Slow-Running Queries

Slow-running or long-running queries can contribute to excessive resource consumption. They can be the consequence of blocked queries.

Excessive resource consumption is not restricted to CPU resources, but it can also include I/O storage bandwidth and memory bandwidth. Even if SQL Server queries are designed to avoid full table scans by restricting the result set with a reasonable WHERE clause, they might not perform as expected if there is not an appropriate index supporting that particular query. Also, WHERE clauses can be dynamically constructed by applications, depending on the user input. Given this, existing indexes cannot cover all possible cases of restrictions. Excessive CPU, I/O, and memory consumption by Transact-SQL statements are covered earlier in this white paper.

In addition to missing indexes, there may be indexes that are not used. Because all indexes have to be maintained, this does not impact the performance of a query, but it does impact the DML queries.

Queries can also run slowly because of wait states for logical locks and for system resources that are blocking the query. The cause of the blocking can be a poor application design, bad query plans, the lack of useful indexes, and a SQL Server instance that is improperly configured for the workload.

This section focuses on two causes of a slow-running query—blocking and index problems.

Blocking

Blocking is primarily waits for logical locks, such as the wait to acquire an exclusive (X) lock on a resource or the waits that results from lower-level synchronization primitives such as latches.

Logical lock waits occur when a request to acquire a noncompatible lock on an already-locked resource is made. Although this is needed to provide the data consistency based on the transaction isolation level at which a particular Transact-SQL statement is running, it does give the end user a perception that SQL Server is running slowly. When a query is blocked, it is not consuming any system resources, so you will find it is taking longer but the resource consumption is low. For more information about concurrency control and blocking, see SQL Server 2008 Books Online.

Waits on lower-level synchronization primitives can result if your system is not configured to handle the workload.

The common scenarios for blocking and waits are:

- Identifying the blocker.
- Identifying long blocks.
- Blocking per object.
- Page latching issues.
- Overall performance effect of blocking using SQL Server waits.

A SQL Server session is placed in a wait state if system resources (or locks) are not currently available to service the request. In other words, a wait occurs if the resource has a queue of outstanding requests. DMVs can provide information for any sessions that are waiting on resources.

SQL Server 2008 provides detailed and consistent wait information, reporting approximately 125 wait types. The DMVs that provide this information range from **sys.dm_os_wait_statistics** for overall and cumulative waits for SQL Server to the session-specific **sys.dm_os_waiting_tasks**, which breaks down waits by session. The following DMV provides details on the wait queue of the tasks that are waiting on some resource. It is a simultaneous representation of all wait queues in the system. For example, you can find out the details about the blocked session 56 by running the following query.

```
select * from sys.dm_os_waiting_tasks where session_id=56
```

This result shows that session 56 is blocked by session 53 and that session 56 has been waiting for a lock for 1,103,500 milliseconds:

- waiting_task_address: 0x022A8898
- session_id: 56
- exec_context_id: 0
- wait_duration_ms: 1103500
- wait_type: LCK_M_S
- resource_address: 0x03696820
- blocking_task_address: 0x022A8D48
- blocking_session_id: 53
- blocking_exec_context_id: NULL
- resource_description: ridlock fileid=1 pageid=143 dbid=9 id=lock3667d00 mode=X associatedObjectId=72057594038321152

To find sessions that have been granted locks or waiting for locks, you can use the **sys.dm_tran_locks** DMV. Each row represents a currently active request to the lock manager that has either been granted or is waiting to be granted as the request is blocked by a request that has already been granted. For regular locks, a granted request indicates that a lock has been granted on a resource to the requestor. A waiting request indicates that the request has not yet been granted. For example, the following query and its output show that session 56 is blocked on the resource 1:143:3 that is held in X mode by session 53.

```
select
    request_session_id as spid,
    resource_type as rt,
    resource_database_id as rdb,
    (case resource_type
      WHEN 'OBJECT' then object_name(resource_associated_entity_id)
      WHEN 'DATABASE' then ' '
      ELSE (select object_name(object_id)
            from sys.partitions
            where hobt_id=resource_associated_entity_id)
    END) as objname,
```

```
    resource_description as rd,
    request_mode as rm,
    request_status as rs
from sys.dm_tran_locks
```

