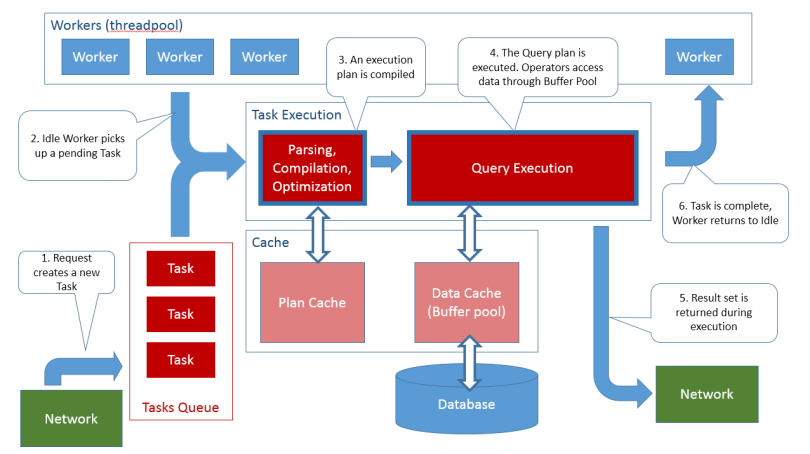
**SQL Server Architecture**



SQL Server retrieves threads from Windows. The SQL Server configuration setting max worker threads (set at instance level) determines how many threads will be retrieved. SQL Server has its own internal scheduling system, independent of the scheduling performed by the operating system.

Instead of using Windows threads directly, SQL Server creates a pool of worker threads that are mapped to Windows threads whenever work needs to be performed.  
When a SQL Server component needs to execute code, the component creates a task that represents the unit of work to be done.

For example, if you send a batch of Transact-SQL commands to the server, it’s  
likely that the batch will be executed within a task.  
When a SQL Server component creates a task, it is assigned the next available worker thread that is not in use. If no worker threads are available, SQL Server will try to retrieve another Windows thread, up to the point that the max worker threads configuration limit is reached.

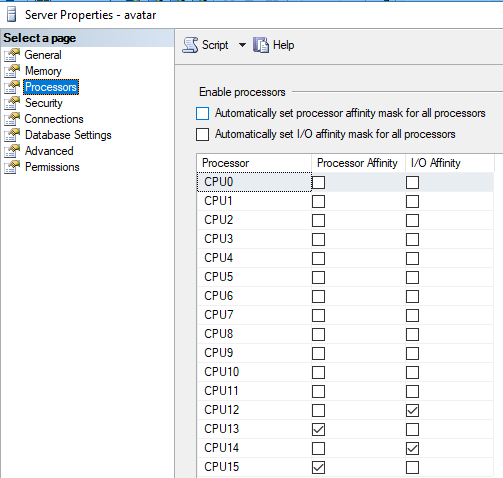
At that point, the new task would need to wait to get a worker thread. THREADPOOL wait type is when there are no available threads in the server’s thread pool, during parallelism multiple worker threads are required. So, this wait type ir related with CXPACKET.

I suggest reducing parallelism not setting MAXDOP 1. Try to increase **Cost threshold for parallelism** if the setting is too low (default is 5) and analyze the queries with parallelism, see the indexes and statistics to understand if the query optimizer is estimating right.

**AffinityI/O mask**

On a computer that has more than one processor, the **affinity mask** and the **affinity**I/O **mask** options are used to designate which CPUs are used by **SQL Server**. Enabling a CPU with both the **affinity mask** and the **affinity** I/O **mask** can slow performance by forcing the processor to be overused.

**Allocate CPU core to SQL Server**



**Best Practices Recommendations**

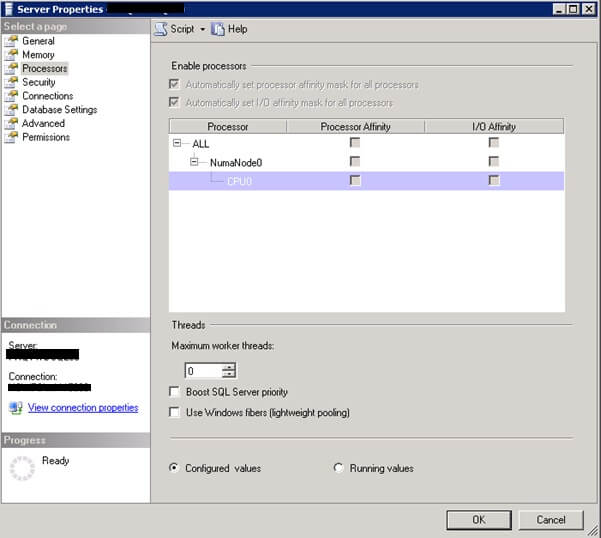
When you specify either the affinity mask or the affinity I/O mask options, you should specify both, but only enable each CPU no more than once.

Do not enable the same CPU in both the affinity mask option and the affinity I/O mask option. The bits that correspond to each CPU should be in one of the following states:

* 0 in both the affinity mask option and the affinity I/O mask option
* 0 in the affinity mask option and 1 in the affinity I/O mask option
* 1 in the affinity mask option and 0 in the affinity I/O mask option

## Checking SQL Server Processor Affinity

Before adding a new vCPU, let's check the SQL Server Processor properties page to see how SQL Server is setup for this 1 vCPU. Right click on the instance name in SSMS, select Properties and go to the Processors page.  The checkbox "Automatically set processor affinity mask for all processors" is grayed out because there is only 1 vCPU available. Hence, SQL Server will not allow the processor affinity mask to be set.

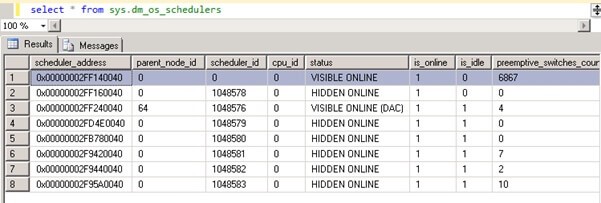


## Checking SQL Server Schedulers

Check the SQL Server Schedulers DMV to confirm there is only one scheduler (Scheduler ID 0) that can accept user requests. The DMV sys.dm\_os\_schedulers returns one row per scheduler in SQL Server and each scheduler is mapped to an individual processor.  Schedulers with ID numbers less than 1048576 are used to schedule regular queries from users and ID numbers greater or equal to 1048576 are used internally by SQL Server such as the Dedicated Admin Connection.

SELECT \* FROM sys.dm\_os\_schedulers

GO



## Check SQL Server Affinity Using T-SQL

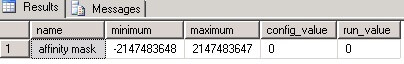
Checking the SQL Server affinity mask confirms SQL Server is configured to utilize all vCPU(s) indicated by the value 0 as shown below.

