements by optimized I/O for transaction log file using the log pool

SQL Server 2012 Availability Groups introduces another improvement, the dynamic cache capability of the log pool. This capability increases throughput capacity on the AlwaysOn Availability Groups databases. The log pool is a new common cache mechanism for log records consumers. When a transaction is run on the primary replica, the transaction log records are written to the log cache, and at the same time they are sent to the log pool to be passed along to the secondary replica. Figure 2 shows an example of this with a single secondary replica, although the logic is the same for multiple secondary replicas. If an unsent log is not in cache, AlwaysOn Availability Groups or Database Mirroring log capture threads have to read the log from the file. The log pool serves as a dynamic cache that can grow until unsent log entries fit into the cache, if there is no memory pressure. Then AlwaysOn Availability Groups adds less I/O to the log file for read than Database Mirroring does.

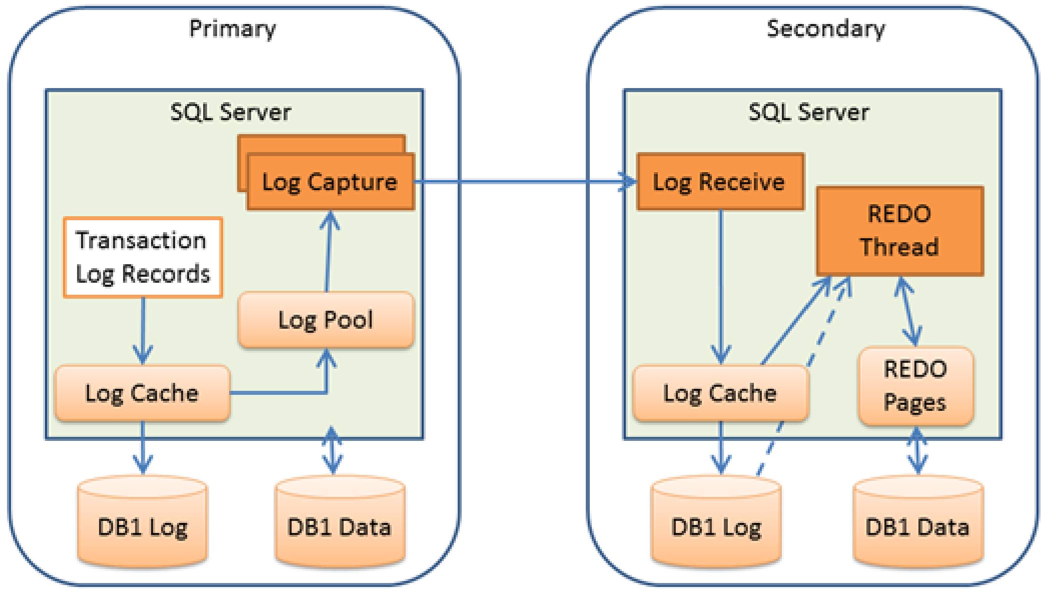


Figure 2: Data movement architecture of AlwaysOn Availability Groups

In Figure 3, the red line shows Batch Requests/sec, the blue line shows Log write waits, and the green line shows Disk Read Bytes/sec for the transaction log. These results are all taken from the primary replica or the principal database.