Here is the sample output.

```
spid     rt           rdb         objname       rd            rm
rs
-------------------------------------------------------------------------
---
56    DATABASE    9                                S          GRANT
53    DATABASE    9                                S          GRANT
56    PAGE        9       t_lock      1:143        IS         GRANT
53    PAGE        9       t_lock      1:143        IX         GRANT
53    PAGE        9       t_lock      1:153        IX         GRANT
56   OBJECT       9       t_lock                   IS         GRANT
53   OBJECT       9        t_lock                  IX         GRANT
53    KEY         9        t_lock      (a400c34cb X          GRANT
53    RID         9        t_lock      1:143:3     X          GRANT
56    RID         9        t_lock      1:143:3     S         WAIT
```

Locking Granularity and Lock Escalation

One of keys to understanding blocking is the transaction isolations and the locking granularity. Transaction isolation levels govern the duration for S mode locks but they don't impact the duration of exclusive (X) locks that are held for the duration of the transaction under all transaction isolation levels. The locking granularity determines whether the lock is to be acquired at row, level, page level, or table level. Clearly, the higher the locking granularity, the lower the concurrency. For example, if a transaction takes an exclusive (X) lock on the table to modify one row in a table, it blocks other transactions that want to read or modify completely different rows. On the other hand, the benefit of higher locking granularity is that SQL Server does not need to acquire as many locks, which saves both memory and the CPU cost of acquiring and releasing the locks. The locking granularity is determined by SQL Server using a heuristic model that works pretty well, but there may be cases where does not behave as expected. For cases like this, user can use **sp_tableoption** or ALTER INDEX DDL to control the locking granularity on the object or by providing locking hints.

There is another interesting twist with locking. SQL Server can escalate the lock on a table if it determines at run time that the number of locks acquired on an object in a

statement has exceeded a threshold. A lock escalation is triggered when any of the following conditions is true:

- The number of locks held (as opposed to those that are acquired and then released; for example, when one or more rows are read under the read committed isolation level) by a statement on an index or a heap within a statement exceeds the threshold, which is set to approximately 5000 by default. These locks include intent locks as well. Note that the lock escalation will not trigger if:
    - The transaction acquires 2,500 locks each on two indexes or heaps in a single statement.
    - The transaction acquires 2,500 locks on the nonclustered index and 2,500 locks on the corresponding base table in a single statement.
    - The same heap or index is referenced more than one time in a statement; the locks on each instance of those are counted separately. So for example, in the case of a self-join on a table t1, if each instance has 3,000 locks within the statement, lock escalation is not triggered.
- The memory taken by lock resources is greater than 40% of the non-AWE (32-bit) or regular (64-bit) enabled memory if the locks configuration option is set to 0, the default value. In this case, the lock memory is allocated dynamically as needed.

When the lock escalation is triggered, SQL Server attempts to escalate the lock to table level, but the attempt may fail if there are conflicting locks. So for example, if the SH locks need to be escalated to the table level and there are concurrent exclusive (X) locks on one or more rows or pages of the target table, the lock escalation attempt fails. However, SQL Server periodically, for every 1,250 (approximately) new locks acquired by the lock owner (that is, the transaction that initiates the lock), attempts to escalate the lock. If the lock escalation succeeds, the SQL Server releases the lower-granularity locks, and the associated lock memory, on the index or the heap. Because at the time of lock escalation, there cannot be any conflicting access, a successful lock escalation can potentially lead to blocking of future concurrent access to the index or the heap by transactions in conflicting lock modes. Therefore, lock escalation is not always a good idea for all applications.