EXEC sp\_configure 'show advanced options', 1

RECONFIGURE

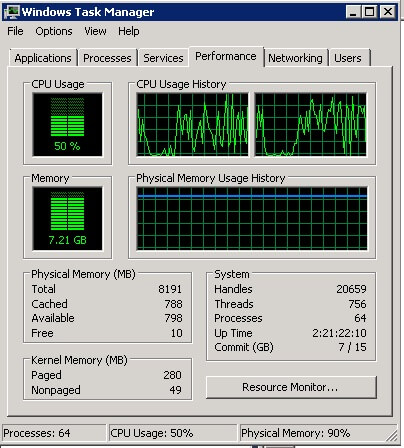
EXEC sp\_configure 'affinity mask'

GO



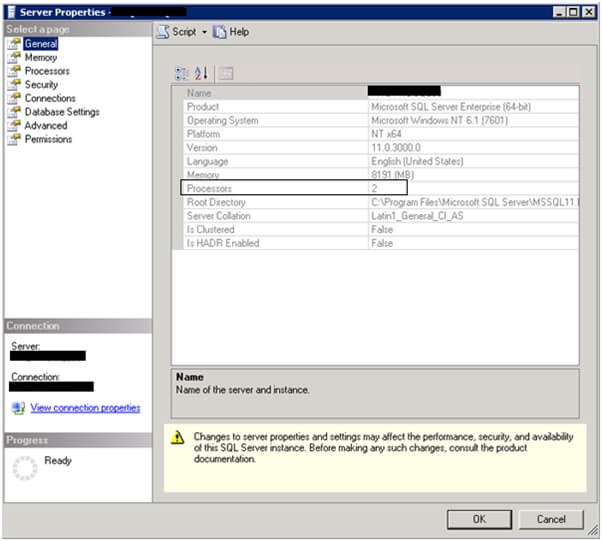
## Adding a New vCPU to SQL Server

Now the VM administrator hot-adds 1 more vCPU to the virtual server. This can be done online without interrupting any running processes on the server. After hot-added the vCPU can be seen in Windows Task Manager as expected. Let's stop the workload query that we ran above and re-execute.  Now it seems both vCPUs have a "balanced" load, but they are only being utilized 50% of the time. This is expected since SQL Server is still unable to utilize the hot-added vCPU.



## Check Number of Processors SQL Server Can See

In SSMS, if we right click on the server name, select Properties and go to the General page we can check the SQL Server instance properties and see that SQL Server is now showing 2 vCPUs.



SQL Server schedulers are the component within SQLOS that schedules CPU time for a task. If we re-check the steps above we will see that SQL Server still sees 1 vCPU and 1 scheduler.  Hence, SQL Server will not be able to utilize the hot-added vCPU yet because there isn't a scheduler that can assign a workload to the hot-added vCPU.

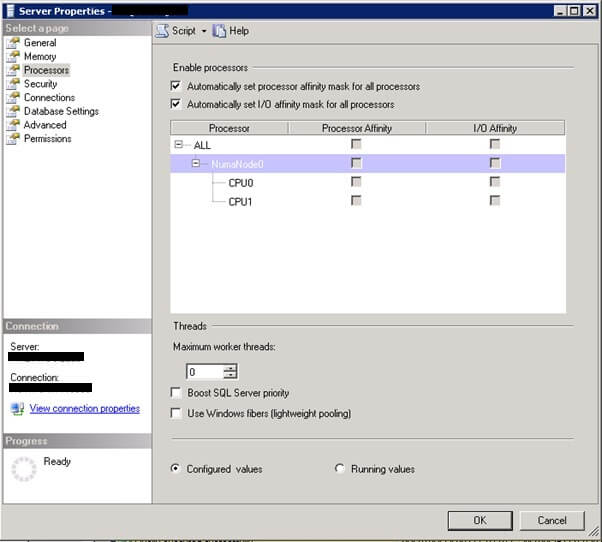
## Run Reconfigure for SQL Server to See Hot-added vCPU

The command below will trigger SQL Server to detect the hot-added vCPU as a resource

RECONFIGURE

GO

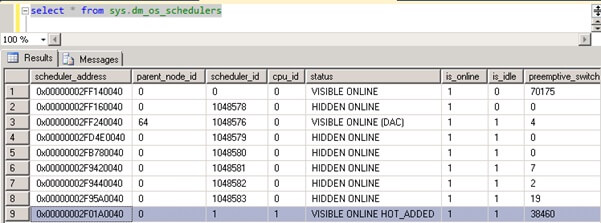
Check the SQL Server Processors properties page again and the hot-added vCPU appears as CPU1.



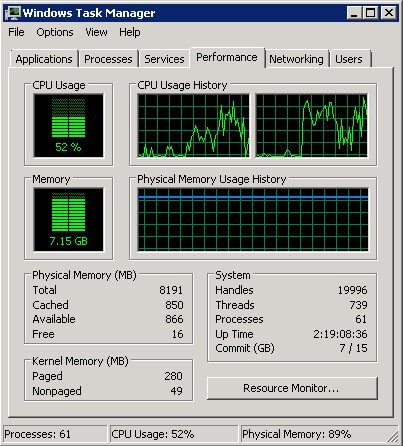
Execute the query below to check the SQL Server Schedulers DMV again, SQL Server has now created a scheduler (Scheduler ID 1) for the hot-added vCPU with the description VISIBLE ONLINE HOT\_ADDED in the status column.

SELECT \* FROM sys.dm\_os\_schedulers

GO



Stop and re-execute the workload query. It still appears the workload behavior is similar where we only have 50% utilization. Even though an additional vCPU is presented to Windows Server, SQL Server somehow does not seem to be utilizing the hot-added vCPU.



The workload query is an intentional flaw introduced to execute the query in serial mode or single threaded. How do I know? The workload running the CROSS JOIN is on session id 59. If I execute the query below, it will provide the query details and number of threads created to execute the workload. In my case, only 1 thread is running the workload, essentially meaning that the query is running serially on Scheduler ID 1.

SELECT

STasks.session\_id, SThreads.os\_thread\_id, Txt.text, Sch.scheduler\_id

FROM sys.dm\_os\_tasks AS STasks

INNER JOIN sys.dm\_os\_threads AS SThreads

ON STasks.worker\_address = SThreads.worker\_address

INNER JOIN sys.dm\_os\_schedulers Sch

ON Sch.scheduler\_address = SThreads.scheduler\_address

INNER JOIN sys.dm\_exec\_requests Req

ON Req.session\_id = Stasks.session\_id

CROSS APPLY sys.dm\_exec\_sql\_text(Req.sql\_handle) Txt

WHERE STasks.session\_id = 59

GO

# Memory Management Architecture

SQL Server Memory Architecture:

SQL Server dynamically acquires and frees memory as required. Typically, an administrator does not have to specify how much memory should be allocated to SQL Server, although the option still exists and is required in some environments.

One of the primary design goals of all database software is to minimize disk I/O because disk reads and writes are among the most resource-intensive operations. SQL Server builds a buffer pool in memory to hold pages read from the database. Much of the code in SQL Server is dedicated to minimizing the number of physical reads and writes between the disk and the buffer pool. SQL Server tries to reach a balance between two goals:

* Keep the buffer pool from becoming so big that the entire system is low on memory.
* Minimize physical I/O to the database files by maximizing the size of the buffer pool.