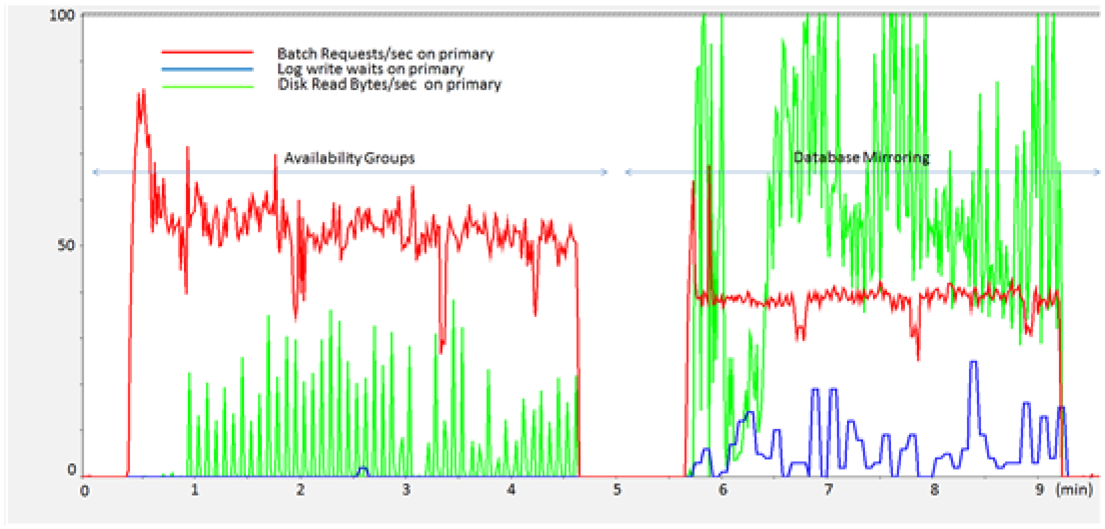


Figure 3: Batch Requests/sec

For AlwaysOn Availability Groups, the average value for Batch Requests/sec is about 5000, and for Database Mirroring, the average is 4000. This improvement in throughput is caused by a difference in the Log writes waits counter and heavier read I/O (as measured by the Disk Read Byes/sec counter*)* on the transaction log file for Database Mirroring. This enhancement comes from the log pool with a dynamically sized log cache. Database Mirroring, on the other hand, uses a fixed-size log cache. These log pool changes enable AlwaysOn Availability Groups to handle spikes in log volume much better, which results in an increase in capacity over Database Mirroring.

Section 3: SQL Server Mirroring

Mirroring a Large Number of Databases in a Single SQL Server Instance

Overview

A frequently asked question regarding database mirroring is: How many databases can be mirrored in a single instance of Microsoft® SQL Server®? This question is often raised by customers who are consolidating many databases into fewer instances and the high availability or disaster recovery service-level agreement (SLA) requires deployment of database mirroring.

Sometimes the question is raised when readers misconstrue the restriction documented in the SQL Server Books Online stating that "On a 32-bit system, database mirroring can support a maximum of about 10 databases per server instance because of the numbers of worker threads that are consumed by each database mirroring session."

It is important to mention two significant points:

- The restriction of 10 databases doesn't apply to 64-bit systems. In SQL Server Books Online, there are no such documented restrictions on the number of databases in a 64-bit system.
- Various customers have successfully deployed database mirroring with more than 10 databases in a 64-bit environment.

This article explains the number of worker threads required for database mirroring for each database and illustrates the observed performance of an application with many databases. System administrators and database administrators may find this information useful as they examine, test, and deploy systems in the production environment.

Threads Used by Database Mirroring in SQL Server 2008

The number of threads used on a server for database mirroring depends upon:

- The role of the server – principal or mirror
- The number of databases mirrored in an instance
- The number of logical processors on the mirror server

The following table summarizes the number of database mirroring threads used.

Role of server	**Thread function**	**Number of threads for the specific function**
Principal	Database mirroring communications	1 per instance
	Event processing	1 per mirrored database
	Log send	1 per mirrored database
Mirror	Database mirroring communications	1 per instance

	Event processing	1 per mirrored database
	Log hardening	1 per mirrored database
	Redo manager	1 per mirrored database
	Redo threads	FLOOR ((number of logical processors +3) / 4)

In summary, the number of database mirroring threads:

- On a principal is equal to (2 * number of mirrored databases) + 1.
- On a mirror equals ((3 + FLOOR ((number of logical processors +3) / 4)) * number of mirrored databases) + 1.

For example: If your principal and mirror servers have 8 logical processors each, and you are mirroring 20 databases, the total number of database mirroring threads used on the principal will be 41, and the total number of database mirroring threads used on the mirror will be 101.

To view the threads used for database mirroring, query the DMV sys.dm_exec_requests.

```
SELECT SESSION_ID, STATUS, COMMAND, WAIT_TYPE

FROM SYS.DM_EXEC_REQUESTS WHERE COMMAND = 'DB MIRROR'
```

Here is sample output from this query on the mirror server with 8 logical processors, and with one mirrored database.