Disabling Lock Escalation

SQL Server provides the following trace flags to disable lock escalation:

- TraceFlag-1211: It disables lock escalation at the current threshold (5000) on a per-index or per-heap per-statement basis. If this trace flag is in effect, the locks are never escalated. This trace flag also instructs SQL Server to ignore the memory acquired by the lock manager up to a maximum statically allocated lock memory or 60% of non-AWE (32-bit) or regular (64-bit) dynamically allocated memory. At this time an out-of-lock-memory error is generated. This can potentially be damaging, because an application can exhaust SQL Server memory by acquiring large number of locks. This, in the worst case, can stall the server or degrade its performance to an unacceptable level. For these reasons, you must exercise caution when you use this trace flag.
- TraceFlag-1224: This trace flag is similar to trace flag 1211 with one key difference. It enables lock escalation if the lock manager acquires 40% of the statically allocated memory or 40% of the non-AWE (32-bit) or regular (64-bit) dynamically allocated memory. Additionally, if this memory cannot be allocated because other components are taking up more memory, the lock escalation can be triggered

earlier. SQL Server generates an out-of-memory error when memory allocated to the lock manager exceeds the statically allocated memory or 60% of non-AWE (32-bit) or regular (64-bit) dynamically allocated memory.

If both trace flags (1211 and 1224) are set at the same time, trace flag 1211 takes precedence. You can use the DBCC TRACESTATUS (-1) command to find the status of all trace flags enabled in SQL Server. The limitation with these trace flags is that they are coarse in granularity (that is, they are at the SQL Server instance level). SQL Server 2008 addresses this by providing an option at the table level to enable or disable lock escalation. Secondly, SQL Server 2008 provides lock escalation at the partition level so that lock escalation due to DML activities in one partition does not impact access to other partitions.

Identifying Long Blocks

As mentioned earlier, blocking in SQL Server is common and is a manifestation of the logical locks that are needed to maintain the transactional consistency. However, when the wait for locks exceeds a threshold, it may impact the response time. To identify long-running blocking, you can use the BlockedProcessThreshold configuration parameter to establish a user configured server-wide block threshold. The threshold defines a duration in seconds. Any block that exceeds this threshold will fire an event that can be captured by SQL Trace.

For example, a 200-second blocked process threshold can be configured in SQL Server Management Studio as follows:

1. Execute `sp_configure 'blocked process threshold', 200`.
2. Reconfigure with override.
3. After the blocked process threshold is established, capture the trace event. The trace events of blocking events that exceed the user configured threshold can be captured with SQL Trace or SQL Server Profiler.
4. If you are using SQL Trace, use **sp_trace_setevent** and **event_id**=137.
5. If you are using SQL Server Profiler, select the Blocked process report event class (under the Errors and Warnings object). See Figure 1.