### Providing the maximum amount of memory to SQL Server

By using AWE and the Locked Pages in Memory privilege, you can provide the following amounts of memory to the SQL Server Database Engine.

| **32-bit 1** | **64-bit** |
| --- | --- |
| Conventional memory | All SQL Server editions. Up to process virtual address space limit:  - 2 GB - 3 GB with /3gb boot parameter 2  - 4 GB on WOW64 3 | All SQL Server editions. Up to process virtual address space limit:  - 7 TB with IA64 architecture (IA64 not supported in SQL Server 2012 (11.x) and above) - Operating system maximum with x64 architecture 4 |
| AWE mechanism (Allows SQL Server to go beyond the process virtual address space limit on 32-bit platform.) | SQL Server Standard, Enterprise, and Developer editions: Buffer pool is capable of accessing up to 64 GB of memory. | Not applicable 5 |
| Lock pages in memory operating system (OS) privilege (allows locking physical memory, preventing OS paging of the locked memory.) 6 | SQL Server Standard, Enterprise, and Developer editions: Required for SQL Server process to use AWE mechanism. Memory allocated through AWE mechanism cannot be paged out.  Granting this privilege without enabling AWE has no effect on the server. | Only used when necessary, namely if there are signs that sqlservr process is being paged out. In this case, error 17890 will be reported in the Errorlog, resembling the following example: A significant part of sql server process memory has been paged out. This may result in a performance degradation. Duration: #### seconds. Working set (KB): ####, committed (KB): ####, memory utilization: ##%. |

1 32-bit versions are not available starting with SQL Server 2014 (12.x).  
2 /3gb is an operating system boot parameter. For more information, visit the MSDN Library.  
3 WOW64 (Windows on Windows 64) is a mode in which 32-bit SQL Server runs on a 64-bit operating system.  
4 SQL Server Standard Edition supports up to 128 GB. SQL Server Enterprise Edition supports the operating system maximum.  
5 Note that the sp\_configure awe enabled option was present on 64-bit SQL Server, but it is ignored.  
6 If lock pages in memory privilege (LPIM) is granted (either on 32-bit for AWE support or on 64-bit by itself), we recommend also setting max server memory.

**Changes to Memory Management starting with SQL Server 2012 (11.x)**

In earlier versions of SQL Server ( SQL Server 2005 (9.x), SQL Server 2008 and SQL Server 2008 R2), memory allocation was done using five different mechanisms:

* **Single-page Allocator (SPA)**, including only memory allocations that were less than, or equal to 8-KB in the SQL Server process. The *max server memory (MB)* and *min server memory (MB)* configuration options determined the limits of physical memory that the SPA consumed. THe buffer pool was simultaneously the mechanism for SPA, and the largest consumer of single-page allocations.
* **Multi-Page Allocator (MPA)**, for memory allocations that request more than 8-KB.
* **CLR Allocator**, including the SQL CLR heaps and its global allocations that are created during CLR initialization.
* Memory allocations for **thread stacks** in the SQL Server process.
* **Direct Windows allocations (DWA)**, for memory allocation requests made directly to Windows. These include Windows heap usage and direct virtual allocations made by modules that are loaded into the SQL Server process. Examples of such memory allocation requests include allocations from extended stored procedure DLLs, objects that are created by using Automation procedures (sp\_OA calls), and allocations from linked server providers.

The following table indicates whether a specific type of memory allocation is controlled by the *max server memory (MB)* and *min server memory (MB)* configuration options:

| **Type of memory allocation** | **SQL Server 2005 (9.x), SQL Server 2008 and SQL Server 2008 R2** | **Starting with SQL Server 2012 (11.x)** |
| --- | --- | --- |
| Single-page allocations | Yes | Yes, consolidated into "any size" page allocations |
| Multi-page allocations | No | Yes, consolidated into "any size" page allocations |
| CLR allocations | No | Yes |
| Thread stacks memory | No | No |
| Direct allocations from Windows | No | No |

Starting with SQL Server 2012 (11.x), SQL Server might allocate more memory than the value specified in the max server memory setting. This behavior may occur when the ***Total Server Memory (KB)*** value has already reached the ***Target Server Memory (KB)*** setting (as specified by max server memory). If there is insufficient contiguous free memory to meet the demand of multi-page memory requests (more than 8 KB) because of memory fragmentation, SQL Server can perform over-commitment instead of rejecting the memory request.

As soon as this allocation is performed, the *Resource Monitor* background task starts to signal all memory consumers to release the allocated memory, and tries to bring the *Total Server Memory (KB)* value below the *Target Server Memory (KB)* specification. Therefore, SQL Server memory usage could briefly exceed the max server memory setting. In this situation, the *Total Server Memory (KB)* performance counter reading will exceed the max server memory and *Target Server Memory (KB)* settings.

This behavior is typically observed during the following operations:

* Large Columnstore index queries.
* Columnstore index (re)builds, which use large volumes of memory to perform Hash and Sort operations.
* Backup operations that require large memory buffers.
* Tracing operations that have to store large input parameters.

## Dynamic Memory Management

The default memory management behavior of the SQL Server Database Engine is to acquire as much memory as it needs without creating a memory shortage on the system. The SQL Server Database Engine does this by using the Memory Notification APIs in Microsoft Windows.

When SQL Server is using memory dynamically, it queries the system periodically to determine the amount of free memory. Maintaining this free memory prevents the operating system (OS) from paging. If less memory is free, SQL Server releases memory to the OS. If more memory is free, SQL Server may allocate more memory. SQL Server adds memory only when its workload requires more memory; a server at rest does not increase the size of its virtual address space.

The following query returns information about currently allocated memory:

SELECT

physical\_memory\_in\_use\_kb/1024 AS sql\_physical\_memory\_in\_use\_MB,

large\_page\_allocations\_kb/1024 AS sql\_large\_page\_allocations\_MB,

locked\_page\_allocations\_kb/1024 AS sql\_locked\_page\_allocations\_MB,

virtual\_address\_space\_reserved\_kb/1024 AS sql\_VAS\_reserved\_MB,

virtual\_address\_space\_committed\_kb/1024 AS sql\_VAS\_committed\_MB,

virtual\_address\_space\_available\_kb/1024 AS sql\_VAS\_available\_MB,

page\_fault\_count AS sql\_page\_fault\_count,

memory\_utilization\_percentage AS sql\_memory\_utilization\_percentage,

process\_physical\_memory\_low AS sql\_process\_physical\_memory\_low,

process\_virtual\_memory\_low AS sql\_process\_virtual\_memory\_low

FROM sys.dm\_os\_process\_memory;

| **SQL Server Architecture** | **OS Architecture** | **Stack Size** |
| --- | --- | --- |
| x86 (32-bit) | x86 (32-bit) | 512 KB |
| x86 (32-bit) | x64 (64-bit) | 768 KB |
| x64 (64-bit) | x64 (64-bit) | 2048 KB |
| IA64 (Itanium) | IA64 (Itanium) | 4096 KB |

2 CLR memory is managed under max\_server\_memory allocations starting with SQL Server 2012 (11.x).

SQL Server uses the memory notification API **QueryMemoryResourceNotification** to determine when the SQL Server Memory Manager may allocate memory and release memory.

When SQL Server starts, it computes the size of virtual address space for the buffer pool based on a number of parameters such as amount of physical memory on the system, number of server threads and various startup parameters. SQL Server reserves the computed amount of its process virtual address space for the buffer pool, but it acquires (commits) only the required amount of physical memory for the current load.

The instance then continues to acquire memory as needed to support the workload. As more users connect and run queries, SQL Server acquires the additional physical memory on demand. A SQL Server instance continues to acquire physical memory until it either reaches its max server memory allocation target or the OS indicates there is no longer an excess of free memory; it frees memory when it has more than the min server memory setting, and the OS indicates that there is a shortage of free memory.

As other applications are started on a computer running an instance of SQL Server, they consume memory and the amount of free physical memory drops below the SQL Server target. The instance of SQL Server adjusts its memory consumption. If another application is stopped and more memory becomes available, the instance of SQL Server increases the size of its memory allocation. SQL Server can free and acquire several megabytes of memory each second, allowing it to quickly adjust to memory allocation changes.

