IDERA® | SQL Server Whitepaper

TROUBLESHOOTING LOCKING WITH DMVS

BY ROBERT L DAVIS

An advanced guide for DBAs on the troubleshooting locking problems in SQL Server

INTRODUCTION

Database administrators (DBAs) are often called upon to investigate performance problems related to locking, but most investigations into locking rarely go beyond checking for blocks or investigating deadlocks. It is not very often that DBAs or SQL developers try to get a deeper understanding of locks that are occurring.

This paper will outline some techniques using Dynamic Management Views (DMVs) for understanding the full impact of the locks generated by a query. Then it will show you how you can combine the transaction locks DMV (sys.dm_tran_locks) with the execution DMVs (sys.dm_ exec_requests, sys.dm_exec_sql_text, and sys.dm_exec_ query_plan) to capture a snapshot that gives you the complete picture of locking activity hitting a specific table.

LOCKING, BLOCKING, AND DEADLOCKS

Before we can dive into an advanced topic like troubleshooting locking, we need to define some terms to differentiate concepts that may not be well understood by a lot of database administrators (DBAs) and SQL developers. Locking, blocking, and deadlocks are related, to a degree, but they are not interchangeable.

Locks: a logical mechanism for controlling access to an object.

- Locks control concurrent access to an object for non-conflicting operations (such as reading data rows)
- Locks prevent concurrent access to an object for conflicting operations (such as updating the same data rows)

Blocks: an incident where one query is waiting for a resource that another query is consuming.

- Most common scenario is caused by conflicting locks for objects
- May be caused by other conflicts, such as a wait for a memory grant

Deadlocks: a blocking scenario where two or more queries are blocking each other while waiting for locks held by another participating query

- Most common scenario is for two queries that each hold locks for which the other query is waiting
- May involve many queries that involve a circular locking chain that results in a deadlock
- While deadlocks are a product of blocking, the blocking may be caused by resource waits other than locks for objects, such as waits for memory grants

The simplest way to think of the relationship between locking, blocking, and deadlocks is that locking causes blocking, and blocking causes deadlocks. When troubleshooting deadlocks, it is necessary to look at the blocking that generated the deadlocks. In order to understand blocking (that resulted from locks) that causes deadlocks, you must look at the locks that were being held during the participating transactions.

LOCK CONVERSION

One of the hidden problems with locking is lock conversion. This is a problem with complex data modification (update, insert, or delete) queries. Query writers are often only concerned with the locks required to perform the data modification without any thought about the locks that must be taken to identify the rows that are being changed.

Lock conversion is the process of taking one type of lock and then converting it to more restrictive or more granular lock. For example, a query may start by taking shared locks on a number of rows in order to identify which rows to modify. At this point, it may release the shared locks on the rows it is not going to modify while converting the rows it does intend to modify exclusive locks.

Depending on the search criteria and the indexes available, the number of shared locks taken can be much greater than the number of exclusive locks required to modify the data. On the next page, let's take a look at the different types of locks (also called lock modes), all of which come into play with lock conversion:

Lock Code	Lock Mode	Description
S	Shared	A read-only lock. Multiple transactions can have shared locks on the same resource.
U	Update	An exclusive lock that can be taken instead of a shared lock when the resource may be updated later in the process.
X	Exclusive	Lock on a resource that is going to be modified to ensure that no other operation may obtain a lock on the resource until the transaction has committed.
I(X)	Intent	An intent lock indicates that the is the potential to escalate a lower level lock to a higher level such as page, partition, or table. Intent is paired with either shared, update, or exclusive locks to indicate the type of lock that may be taken. Intent locks also allow other queries to see what lower level locks exist without having to examine all lower level resources. Intent locks are compatible, but if they conversion to full locks may not be. For example, 2 queries can have IX locks on the table, but neither query can get an X lock on the table as long as another IX lock exists.
Sch-S	Schema Stability	Schema stability locks are taken to ensure the stability of the object schema during an operation that is dependent on the object not changing. Sch-S locks are compatible with all locks except Sch-M locks.
Sch-M	Schema Modification	Schema modification locks are taken when an operation is making changes to the schema and is an exclusive lock that is not compatible with other locks.
BU	Bulk Update	Bulk update locks are exclusive locks taken during bulk copy processes that specify the tablock (exclusive table lock) locking hint
Range(x)	Key Range	Range locks represent a lock on a range of rows in serializable isolation. Range locks place a lock on the consecutive rows of the index and include all rows between. This protects the range from any changes within the range in the index and ensures that the rows can be queried repeatedly and always receive the exact same set of rows.

For more details on lock modes see http://technet.microsoft.com/en-us/library/ms175519.aspx.
For more information in types of Range locks see http://technet.microsoft.com/en-us/library/ms191272.aspx.
These pages have not been updated for SQL Server 2012, SQL Server 2014, or SQL Server 2016, but they are still accurate for current versions.