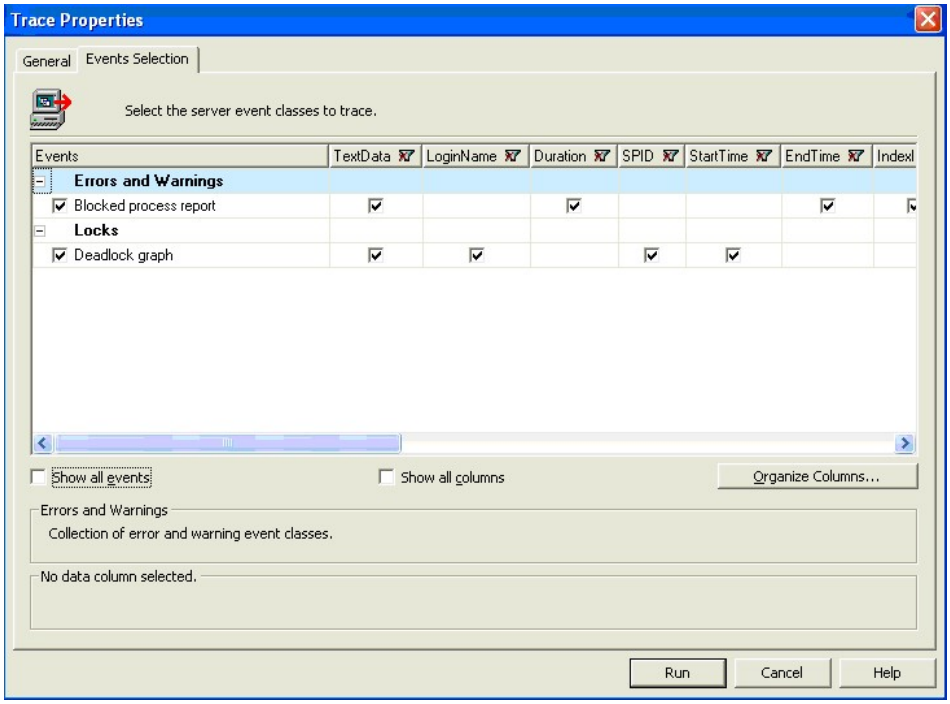

Figure 1: Tracing long blocks and deadlocks

Note This is a lightweight trace, because events are only captured when (1) a block exceeds the threshold, or (2) a deadlock occurs. For each 200-second interval that a lock is blocked, a trace event fires. This means that a single lock that is blocked for 600 seconds results in three trace events. See Figure 2.

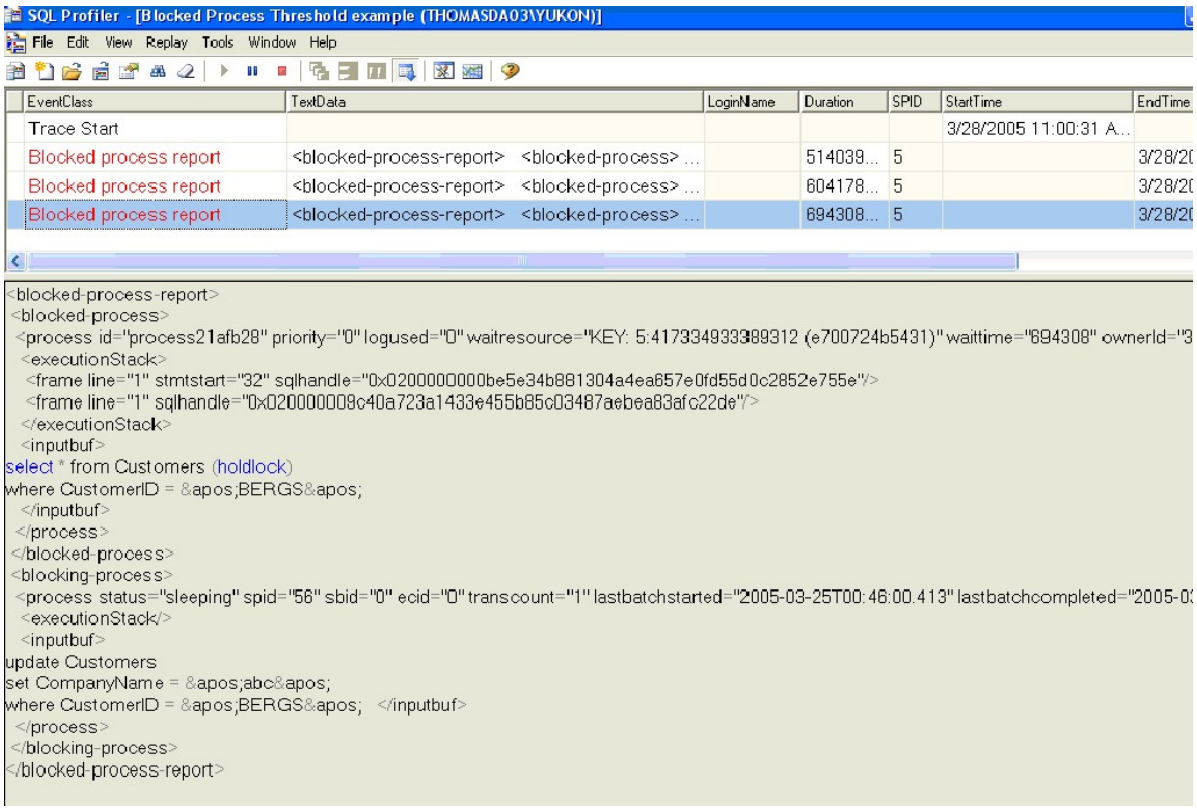

Figure 2: Reporting Blocking > block threshold

The traced event includes the entire SQL statements of both the blocker and the one blocked. In this case the UPDATE Customers statement is blocking the SELECT FROM Customers statement.