## Buffer management

The primary purpose of a SQL Server database is to store and retrieve data, so intensive disk I/O is a core characteristic of the Database Engine. And because disk I/O operations can consume many resources and take a relatively long time to finish, SQL Server focuses on making I/O highly efficient. Buffer management is a key component in achieving this efficiency. The buffer management component consists of two mechanisms: the **buffer manager** to access and update database pages, and the **buffer cache** (also called the **buffer pool**), to reduce database file I/O.

### How buffer management works

A buffer is an 8 KB page in memory, the same size as a data or index page. Thus, the buffer cache is divided into 8 KB pages. The buffer manager manages the functions for reading data or index pages from the database disk files into the buffer cache and writing modified pages back to disk. A page remains in the buffer cache until the buffer manager needs the buffer area to read in more data. Data is written back to disk only if it is modified. Data in the buffer cache can be modified multiple times before being written back to disk. For more information, see Reading Pages and Writing Pages.

When SQL Server starts, it computes the size of virtual address space for the buffer cache based on a number of parameters such as the amount of physical memory on the system, the configured number of maximum server threads, and various startup parameters. SQL Server reserves this computed amount of its process virtual address space (called the memory target) for the buffer cache, but it acquires (commits) only the required amount of physical memory for the current load. You can query the **bpool\_commit\_target** and **bpool\_committed columns**in the sys.dm\_os\_sys\_info catalog view to return the number of pages reserved as the memory target and the number of pages currently committed in the buffer cache, respectively.

The interval between SQL Server startup and when the buffer cache obtains its memory target is called ramp-up. During this time, read requests fill the buffers as needed. For example, a single 8 KB page read request fills a single buffer page. This means the ramp-up depends on the number and type of client requests. Ramp-up is expedited by transforming single page read requests into aligned eight page requests (making up one extent). This allows the ramp-up to finish much faster, especially on machines with a lot of memory.

### Disk I/O

The buffer manager only performs reads and writes to the database. Other file and database operations such as open, close, extend, and shrink are performed by the database manager and file manager components.

Disk I/O operations by the buffer manager have the following characteristics:

* All I/Os are performed asynchronously, which allows the calling thread to continue processing while the I/O operation takes place in the background.
* All I/Os are issued in the calling threads unless the affinity I/O option is in use. The affinity I/O mask option binds SQL Server disk I/O to a specified subset of CPUs. In high-end SQL Server online transactional processing (OLTP) environments, this extension can enhance the performance of SQL Server threads issuing I/Os.
* Multiple page I/Os are accomplished with scatter-gather I/O, which allows data to be transferred into or out of noncontiguous areas of memory. This means that SQL Server can quickly fill or flush the buffer cache while avoiding multiple physical I/O requests.

#### Long I/O requests

The buffer manager reports on any I/O request that has been outstanding for at least 15 seconds. This helps the system administrator distinguish between SQL Server problems and I/O subsystem problems. Error message 833 is reported and appears in the SQL Server error log as follows:

SQL Server has encountered ## occurrence(s) of I/O requests taking longer than 15 seconds to complete on file [##] in database [##] (#). The OS file handle is 0x00000. The offset of the latest long I/O is: 0x00000.

A long I/O may be either a read or a write; it is not currently indicated in the message. Long-I/O messages are warnings, not errors. They do not indicate problems with SQL Server but with the underlying I/O system. The messages are reported to help the system administrator find the cause of poor SQL Server response times more quickly, and to distinguish problems that are outside the control of SQL Server. As such, they do not require any action, but the system administrator should investigate why the I/O request took so long, and whether the time is justifiable.

#### Causes of Long-I/O Requests

A long-I/O message may indicate that an I/O is permanently blocked and will never complete (known as lost I/O), or merely that it just has not completed yet. It is not possible to tell from the message which scenario is the case, although a lost I/O will often lead to a latch timeout.

Long I/Os often indicate a SQL Server workload that is too intense for the disk subsystem. An inadequate disk subsystem may be indicated when:

* Multiple long I/O messages appear in the error log during a heavy SQL Server workload.
* Perfmon counters show long disk latencies, long disk queues, or no disk idle time.

Long I/Os may also be caused by a component in the I/O path (for example, a driver, controller, or firmware) continually postponing servicing an old I/O request in favor of servicing newer requests that are closer to the current position of the disk head. The common technique of processing requests in priority based upon which ones are closest to the current position of the read/write head is known as "elevator seeking." This may be difficult to corroborate with the Windows System Monitor (PERFMON.EXE) tool because most I/Os are being serviced promptly. Long I/O requests can be aggravated by workloads that perform large amounts of sequential I/O, such as backup and restore, table scans, sorting, creating indexes, bulk loads, and zeroing out files.

Isolated long I/Os that do not appear related to any of the previous conditions may be caused by a hardware or driver problem. The system event log may contain a related event that helps to diagnose the problem.

### Memory pressure detection

Memory pressure is a condition resulting from memory shortage, and can result in:

* Extra I/Os (such as very active lazy writer background thread)
* Higher recompile ratio
* Longer running queries (if memory grant waits exist)
* Extra CPU cycles

This situation can be trigerred by external or internal causes. External causes include:

* Available physical memory (RAM) is low. This causes the system to trim working sets of currently running processes, which may result in overall slowdown. SQL Server may reduce the commit target of the buffer pool and start trimming internal caches more often.
* Overall available system memory (which includes the system page file) is low. This may cause the system to fail memory allocations, as it is unable to page out currently allocated memory. Internal causes include:
* Responding to the external memory pressure, when the SQL Server Database Engine sets lower memory usage caps.
* Memory settings were manually lowered by reducing the max server memory configuration.
* Changes in memory distribution of internal components between the several caches.

The SQL Server Database Engine implements a framework dedicated to detecting and handling memory pressure, as part of its dynamic memory management. This framework includes the backgroud task called **Resource Monitor**. The Resource Monitor task monitors the state of external and internal memory indicators. Once one of these indicators changes status, it calculates the corresponding notification and it broadcasts it. These notifications are internal messages from each of the engine components, and stored in ring buffers.

Two ring buffers hold information relevant to dynamic memory management:

* The Resource Monitor ring buffer, which tracks Resource Monitor activity like was memory pressure signaled or not. This ring buffer has status information depending on the current condition of RESOURCE\_MEMPHYSICAL\_HIGH, RESOURCE\_MEMPHYSICAL\_LOW, RESOURCE\_MEMPHYSICAL\_STEADY, or RESOURCE\_MEMVIRTUAL\_LOW.
* The Memory Broker ring buffer, which contains records of memory notifications for each Resource Governor resource pool. As internal memory pressure is detected, low memory notification is turned on for components that allocate memory, to trigger actions meant to balance the memory between caches.

Memory brokers monitor the demand consumption of memory by each component and then based on the information collected, it calculates and optimal value of memory for each of these components. There is a set of brokers for each Resource Governor resource pool. This information is then broadcast to each of the components, which grow or shrink their usage as required.

#### Checksum Protection

Checksum protection, introduced in SQL Server 2005 (9.x), provides stronger data integrity checking. A checksum is calculated for the data in each page that is written, and stored in the page header. Whenever a page with a stored checksum is read from disk, the database engine recalculates the checksum for the data in the page and raises error 824 if the new checksum is different from the stored checksum. Checksum protection can catch more errors than torn page protection because it is affected by every byte of the page, however, it is moderately resource intensive. When checksum is enabled, errors caused by power failures and flawed hardware or firmware can be detected any time the buffer manager reads a page from disk.