SESSION_ID	**STATUS**	**COMMAND**	**WAIT_TYPE**	
15	background	DB MIRROR	DBMIRROR_EVENTS_QUEUE	Database mirroring communications
16	background	DB MIRROR	DBMIRROR_EVENTS_QUEUE	Event processing
16	background	DB MIRROR	DBMIRROR_DBM_MUTEX	Redo manager
16	background	DB MIRROR	DBMIRROR_DBM_MUTEX	Redo thread
16	background	DB MIRROR	DBMIRROR_DBM_MUTEX	Redo thread
17	background	DB MIRROR	DBMIRROR_SEND	Log hardening

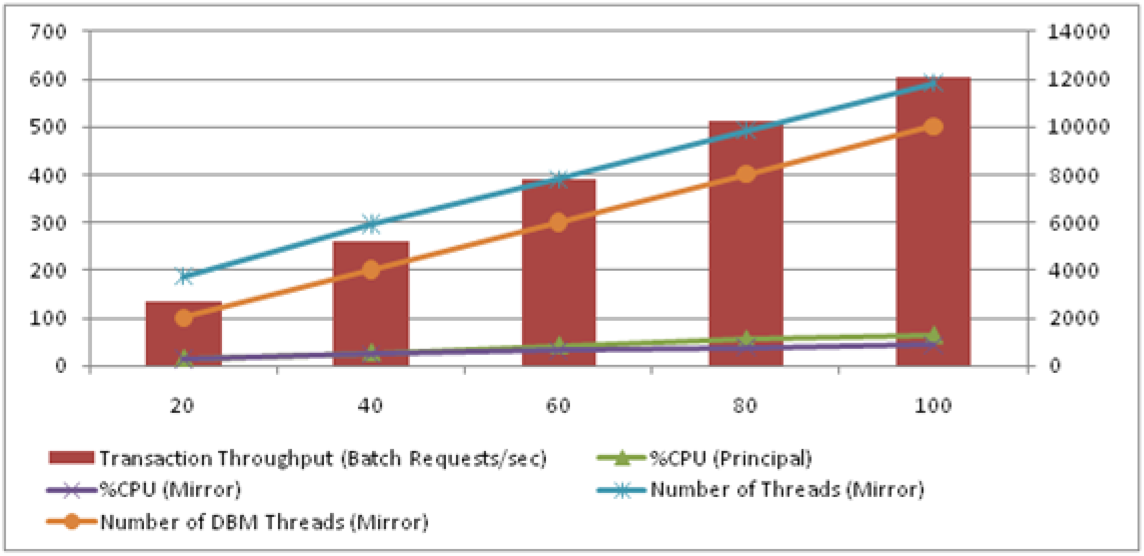
Test Results

To illustrate database mirroring in a consolidated environment, we ran tests with a workload that continuously inserted data into a number of databases. The databases were deployed with synchronous mirroring between two identical servers (for more information, see “Test Hardware and Software” in the Appendix). The application opened 20 connections to each database, and each connection continuously inserted 6,000 rows (one by one) into the database. The insertion rate of the workload scales linearly as we added more databases.

The focus of our tests was to demonstrate the ability to mirror multiple databases in a consolidated database environment. As an aside, we also observed linear scalability of the workload throughput. For more information about database mirroring performance topics, see Database Mirroring Best Practices and Performance Considerations.

We measured the thread counts, percent of CPU used, and application throughput as we increased the number of mirrored databases. For estimating CPU usage and throughput in your environment, we recommend that you test with your workload.

Number of databases mirrored	**Transaction throughput (batch requests/sec)**	**%CPU (principal)**	**%CPU (mirror)**	**Number of all threads (principal)**	**Number of DBM threads (principal)**	**Number of all threads (mirror)**	**Number of DBM threads (mirror)**
20	2,680	15	13.9	212	41	187	101
40	5,244	27.3	24.4	342	81	296	201
60	7,814	42.1	32.5	603	121	391	301
80	10,251	55.1	38.6	612	161	492	401
100	12,102	63.5	42.9	612	201	592	501



The line "Number of Threads (Mirror)" shows the total number of threads used on the mirror server. This includes the number of threads used by database mirroring represented by the line "Number of DBM Threads (Mirror)."

Additional Considerations

There are two additional considerations – one on application throughput, and another on ping traffic.

Impact of Threads on Throughput

As an aside, note that the overall system performance may be impacted by the number of threads available for processing the workload:

- The database mirroring sessions will consume some number of threads, as discussed above.
- The remaining threads are available to service user requests.