Blocking per Object with sys.dm_db_index_operational_stats

The DMV **sys.dm_db_index_operational_stats** provides comprehensive index usage statistics, including blocks. In terms of blocking, it provides a detailed accounting of locking statistics per table, index, and partition. Examples of this include a history of accesses, locks (**row_lock_count**), blocks (**process_virtual_memory_low**), and waits (**row_lock_wait_in_ms**) for a given index or table.

The type of information available through this DMV includes:

- The count of locks held, for example, row or page.
- The count of blocks or waits, for example, row or page.
- The duration of blocks or waits, for example, row or page.
- The count of page latches waits. The duration of **page_latch_wait**: This involves contention for a particular page for say, ascending key inserts. In such cases, the hot spot is the last page, so multiple writers to the same last page each try to get an exclusive page latch at same time. This will show up as Pagelatch waits.
- The duration of **page_io_latch_wait**: An I/O latch occurs when a user requests a page that is not in the buffer pool. A slow or overworked I/O subsystem can

sometimes experience high PageIOlatch waits that are actually I/O issues. These issues can be compounded by cache flushes or missing indexes.

- The duration of page latch waits.

Other types of information are also kept for access to indexes:

- Types of access, for example, range or singleton lookups.
- Inserts, updates, and deletes at the leaf level.
- Inserts, updates, deletes above the leaf level. Activity above the leaf is index maintenance. The first row on each leaf page has an entry in the level above. If a new page is allocated at the leaf, the level above will have a new entry for the first row on the new leaf page.
- Pages merged at the leaf level. These pages represent empty pages that were deallocated because rows were deleted.
- Index maintenance. Pages merged above the leaf level are empty pages that were deallocated because to rows were deleted at the leaf, thereby leaving intermediate level pages empty. The first row on each leaf page has an entry in the level above. If enough rows are deleted at the leaf level, intermediate level index pages that originally contained entries for the first row on leaf pages will be empty. This causes merges to occur above the leaf.

This information is cumulative from instance startup. The information is not retained across instance restarts, and there is no way to reset it. The data returned by this DMV exists only as long as the metadata cache object that represents the heap or index is available. The values for each column are set to zero whenever the metadata for the heap or index is brought into the metadata cache. Statistics are accumulated until the cache object is removed from the metadata cache. However, you can periodically poll this table and collect this information in table so that you can query it further.

Overall Performance Effect of Blocking Using Waits

SQL Server 2008 provides over 100 additional wait types for tracking application performance. Any time a user connection is waiting, SQL Server accumulates wait time. For example, the application requests resources such as I/O, locks, or memory and can wait for the resource to be available. This wait information is summarized and categorized across all connections so that a performance profile can be obtained for a given workload. Thus, SQL wait types identify and categorize user (or thread) waits from an application workload or user perspective.

This query lists the top 10 waits in SQL Server. These waits are cumulative but you can reset them using DBCC SQLPERF ([sys.dm_os_wait_stats], clear).

```
select top 10 *
from sys.dm_os_wait_stats
order by wait_time_ms desc
```

Following is the output. A few key points to notice are:

- Some waits are normal, such as the waits encountered by background threads such as lazy writer.
- Some sessions waited a long time to get a SH lock.

- The *signal wait* is the time between when a worker has been granted access to the resource and the time it gets scheduled on the CPU. A long signal wait may imply high CPU contention.

```
wait_type     waiting_tasks_count   wait_time_ms   max_wait_time_ms    signal_wait_time_ms
----------------- -------------------- -------------------- ------------------------------
LAZYWRITER_SLEEP   415088           415048437      1812                156
SQLTRACE_BUFFER_FLUSH 103762        415044000      4000                0
LCK_M_S        6                    25016812       23240921            0
WRITELOG       7413                 86843          187                 406
LOGMGR_RESERVE_APPEND 82            82000          1000                0
SLEEP_BPOOL_FLUSH     4948          28687          31                  15
LCK_M_X         1                   20000          20000               0
PAGEIOLATCH_SH   871                11718          140                 15
PAGEIOLATCH_UP   755                9484           187                 0
IO_COMPLETION    636                7031           203                 0
```

To analyze the wait states, you need to poll the data periodically and then analyze it later.