**Transaction log**

These **logs** will be written **to** and read from the **transaction log**file sequentially. **SQL Server** allows us **to** create multiple **SQL Server transaction log** files on each database. ... The number and the size of the VLFs affect the performance of the database startup, backup and restore operations

SQL Server allows us to create multiple SQL Server transaction log files on each database. Each transaction log file is divided internally into multiple **Virtual Log Files**, also known as VLFs. The size and number of the VLFs on each transaction log file is dynamic, where the SQL Server Database Engine starts with the least possible number of VLFs on the transaction log file and extend it, based on the defined increment, when the file runs out of free space.

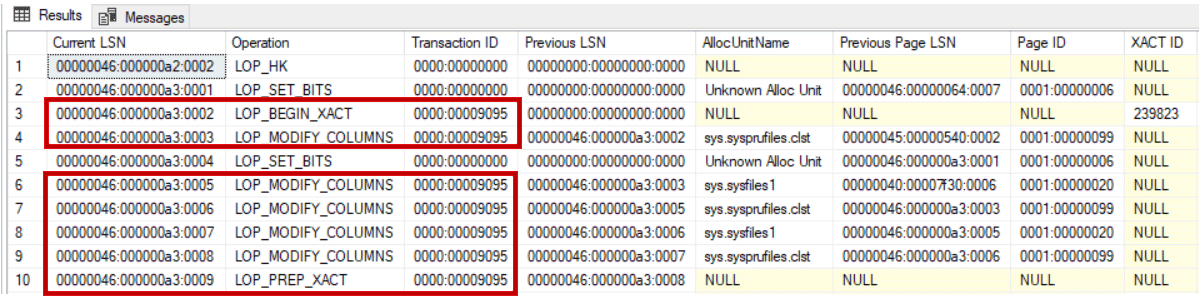
The number and the size of the VLFs affect the performance of the database startup, backup and restore operations. To override these performance issues, we should tune both the SQL transaction log file initial size and auto-growth increment properly. We will cover the VLFs subject completely in the next article. It is not always recommended to have multiple transaction log files in your database, as it may impact the performance of your database, due to writing the data sequentially and not in parallel. You can create another transaction log file as a workaround, in case of running the current hosting disk drive out of free space.

Transaction log LSN

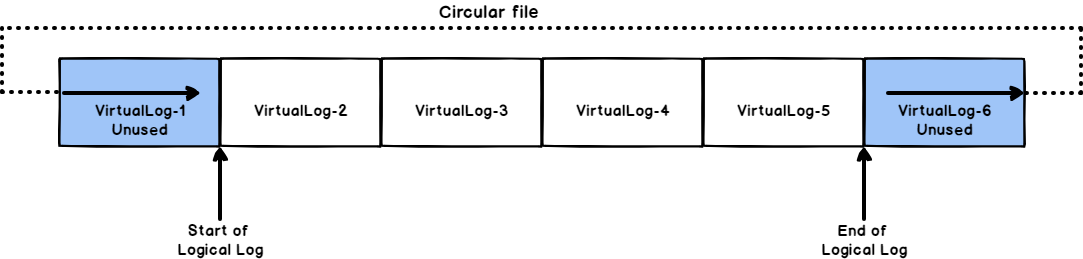
Each Transaction log record that is written to the SQL Server transaction log file can be identified by its **Log Sequence Number** (LSN). When the database is created, the Database Engine starts writing at the beginning of the logical transaction log file, which is the beginning of the actual physical transaction log file, and mark the end of the written log as the end of the logical log file. When a new transaction is performed, the log records will be written serially to the end of the logical transaction log file, with an LSN value higher than the LSN value of the previous log record.

The serially inserted log records contain other useful information, such as the ID of the transaction that this record belongs to. In this case, all log records associated with a specific transaction will be grouped and linked in a chain based on the transaction ID, that speed the rollback process of that transaction. For example, querying the **sys.fn\_dblog** system DMO using the following script shows us number of log records, with different LSNs, that belong to the same transaction, and linked together using the Transaction ID value, as shown clearly below:

|  |
| --- |
| SELECT [Current LSN],[Operation] ,[Transaction ID],[Previous LSN] ,[AllocUnitName],[Previous Page LSN],  [Page ID],[XACT ID],[Begin Time],[End Time]  FROM sys.fn\_dblog (NULL, NULL) |



You can imagine the SQL Server transaction log file as a circular tape. When the end of the logical log reaches the end of the actual physical log, the Database Engine will write the new log by wrapping it around the beginning of the actual log file, in a circular way, or to the next transaction log file, if the database consists of multiple transaction log file, as shown below:

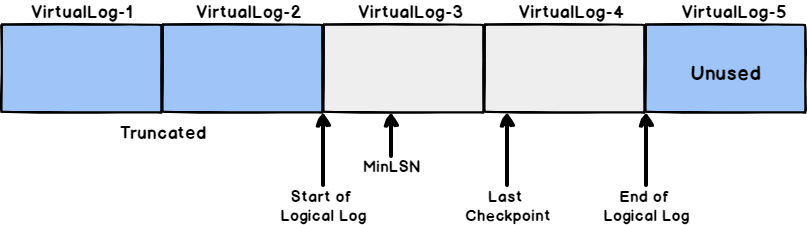


If the end of the logical log reaches the start of the logical log, due to one of the following reasons, the SQL Server Database Engine will return error number **9002**, as there is no room available for the new transaction log record to be written in the SQL Server transaction log file:

* No proper truncate process is performed.
* Auto-growth option is not enabled.
* Auto-growth enabled but the disk drive is running out of free space.

MinLSN

The **Minimum transaction log Sequence Number,**also known as MinLSN,is a special type of LSN, thatshows the LSN of the oldest active log record that is required to perform a successful database rollback process. The portion of the SQL Server transaction log file between the MinLSN and the end of the logical log that is required for the full database recovery, is called the **Active Log**, as shown below:



Truncation

Log truncation process deletes all inactive VLFs from the SQL Server transaction log file. No part of the active log can ever be truncated. VLF is the smallest unit of truncation in the transaction log file. If there is one **active** log record within an VLF, the overall VLF will be considered as part of the active log.

To be able to truncate the SQL Server transaction log:

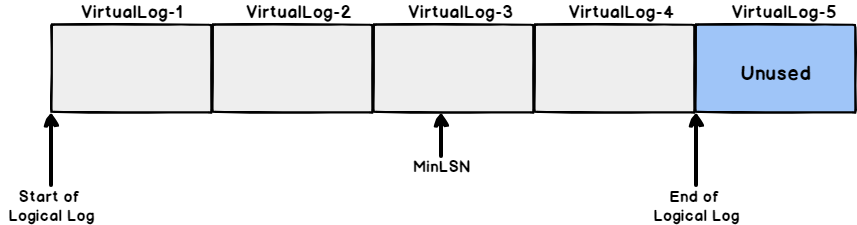
* The transaction should be committed
* The transaction log is not pending any backup or high availability feature
* A Checkpoint operator should be triggered to mark the inactive portion of the transaction log as reusable

When a data insertion or modification is performed on your database, the Database Engine keeps the performed change in the buffer pool memory, rather than applying it directly to the database files. In this way, it will perform less frequent I/O operations. The data pages that are stored in the buffer pool memory but not reflected yet to the database files are known as **Dirty Pages**. The process used by the Database Engine to reflect the dirty pages to the database files periodically is called **Checkpoints**.