One method I use to understand the locking that must occur for a data modification query is to think of it as a two part query. The first part operates similar to a select query and must take shared locks on all rows (or pages or partition or table, etc.) necessary to identify the rows for modification. The second part is the data modification which takes exclusive locks on only the rows (pages, partitions, table, etc.) that are actually being modified. The problem is, this is hard to visualize; however, there are some techniques to help us out here.

This exercise uses the sample database AdventureWorks2014 available for download from Codeplex (http://msftdbprodsamples.codeplex.com/). The query below seems like it would be pretty straightforward with regard to locking. If there are any rows that match the criteria, they will be locked. In my copy of the database, there are 4,472 rows with a quantity of 4, but because we are only deleting the top 100 rows, it should only need to locks those rows.

```
Delete Top(100)
From Production.TransactionHistory
Where Quantity = 4;
```

Measuring the locks of a short transaction like this can be tricky unless you are using a tool like Extended Events or a SQL Trace. However, we can measure the locks by using an explicit transaction and the DMV sys.dm_tran_locks. I can query the DMV for my current transaction by specifying where the column session_id equals the function @@SPID. The @@SPID function returns the session ID for the current session. While the transaction is not committed, I can get a view of the locks being held at the end of the transaction.

I group the DMV results on resource_type (the type of object being locked) and request_mode (the type of lock that was taken). This gives me a total of all current locks on all objects for the session.

```
SELECT resource_type,
    request_mode,
    LockCount = COUNT(*)
FROM sys.dm_tran_locks
WHERE request_session_id = @@SPID
Group By resource_type, request_mode;
```

This is wrapped inside of the explicit transaction, as I mentioned above. The query as a whole is as below:

```
Begin Tran;

Delete Top(100)
From Production.TransactionHistory
Where Quantity = 4;

SELECT resource_type,
    request_mode,
    LockCount = COUNT(*)
FROM sys.dm_tran_locks
WHERE request_session_id = @@SPID
Group By resource_type, request_mode;

Rollback;
```

The output of the DMV show the locks that are required to delete the top 100 records that match the criteria:

RESOURCE_TYPE	REQUEST_MODE	LOCKCOUNT
OBJECT	IX	1
PAGE	IX	72
DATABASE	S	1
KEY	X	300

The shared (S) lock on the database can be ignored for this scenario. All queries take a shared lock at the database level. There are intent-exclusive (IX) locks at the object (table) and page levels. IX locks serve two purposes.

1. **Intent locks establish a hierarchy of locking in case lock escalation needs to occur.**
   If multiple queries need to escalate locks at the same time, the timing of the intent locks determine which query is able to escalate its locks first.
2. **Simplifies checking for incompatible locks at different levels.**
   Intent locks at higher levels let other queries know that there is incompatible lock at a lower level. If a query wants to take an exclusive lock at a higher level like page or table, it doesn't have to inspect every row involved to see if any incompatible locks exist. It merely needs to check for the intent locks at its own level to know if incompatible locks exist at lower levels.

Lastly, we see 300 exclusive key locks. A key lock is a row lock in an index. Because we are only deleting the top 100, we may have expected to see only 100 row locks; however, there are a total of three indexes on this table (clustered primary key plus two nonclustered indexes) and locks are taken on all three indexes

We can join sys.dm_tran_locks.resource_associated_entity_id to sys.partitions.partition_id to include the index_ id in the output to get a clearer picture of the current locks.

```
SELECT L.resource_type,
      L.request_mode,
      P.index_id,
      LockCount = COUNT(*)
FROM sys.dm_tran_locks L
Left Join sys.partitions P On P.partition_id =
L.resource_associated_entity_id
WHERE L.request_session_id = @@SPID
Group By L.resource_type, L.request_mode, P.index_id;
```

RESOURCE_TYPE	REQUEST_MODE	INDEX_ID	LOCKCOUNT
DATABASE	S	NULL	1
KEY	X	1	100
KEY	X	2	100
KEY	X	3	100
OBJECT	IX	NULL	1
OBJECT	Sch-S	NULL	1
PAGE	IX	1	7
PAGE	IX	2	61
PAGE	IX	3	4

We can now clearly see that there are 100 exclusive row locks taken to delete the top 100 rows that match the specified criteria. But this only represents that second part of a data modification query that I talked about earlier. This does not account for queries that are taken for the first part, the SELECT-like phase of the query.

In order to make sure we are able to capture all locks that the query takes, we can use the holdlock query hint. Holdlock is synonymous with serializable and will hold locks for the duration of the transaction. With our previous check, we were only seeing some of the locks taken because some initial locks are dropped once they are no longer needed.