Session-Level Wait Statistics

To identify session-level or statement-level wait statistics, which was not possible in previous versions of SQL Server, Extended Events is an ideal tool. It is scalable and able to trace wait statistics during run time.

This example Extended Events session traces all waits, including those that are both internal and external to SQL Server, for session id 54.

```
-- sqlserver.session_id is the ID of the target session you want to trace.
I am using 54 in the example. Replace it accordingly

-- make sure C:\xevent folder exists

create event session session_waits on server

add event sqlos.wait_info

      (action (sqlserver.sql_text, sqlserver.plan_handle,
sqlserver.tsql_stack)

WHERE sqlserver.session_id=54 and duration>0)

, add event sqlos.wait_info_external

      (action (sqlserver.sql_text, sqlserver.plan_handle,
sqlserver.tsql_stack)
```

```
WHERE sqlserver.session_id=54 and duration>0)

add target package0.asynchronous_file_target

      (SET filename=N'C:\xevent\wait_stats.xel',
metadatafile=N'C:\xevent\wait_stats.xem');


alter event session session_waits on server state = start;

go


-- wait for monitored workload in target session (54 in this example) to
finish.
```

To read the results from the output file, run the following queries.

```
alter event session session_waits on server state = stop

drop event session session_waits on server


select

CONVERT(xml, event_data).value('(/event/data/text)[1]','nvarchar(50)') as
'wait_type',

CONVERT(xml, event_data).value('(/event/data/value)[3]','int') as
'duration',

CONVERT(xml, event_data).value('(/event/data/value)[6]','int') as
'signal_duration'

into #eventdata

from sys.fn_xe_file_target_read_file

(N'C:\xevent\wait_stats*.xel', N'C:\xevent\wait_stats*.xem', null, null)

-- save to temp table, #eventdata

select wait_type, SUM(duration) as 'total_duration', SUM(signal_duration)
as 'total_signal_duration'

from #eventdata

group by wait_type
```

```
drop table #eventdata

go
```

Sample output is shown here.

```
wait_type                                  total_duration       total_signal_duration

------------------------------------------------ -------------- ----------------------

NETWORK_IO                                  233                 0

PREEMPTIVE_OS_WAITFORSINGLEOBJECT           231                 576

WAITFOR                                     7000                0

PAGEIOLATCH_UP                              624                 0

PAGELATCH_UP                                2320                45

PAGELATCH_SH                                45                  10

WRITELOG                                    30                  0
```

Monitoring Index Usage

Another aspect of query performance is related to DML queries and queries that delete, insert, and modify data. The more indexes defined on a specific table, the more resources needed to modify data. In combination with locks held over transactions, longer modification operations can hurt concurrency. Therefore, it can be very important to know which indexes are used by an application over time. You can then figure out whether there is a lot of weight in the database schema in the form of indexes that never get used.

SQL Server 2008 provides the **sys.dm_db_index_usage_stats** DMV, which shows which indexes are used and whether they are in use by the user query or only by a system operation. With every execution of a query, the columns in this view are incremented according to the query plan that is used for the execution of that query. The data is collected while SQL Server is up and running. The data in this DMV is kept in memory only and is not persisted. So when the SQL Server instance is shut down, the data is lost. You can poll this table periodically and save the data for later analysis.

The operation on indexes is categorized into user type and system type. *User type* refers to SELECT and INSERT, DELETE, and UPDATE operations. *System type* operations are commands like DBCC statements, DDL commands, or update statistics. The columns for each category of statements differentiate into:

- Seek operations against an index (**user_seeks** or **system_seeks**).
- Lookup operations against an index (**user_lookups** or **system_lookups**).
- Scan operations against an index (**user_scans** or **system_scans**).
- Update operations against an index (**user_updates** or **system_updates**).