* *For detailed information about the checkpoints, see Database checkpoints – Enhancements in SQL Server 2016.*

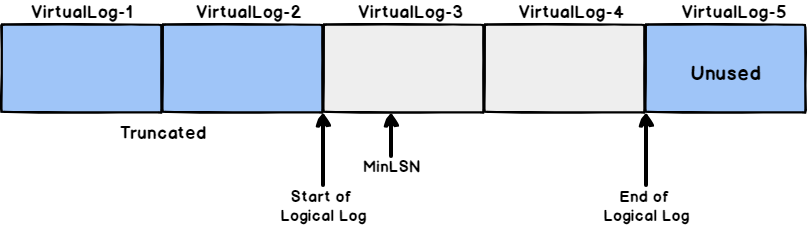
When a log truncation process is performed, the Database Engine will free all the inactive log records, starting from the beginning of the logical log toward the MinLSN, for reuse by the actual physical log. For example, the below SQL Server transaction log file contains:

* Physical log with 5 VLFs
* Logical log, which is the used part of the physical log, occupies the first four VLFs
* The first two VLFs contains inactive logs that cannot be used for now
* The second two VLFs (VLF 3 and VLF 4) contain active log records that cannot be truncated



After performing a truncate process on the previous SQL Server transaction log file, you will see that:

* The first two VLFs, that contain inactive log records, are truncated
* VLF1 and VLF2 are available now for reuse again
* VLF1 and VLF2 are no longer part from the logical log
* No change performed on VLF3 and VLF4 that are contain active log records



For now, we are familiar with the internal structure of the SQL transaction log file, its importance and how it works.

**DBCC HELP:**

* Provides syntax for a specific DBCC command, or lists all commands.
* By default, only supported commands listed
* This command has no impact on performance or data
* Requires sysadmin role

We have two options to run this DBCC Command:

When you just run it as it is, it will list all of the supported DBCC commands

|  |
| --- |
| DBCC HELP ('?'); |

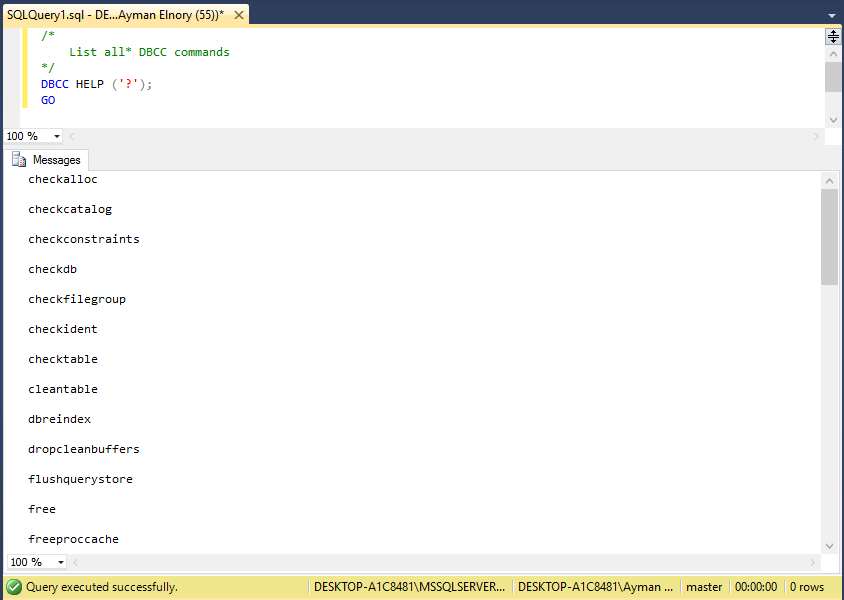


Figure 1 DBCC HELP command

When you run it, providing a specific command, it will give you the syntax for that:

|  |
| --- |
| DBCC HELP (CHECKDB);  GO |

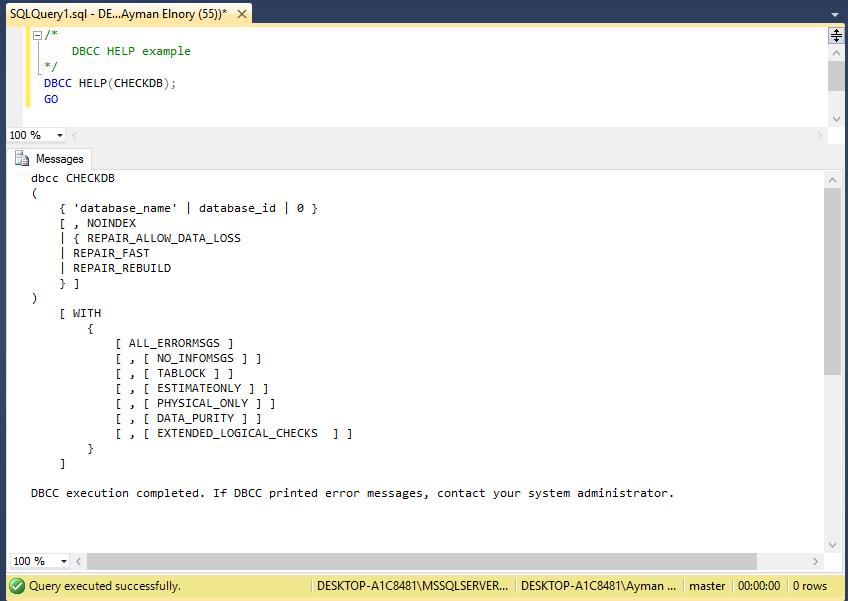


Figure 2 DBCC HELP (CHECKDB) command

It will be very helpful if you don’t have internet access to get easy the syntax of any DBCC command just execute DBCC Help.

**DBCC TRACEON, DBCC TRACEOFF, and DBCC TRACESTATUS:**

If you have worked with trace flags before, you probably know that you can apply them as a startup parameter for the SQL Server service. Well, that is all well and good until you realize that for them to take effect, you have to restart the service. That is not always something you can easily do so that I can list some of the useful features of these commands:

* DBCC TRACEON and DBCC TRACEOFF can be used to enable/disable a trace flag without requiring a service restart.
* With DBCC TRACEON and DBCC TRACEOFF, you can enable/disable trace flags at either the session or the global level.
* Be aware that when you enable or disable a trace flag, it’s quite possible that you can affect performance, depending on what functionality the trace flag changes.
* Both commands (DBCC TRACEON and DBCC TRACEOFF) require the sysadmin role.
* DBCC TRACESTATUS provides status for a specific trace flag, or all of them, and notes whether they are enabled for a session or globally.
* Running DBCC TRACESTATUS does not affect performance or data, nor does it alter the configuration of the instance.
* DBCC TRACESTATUS only needs the public role to be run.