- If **max_worker_threads** is not set high enough, user requests might wait on worker threads even though there are adequate CPU and other resources to service those requests.
- Monitor the wait statistics wait for the worker counter to find out whether you are running low on worker threads, and set **max_worker_threads** appropriately.

Ping Traffic

For the database mirroring session for each database, the servers send and receive a ping from each other. When several databases are mirrored, the servers will send each other many ping messages within a short interval. To keep the database mirroring connection open, a server must receive and send a ping on that connection within the timeout period.

If the database mirroring partner timeout interval has been set too low, false failovers may occur (if a witness is enabled), or the database mirroring session may switch between disconnected and synchronizing states (if there is no witness).

We recommend that you set the timeout interval to a value of 10 seconds (default) or higher, and if you see unexpected timeouts, increase it to a higher value.

Recommendations

- When you set max worker threads (**sp_configure** option), make sure to accommodate threads needed by database mirroring in addition to the threads needed by your workload.
- When you set max server memory, make sure to account for the memory consumed by the threads, which is allocated from outside of the buffer pool memory. For more information, see 64-bit Solutions for SQL Server.
- Set these options to the same value on both partners (principal as well as mirror), so that the experience is consistent after a role change.
- Ensure that application performance is acceptable with an increased number of databases and the associated workload on the consolidated server.

Conclusion

With proper planning, it is possible to mirror hundreds of databases in an instance. Use the information provided in this article for planning, and perform thorough testing with your workload to ensure successful deployment.

Appendix A: Test Hardware and Software

All the tests were performed in the following hardware and software environment.

Server

Two Dell PE6950 servers (used as principal and mirror), each with:

- 4-socket, dual-core AMD Opteron™ processor 8220 @2.80 GHz
- 32 GB RAM

One Dell R805 server (used as client), with:

- 2-socket, quad-core AMD Opteron™ processor 2354 @2.20 GHz
- 32 GB RAM

Storage

One 3PAR SAN, with:

- 240 disks, each 147 GB, 10K RPM
- 12 ports, 2 directly attached per server
- All LUNs are striped across all disks
- 16 GB data cache, 4 GB control cache; data cache is dynamically allocated based on I/O patterns

Purpose	**Drive**	**RAID**	**# LUNs**	**Total GB**
Data	M:	1+0	1	2,000
Log	N:	1+0	1	500

Software

- The 64-bit edition of Windows Server® 2008 Enterprise with Service Pack 1
- The 64-bit edition of SQL Server 2008 Enterprise with Service Pack 1

Asynchronous Database Mirroring with Log Compression in SQL Server 2008

Author: Sanjay Mishra

Reviewers: Prem Mehra, Mike Ruthruff, Michael Thomassy, Peter Byrne, Kaloian Manassiev, Burzin Patel, Juergen Thomas, Tom Davidson

Overview

With asynchronous database mirroring, committing a transaction doesn't wait for the log records to be sent to the mirror. Transactions are committed once the log records are written to the log disk on the principal. Therefore, with asynchronous database mirroring, you will not observe significant impact on throughput resulting from the log compression feature.

With asynchronous database mirroring, log compression helps reduce the send queue. In this study, we took a customer workload (an application at a stock exchange that captures stock quotes and orders), with a high log generation rate (12 MB/sec) and ran performance tests with a pre-release build of SQL Server 2008. The results indicate reduced send queue with log compression.

Test Workload and Test Environment

The workload used for the tests described in this paper consists of a 20 GB database, and 20 concurrent, active user connections.

The test hardware consisted of 2 Unisys x64 servers with 16 processors and 64 GB RAM each. The storage was EMC Clariion SAN. The network between two servers is 1 Gbps, and we used network emulation software to simulate varied network bandwidth between the two servers.

The SQL Server 2008 prerelease build 10.0.1068.0 was used.

Asynchronous database mirroring was used for these tests.

Reduced Send Queue with Log Compression

With asynchronous database mirroring, transactions are committed once the log records are written to the log disk on the principal. The log records are then asynchronously sent to the mirror. Therefore, there could be some log records at the principal that have not yet been sent to the mirror. Unsent log that has accumulated on the principal is known as the send queue. This is reflected in the perfmon counter "Send Queue KB" of the "Database Mirroring" object.