This makes our query look like the below query:

```
Delete Top(100)
From Production.TransactionHistory with(holdlock)
Where Quantity = 4;
```

RESOURCE_TYPE	REQUEST_MODE	INDEX_ID	LOCKCOUNT
DATABASE	S	NULL	1
KEY	RangeS-U	1	802
KEY	RangeX-X	1	100
KEY	X	2	100
KEY	X	3	100
OBJECT	IX	NULL	1
OBJECT	Sch-S	NULL	1
PAGE	IX	1	7
PAGE	IX	2	61
PAGE	IX	3	4

We see two key differences with these results. The 100 exclusive key locks on the index with an ID of 1 is now expressed as exclusive key range locks. Additionally, there are an additional 802 shared key range locks on the same index. The range locks represent the fact that before SQL Server can modify any rows, it first has to identify the rows that will be deleted. In this case, instead of taking individual locks on all 4,472 rows, it is able to take locks on ranges of rows and only take 902 range locks, 100 of which get converted to exclusive locks.

It is very easy to overlook the overhead of these additional locks, in part because they are not easy to capture. As I have shown here, you can easily use the DMV sys.dm_tran_locks to capture the total locks taken by the query.

Another reason it is easy to overlook these additional locks is because they are shared locks that get dropped when they are no longer needed. However, all locks have overhead. They consume memory, and increase the chances that the query will escalate to higher level locks. This technique is particularly useful when you have a query that is escalating to higher level locks even though it is modifying less rows than you think should trigger escalation. The query may have exceeded the lock count threshold during the first part of the query (identifying the rows to modify).

IDENTIFYING COMPETING LOCKS

Another common scenario that DBAs may have to contend with on very busy systems is when there are a lot of blocking activities and quite possibly a long blocking chain. When you look at the DMV sys.dm_exec_requests, you can identify queries that are suspended waiting for a lock and the query that is holding the lock that is incompatible with the lock that is waiting for a grant. What you cannot readily see is the chain of locks that may be ahead of your query that is waiting.

Imagine a scenario where you have a long blocking chain that is blocking many queries. You can see that there is massive blocking, but you cannot readily determine the extent to which your query is blocked.

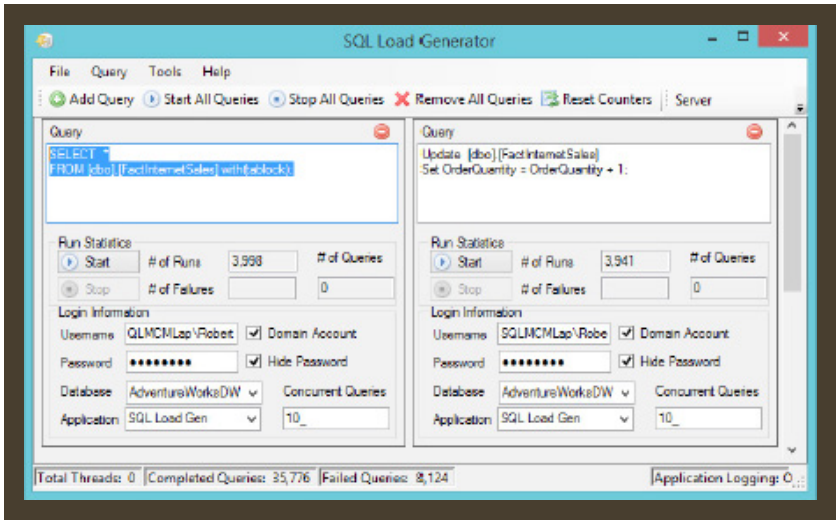
You could go through killing the blocking queries one at a time until the query you're concerned with is able to continue, but that does nothing to address the underlying problem. Before going to this extreme, we can grab a snapshot of what queries are hitting the table where the block is occurring so we can see what queries were conflicting.

To demonstrate this, let's start off with a couple of queries that will generate some conflicting locks on the dbo. FactInternetSales table in the sample database AdventureWorksDW2014. To generate an on-going workload, I will use the free tool, SQL Load Generator, which can be downloaded from the CodePlex site (http://sqlloadgenerator.codeplex.com/). I run each query with ten concurrent queries. The tool will run each query in ten sessions and keep cycling through them until I stop them.