Now, Let us see some examples of using these commands:

Use the following code to list all trace flags enabled just for this connection:

|  |
| --- |
| DBCC TRACESTATUS ();  GO |

Use this code to list all trace flags enabled globally:

|  |
| --- |
| DBCC TRACESTATUS (-1);  GO |

Follow the following scenario to test both of DBCC TRACEON and DBCC TRACEOFF

Run a full backup operation for one of your databases:

|  |
| --- |
| DECLARE @BackupPath NVARCHAR (100);  SET @BackupPath = 'C:\Backup\SQL\_Shack2014\_' +  REPLACE (CONVERT (nvarchar (19), SYSDATETIME (), 126), ':','') + '.bak';    BACKUP DATABASE [SQL\_Shack2014]  TO DISK = @BackupPath  WITH NOFORMAT,  INIT,  COMPRESSION,  STATS = 05;  GO |

Then run a transaction log backup operation on the same database:

|  |
| --- |
| DECLARE @BackupPath NVARCHAR (100);  SET @BackupPath = 'C:\Backup\SQL\_Shack2014\_' +  REPLACE (CONVERT (nvarchar (19), SYSDATETIME (), 126), ':','') + '.trn';    BACKUP LOG [SQL\_Shack2014]  TO DISK = @BackupPath  WITH NOFORMAT,  INIT,  COMPRESSION,  STATS = 05;  GO |

Now, when you check the ERRORLOG, you will notice entries to the SQL Server error log for every successful backup operation (local connection) as in the following figure:

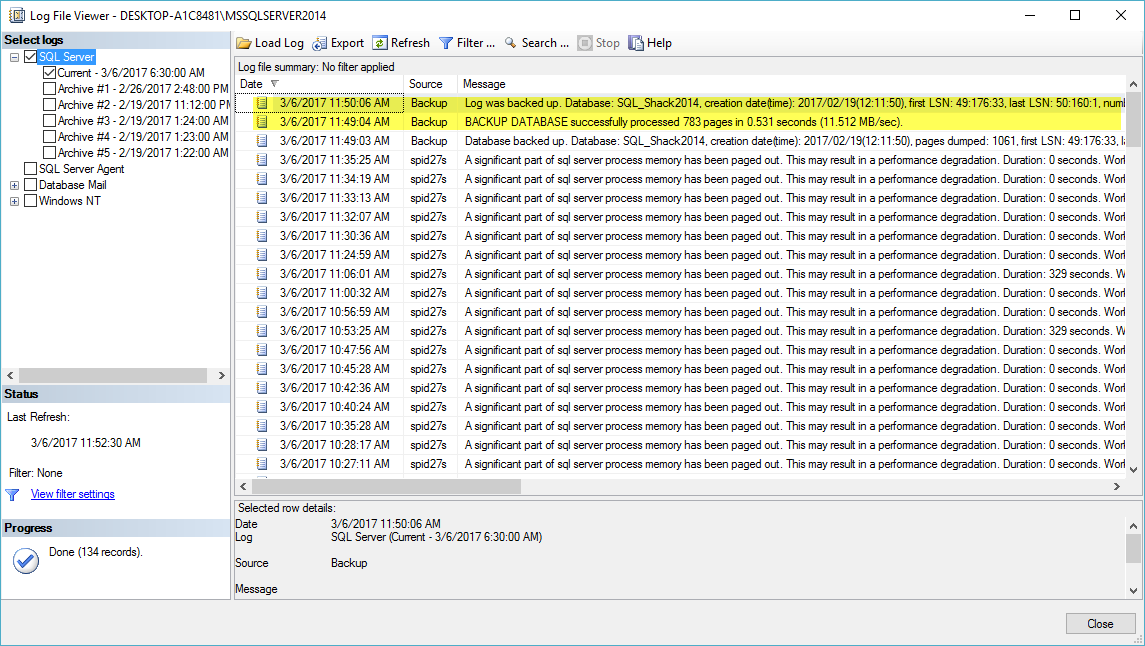
[](https://www.sqlshack.com/wp-content/uploads/2017/03/word-image-266.png)

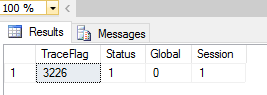
Figure 3 Log file viewer (1)

We can suppress all successful in SQL Server error log by enabling trace flag 3226 using DBCC TRACEON command:

|  |
| --- |
| DBCC TRACEON (3226);  GO |

Then we can use DBCC TRACESTATUS command to verify that the trace is enabled:

|  |
| --- |
| DBCC TRACESTATUS ();  GO |



Run a transaction log Backup operation again and check SQL Server log to see the impact of enabling this trace flag:

|  |
| --- |
| DECLARE @BackupPath NVARCHAR(100);  SET @BackupPath = 'C:\Backup\SQL\_Shack2014\_' +  REPLACE (CONVERT (nvarchar (19), SYSDATETIME (), 126), ':','') + '.trn';    BACKUP LOG [SQL\_Shack2014]  TO DISK = @BackupPath  WITH NOFORMAT,  INIT,  COMPRESSION,  STATS = 05;  GO |

You will notice that there is a log record for enabling the trace flag 3226, and the last transaction log backup has not logged as the following figure:

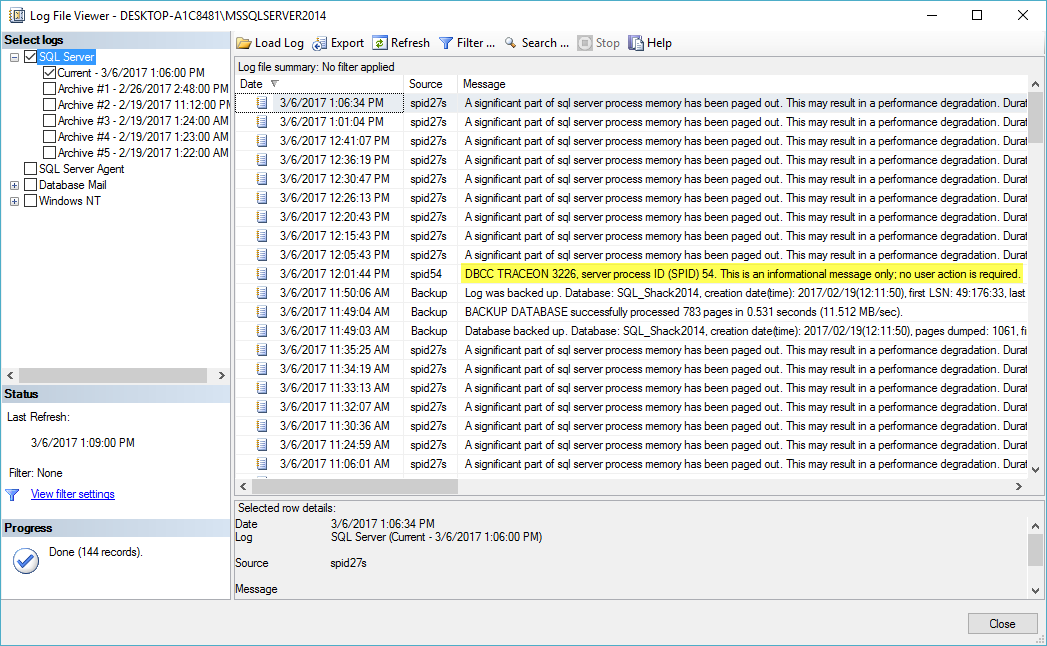
[](https://www.sqlshack.com/wp-content/uploads/2017/03/word-image-268.png)

Figure 4 Log file viewer (2)

You can disable this trace flag using the following code:

|  |
| --- |
| DBCC TRACEOFF (3226);  GO |

To make these commands affect on SQL server instance globally which will be effective till the next restart of the instance, you can use the following codes:

|  |
| --- |
| /\*  Enable trace flag (3226) globally  \*/  DBCC TRACEON (3226, -1);  GO  /\*  Verify again that the trace flag is enabled globally  \*/  DBCC TRACESTATUS (-1);  GO  /\*  Turn off 3226 globally  \*/  DBCC TRACEOFF (3226,-1);  GO |

Informational Commands:

These commands will not change the configuration of our environment, but they will give us information about our environment.