The principal could be generating transaction log records at a rate faster than it can send these log records to the mirror over the network. Under lower bandwidth networks, the send queue can be more pronounced. With asynchronous database mirroring, the send queue represents the data that can be lost if the principal goes down. Therefore, reducing the send queue means reducing the potential data loss.

Compressing the log stream in SQL Server 2008 allows more transaction log records being packed in each packet send to the mirror, and thereby reduces the send queue.

With SQL Server 2008, the log stream is compressed by default. To measure the impact of log compression, we used the trace flag 1462, which disables log compression. Disabling log compression is equivalent of SQL Server 2005 behavior. Figure 1 illustrates the average send queue with and without log compression at various values of network bandwidth.

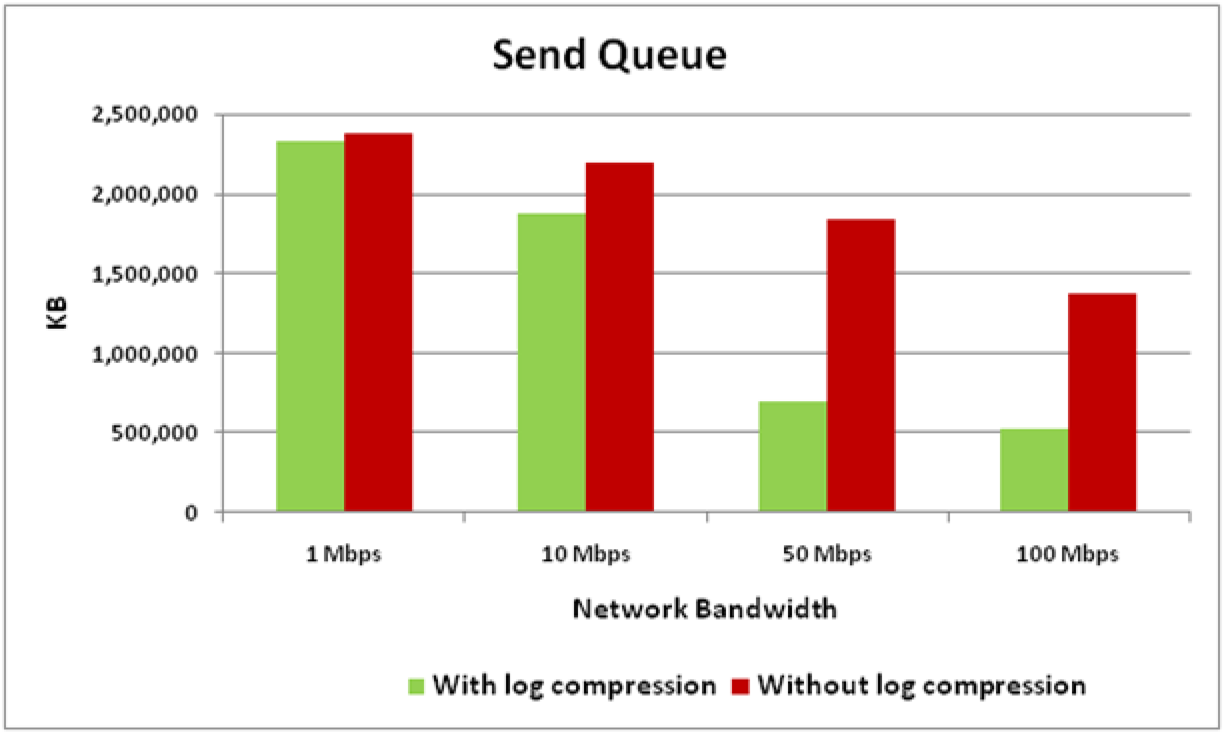


Figure 1: Reduced send queue with log compression

As shown in Figure 1, the log compression results in lesser send queue compared to disabling log compression.

Log Compression Ratio

Log compression ratio indicates the factor by which the log stream has been compressed. One way to express the log compression ratio is to divide the perfmon counter Log Bytes Sent/sec by the counter Log Compressed Bytes Sent/sec. Another way to express the log compression ratio is to compute it as a percent using the following formula:

(Log Bytes Sent/sec- Log Compressed Bytes Sent/sec)*100 / (Log Bytes Sent/sec)

Figure 2 illustrates the percent log compression ratio obtained in our test workload.