The two queries I use to generate a load are a simple select with a table lock (tablock) and an update that updates every row:

```
SELECT       *
FROM [dbo].[FactInternetSales] with(tablock);

Update  [dbo].[FactInternetSales]
Set OrderQuantity = OrderQuantity + 1;
```



The diagnostic query captures everything you will need to diagnose the locking activity on the table in question after you have dealt with clearing out some of the contention, possibly by killing non-critical conflicting sessions. The query captures session ID, lock mode (shared, exclusive, intent exclusive, etc), lock state (wait or grant), general command, query status, current wait, last wait, wait resource, wait time, procedure name, SQL text of currently executing query, and the query plan.

```
Use AdventureWorksDW2014;

-- Define the table you want to search
Declare @TableName nvarchar(257);

Set @TableName = N'dbo.FactInternetSales';

-- Take a snapshot of locking activity on table
Select TableName = @TableName,
      SessionID = R.session_id,
      BlockingSessionID = R.blocking_session_id,
      LockMode = TL.request_mode,
      LockStatus = TL.request_status,
      Command = R.command,
      QueryStatus = R.status,
      CurrentWait = R.wait_type,
      LastWait = R.last_wait_type,
      WaitResource = R.wait_resource,
      WaitTime = R.wait_time,
      ProcedureName = IsNull(OBJECT_NAME(ST.objectid), '** adhoc **'),
      SQLText = SUBSTRING(ST.text, (R.statement_start_offset/2)+1,
            ((Case R.statement_end_offset
                  When -1 Then DATALENGTH(ST.text)
                  Else R.statement_end_offset
            End - R.statement_start_offset)/2) + 1),
      QueryPlan = Q.query_plan
From sys.dm_tran_locks TL
Inner Join sys.dm_exec_requests R On R.session_id = TL.request_session_id
Outer Apply sys.dm_exec_sql_text(R.sql_handle) As ST
Outer Apply sys.dm_exec_query_plan(R.plan_handle) As Q
Where TL.resource_type In ('page', 'key', 'RID', 'object')
And OBJECT_ID(@TableName) = Case
      When resource_type = 'object'
            Then resource_associated_entity_id
      When resource_type In ('page', 'key', 'RID')
            Then (Select top (1) object_id
                  From sys.partitions
                  Where partition_id = resource_associated_entity_id
                  And index_id in (0, 1))
      End
And TL.resource_database_id = DB_ID()
Order By R.session_id;
```

	TableName	SessionID	BlockingSessionID	LockMode	LockStatus	Command	QueryStatus	CurrentWait	LastWait	WaitResource
1	dbo.FactInternetSales	56	65	S	WAIT	SELECT	suspended	LCK_M_S	LCK_M_S	OBJECT: 9:6615773
2	dbo.FactInternetSales	57	65	S	WAIT	SELECT	suspended	LCK_M_S	LCK_M_S	OBJECT: 9:6615773
3	dbo.FactInternetSales	58	65	IX	WAIT	UPDATE	suspended	LCK_M_IX	LCK_M_IX	OBJECT: 9:6615773
4	dbo.FactInternetSales	59	0	X	GRANT	UPDATE	running	NULL	LCK_M_U	KEY: 9:0 (0000112a
5	dbo.FactInternetSales	59	0	X	GRANT	UPDATE	running	NULL	LCK_M_U	KEY: 9:0 (0000112a
6	dbo.FactInternetSales	59	0	X	GRANT	UPDATE	running	NULL	LCK_M_U	KEY: 9:0 (0000112a
7	dbo.FactInternetSales	59	0	X	GRANT	UPDATE	running	NULL	LCK_M_U	KEY: 9:0 (0000112a
8	dbo.FactInternetSales	59	0	X	GRANT	UPDATE	running	NULL	LCK_M_U	KEY: 9:0 (0000112a

The output of the query is quite large. The image above shows a snippet of the output for the activity I generated. It gives us a complete picture of the activity with locks on the table so we can determine where conflicts are occurring and which procedures and queries are causing blocking of other queries and procedures.

CONCLUSION

As database administrators and SQL developers, we can benefit from understanding the complete picture around locking generated by queries we write or deal with on systems we manage. Using the techniques described here for looking at the total locks generated by a query will help you realize the full impact of queries, especially those with broad search criteria.

The transaction locks and execution DMVs enable you to grab a snapshot of locking activity when lock contention occurs which allows you to investigate queries that cause or experience lock contention. Thus you can do more than simply kill blocking queries. It enables you to take a proactive approach to preventing future contention for the locks on objects.

DETAILS

Writer Robert L Davis

Applies To SQL Server 2008, SQL Server 2008 R2, SQL Server 2012, SQL Server 2014, SQL Server 2016

SQL DIAGNOSTIC MANAGER

ACHIEVE 24/7 SQL MONITORING

- Performance monitoring for physical and virtual environments
- Query plan monitoring to see the causes of blocks and deadlocks
- Integrated SQL Doctor expert recommendations
- Easy integration with Microsoft SCOM
- Predictive alerting with settings to avoid false alerts
- Web-based dashboard with at-a-glance views of top issues and alerts

Start for FREE