**DBCC SQLPERF:**

* ProvidesTransaction log space usage for all log files on an instance
* Used to clear out data related to weight statistics, latch statistics, or spinlock statistics
* To run this command, you need to have( **VIEW SERVER STATE, ALTER SERVER STATE** )
* Execution does not affect system performance

To check log space utilization:

|  |
| --- |
| DBCC SQLPERF (LOGSPACE); |

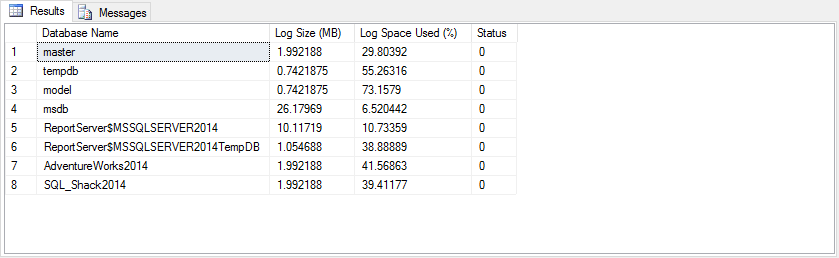
[](https://www.sqlshack.com/wp-content/uploads/2017/03/word-image-269.png)

Figure 5 Result of DBCC SQLPERF Command

To clear wait statistics:

|  |
| --- |
| DBCC SQLPERF ("sys.dm\_os\_wait\_stats", CLEAR); |

To clear latch statistics:

|  |
| --- |
| DBCC SQLPERF ("sys.dm\_os\_latch\_stats", CLEAR); |

To clear spinlock statistics:

|  |
| --- |
| DBCC SQLPERF ("sys.dm\_os\_spinlock\_stats", CLEAR); |

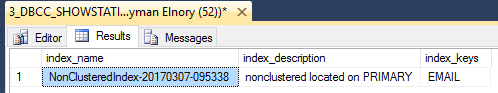
**DBCC SHOW\_STATISTICS:**

* Displays statistics object for an index, indexed view, or column statistic
* Use when troubleshooting and understanding the estimates from a plan and comparing against actual values
* Viewing statistics does not introduce a performance load on the system
* Require sysadmin, db\_owner, or db\_ddladmin roles, or tablet ownership

To implement this DBCC Command go through the following steps:

First, you need to view all indexes contained in the specified table:

|  |
| --- |
| USE [SQL\_Shack2014];  GO  sp\_helpindex '[dbo]. [Students]';  GO |



Then you can use DBCC SHOW\_STATISTICS against the index name you got in step 1:

|  |
| --- |
| DBCC SHOW\_STATISTICS ('[dbo]. [Students]','NonClusteredIndex-20170307-095338');  GO |

The result displays the header, histogram, and density vector based on data stored in the statistics object.

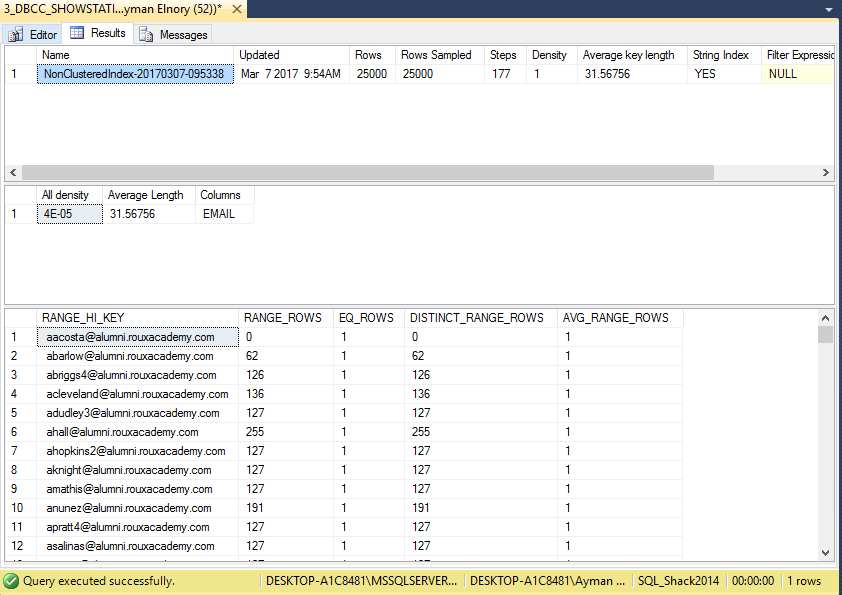
[](https://www.sqlshack.com/wp-content/uploads/2017/03/word-image-271.png)

Figure 6 Result of DBCC SHOW\_STATISTICS Command

And you can specify what section of the result you want to see by using one of the following options:

|  |
| --- |
| DBCC SHOW\_STATISTICS ('[dbo]. [Students]','NonClusteredIndex-20170307-095338') WITH STAT\_HEADER;  GO    DBCC SHOW\_STATISTICS ('[dbo]. [Students]','NonClusteredIndex-20170307-095338') WITH DENSITY\_VECTOR;  GO    DBCC SHOW\_STATISTICS ('[dbo]. [Students]','NonClusteredIndex-20170307-095338') WITH HISTOGRAM;  GO |

**DBCC USEROPTIONS:**

* Returns options set for the current connection like isolation level or QUOTED\_IDENTIFIER
* Use to verify setting for connection to confirm they are correct, or consistent across different connect methods
* Only displays information, does not modify settings
* Require public role

|  |
| --- |
| DBCC USEROPTIONS;  GO |

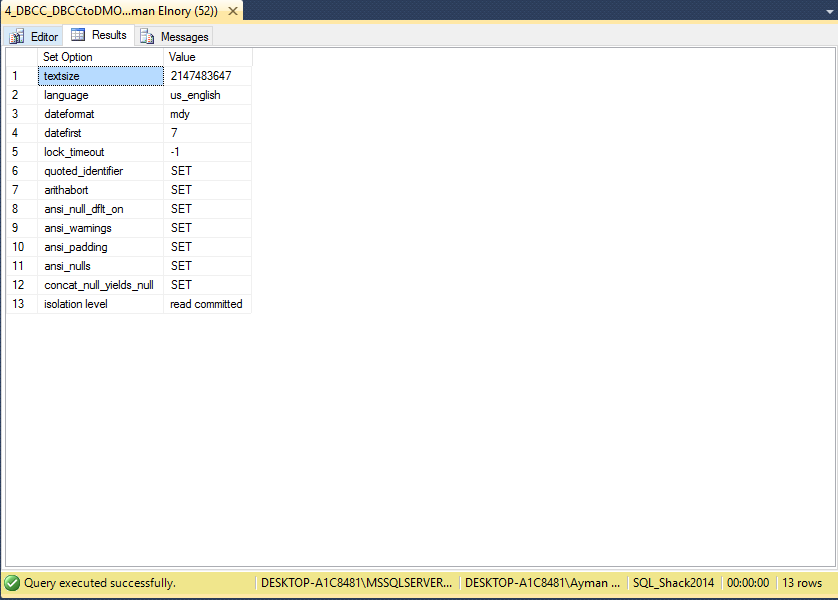
[](https://www.sqlshack.com/wp-content/uploads/2017/03/word-image-272.png)

Figure 7 Result of DBCC USEROPTIONS Command

DBCC LOGINFO

DBCC SQLPERF(LOGSPACE)