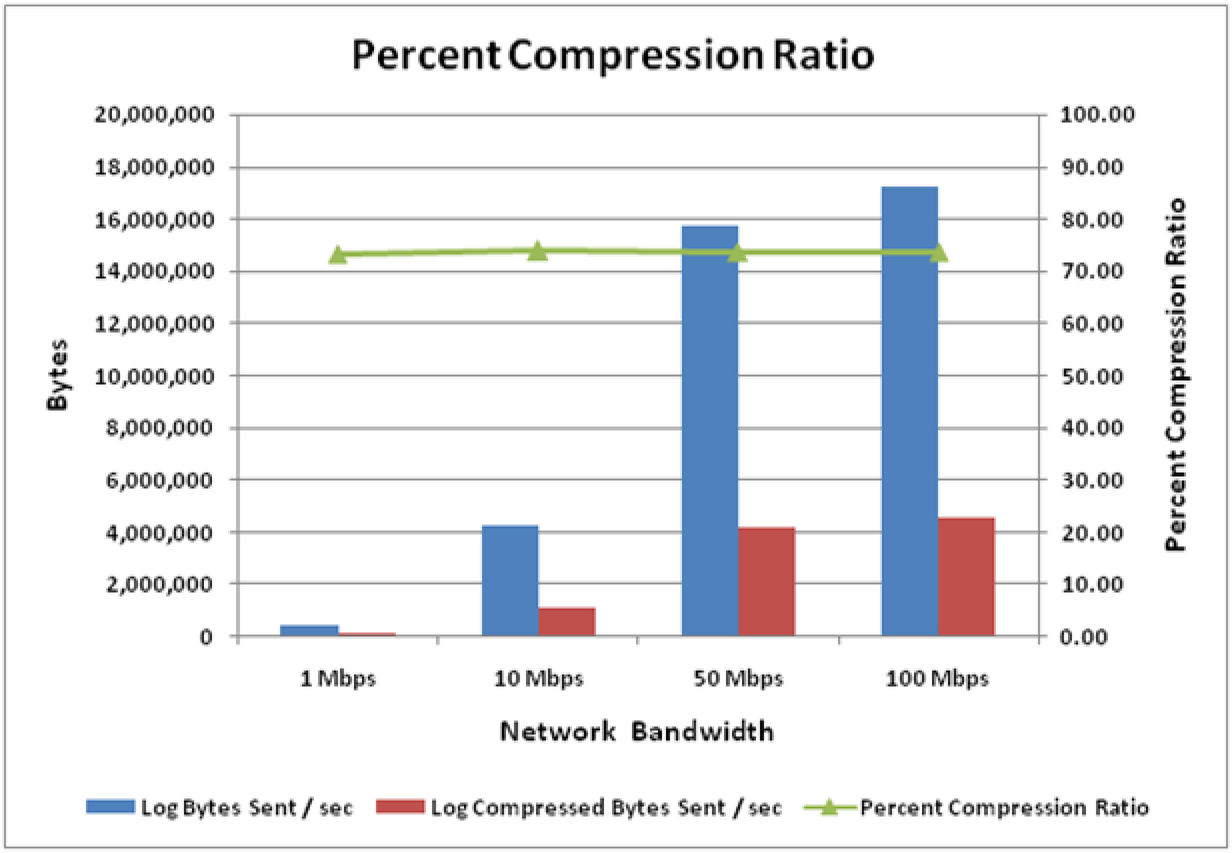


Figure 2: Log compression ratio

The compression ratio depends upon the application and the data it processes. The compression ratio is not an externally configurable parameter; it is an inherent property of the data. As you can observe in Figure 2, the percentage compression achieved in our test workload is around 73% -- the network bandwidth has no impact on the compression ratio.

The transaction log records generated by our test workload provide a very high percent compression ratio – 73%. Not all applications will exhibit as much compression. If the amount of compression achieved is less than 12.5%, then the compressed log stream will not be sent to the mirror; instead the uncompressed log stream will be sent. Sending the uncompressed log stream means that the mirror doesn't need to uncompress the log stream it received, thereby saving some CPU resources (CPU cost of log stream compression is discussed in the next section) on the mirror.