For each of these accesses of indexes, the timestamp of the last access is noted as well.

An index itself is identified by three columns covering its **database_id**, **object_id**, and **index_id**. **index_id**=0 represents a heap table, **index_id**=1 represents a clustered index, and **index_id**>1 represents a nonclustered index.

Over days of run time of an application against a database, the list of indexes getting accessed in **sys.dm_db_index_usage_stats** grows.

The rules and definitions for seek, scan, and lookup work as follows in SQL Server 2008:

- Seek: Indicates how many times the B-tree structure was used to access the data. It doesn't matter whether the B-tree structure is used just to read a few pages of each level of the index in order to retrieve one data row or whether half of the index pages are read in order to read gigabytes of data or millions of rows out of the underlying table. So you can expect most of the hits against an index to be accumulated in this category.
- Scan: Indicates how many times the data layer of the table gets used for retrieval without using one of the index B-trees. For tables that do not have any index defined, this would be the case. In the case of table with indexes defined on it, this can happen when the indexes defined on the table are of no use for the query executed against that statement.
- Lookup: Indicates that a clustered index that is defined on a table was used to look up data that was identified by seeking through a nonclustered index that is also defined on that table. This describes the scenario known as *bookmark lookup* in SQL Server 2000 or earlier. It represents a scenario where a nonclustered index is used to access a table and the nonclustered index does not cover the columns of the query SELECT list and the columns defined in the WHERE clause. SQL Server increments the value of the column **user_seeks** for the nonclustered index used plus the column **user_lookups** for the entry of the clustered index. This count can become very high if there are multiple nonclustered indexes defined on the table. If the number of user seeks against a clustered index of a table is pretty high, the number of user lookups is pretty high as well, and if the number of user seeks against one particular nonclustered index is very high as well, you should consider making the nonclustered index with the high count the clustered index.

The following DMV query can be used to obtain useful information about index usage for all objects in all databases.

```
select object_id, index_id, user_seeks, user_scans, user_lookups
from sys.dm_db_index_usage_stats
order by object_id, index_id
```

You can see the following results for a given table.

```
object_id       index_id    user_seeks    user_scans    user_lookups
-----------------------------------------------------------------------
521690298        1          0             251           123
521690298        2          123           0             0
```

In this case, there were 251 executions of a query directly accessing the data layer of the table without using one of the indexes. There were 123 executions of a query accessing the table by using the first nonclustered index, which does not cover either the SELECT list of the query or the columns specified in the WHERE clause, because we see 123 lookups on the clustered index.

The most interesting columns of **sys.dm_db_index_usage_stats** to look at are the user type columns, including **user_seeks** and **user_scans**. System usage columns like **system_seeks** can be seen as a result of the existence of the index. If the index did not exist, it would not have to be updated in statistics and it would not need to be checked for consistency. Therefore, the analysis needs to focus on the four columns that indicate usage by ad hoc statements or by the user application.

To get information about the indexes of a specific table that has not been used since the last start of SQL Server, this query can be executed in the context of the database that owns the object.

```
select i.name
from sys.indexes i
where i.object_id=object_id('<table_name>') and
    i.index_id NOT IN  (select s.index_id
                        from sys.dm_db_index_usage_stats s
                        where s.object_id=i.object_id and
                        i.index_id=s.index_id and
                        database_id = <dbid> )
```

All indexes that haven't been used yet can be retrieved with the following statement.

```
select object_name(object_id), i.name
from sys.indexes i
where  i.index_id NOT IN (select s.index_id
                          from sys.dm_db_index_usage_stats s
                          where s.object_id=i.object_id and
                          i.index_id=s.index_id and
                          database_id = <dbid> )
order by object_name(object_id) asc
```

In this case, the table name and the index name are sorted according to the table name.

The real purpose of this DMV is to observe the usage of indexes in the long run. It might make sense to take a snapshot of that view or a snapshot of the result of the query and store it away every day to compare the changes over time. If you can identify that particular indexes were not used for months or during periods such as quarter-end reporting or fiscal-year reporting, you could eventually delete those indexes from the database.