Please note that while deciding whether to send the compressed log stream or the uncompressed log stream, SQL Server computes the percentage compression on a per packet basis – it doesn't use the perfmon counters shown in Figure 2. The perfmon counters reflect the aggregated average of log bytes sent over time.

Cost of Log Compression

The benefits of log compression come with some processing cost. The log records are compressed on the principal before being sent to the mirror, where they are uncompressed before being applied to the mirror database. The extra tasks of compressing and uncompressing the log adds some processing overhead on the principal and mirror respectively, resulting in higher CPU usage. Figure 3 illustrates the CPU usage of the principal and mirror servers at various network bandwidths, with and without log compression.

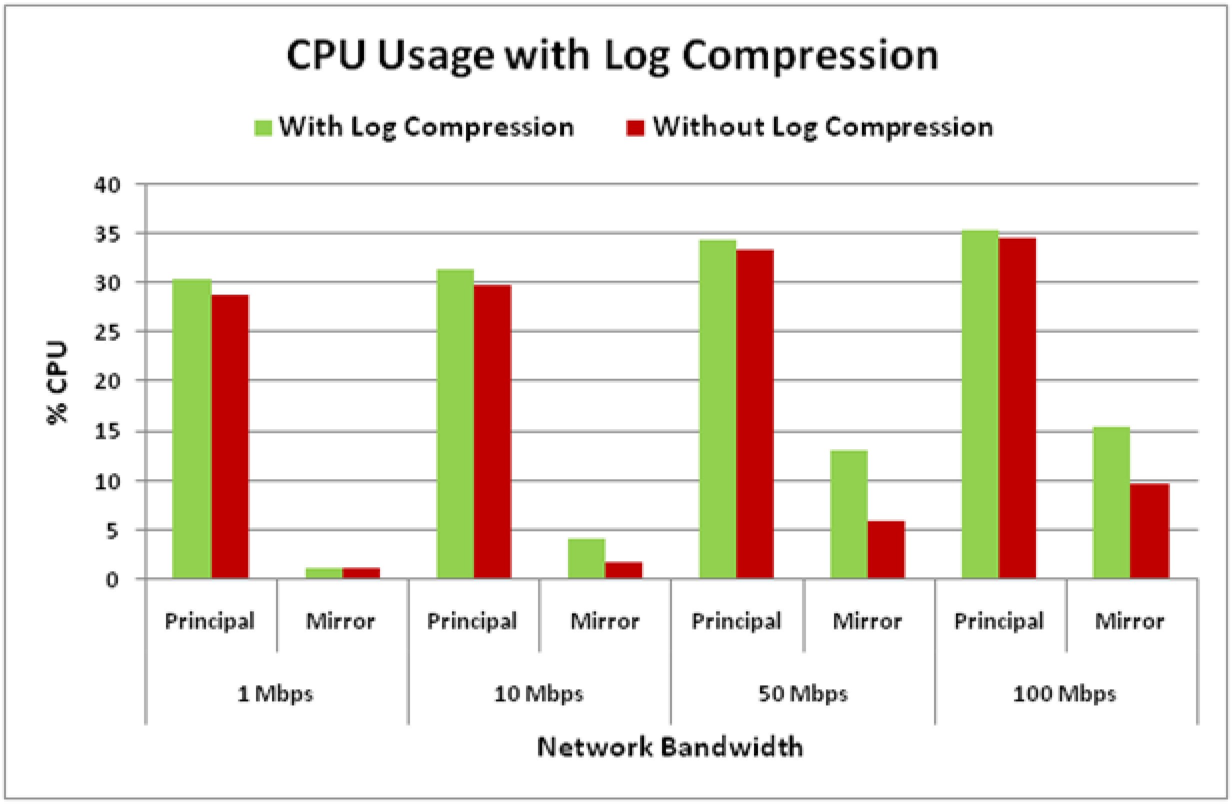


Figure 3: CPU Usage with log compression

As illustrated in Figure 3, the CPU utilization with log compression is slightly more than that without log compression.

As discussed in the earlier section “Log Compression Ratio”, if the compression ratio achieved is less than 12.5%, then the uncompressed log stream will be sent to the mirror. In that case, the principal will incur the CPU overhead, but the mirror will not.

Time to Flush the Send Queue

With asynchronous database mirroring over a low bandwidth network, when you execute a log intensive task (a task that generates huge amount of transaction log), it may take a while for the send queue to die down after the task has finished. You could see the send queue rise during the log intensive task, and then coming down after the task is finished. If the send queue is monotonically increasing during the normal operations, then you may be having severe network capacity (network latency and/or network bandwidth) limitations for the given workload.

Log compression helps not only in reducing the send queue at any given time, it also helps reducing the time it takes for the send queue to die down after a log intensive task. Figure 4 illustrates the time it took to flush the send queue after a log intensive task over a 10Mbps network, with and without log compression. As it is obvious from Figure 4, the log compression feature in SQL Server 2008 drastically reduces this time.

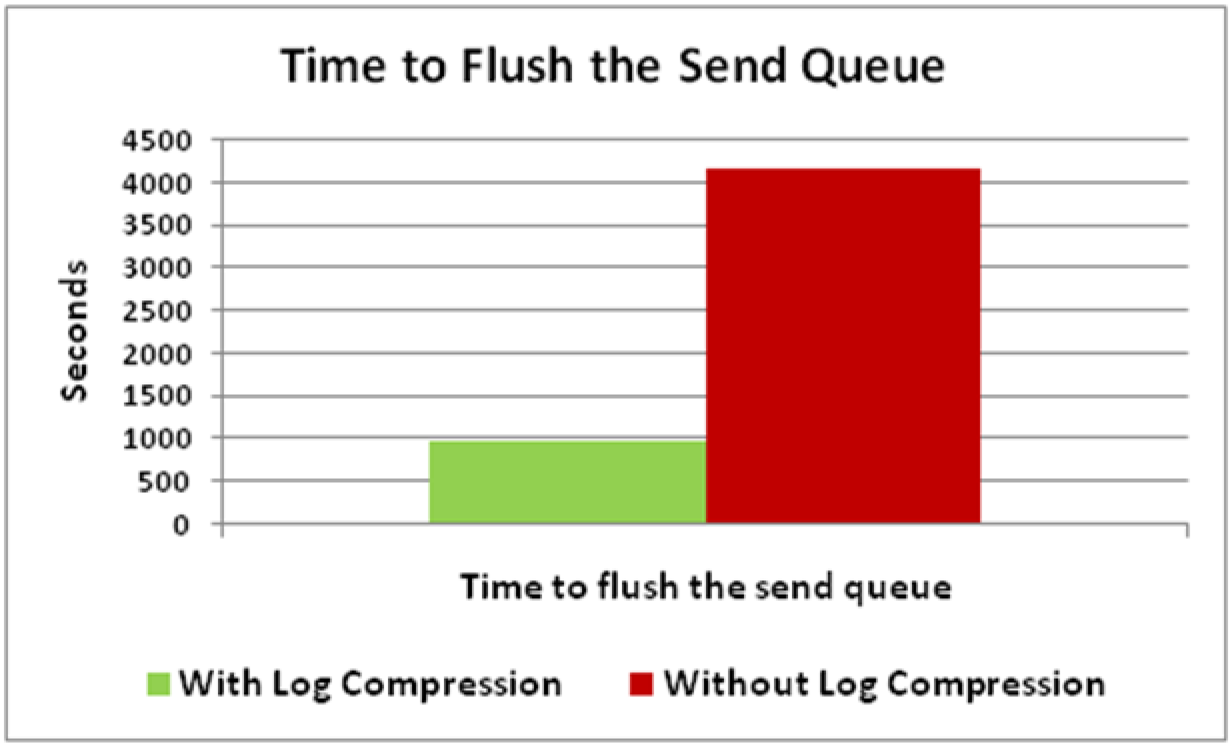


Figure 4: Time to flush the send queue with 10 Mbps network bandwidth

Summary

SQL Server 2008 introduces a new feature that compresses the log stream sent from the principal to the mirror in a database mirroring configuration. Compressing the log stream with asynchronous database mirroring results in reduced send queue. Log compression causes the CPU utilization to increase on the principal as well as the mirror. The reduction in send queue as well as the increase in CPU load are both dependent upon the workload and the network capacity. It is recommended that you test with appropriate workload to estimate the expected impact of log compression in your environment.