Understand and resolve SQL Server blocking problems

03/24/2021 • 31 minutes to read •   

In this article

Applies to: SQL Server (all supported versions), Azure SQL Managed Instance

Original KB number: 224453

Objective

The article describes blocking in SQL Server and demonstrates how to troubleshoot and resolve blocking.

In this article, the term connection refers to a single logged-on session of the database. Each connection appears as a session ID (SPID) or session_id in many DMVs. Each of these SPIDs is often referred to as a process, although it is not a separate process context in the usual sense. Rather, each SPID consists of the server resources and data structures necessary to service the requests of a single connection from a given client. A single client application may have one or more connections. From the perspective of SQL Server, there is no difference between multiple connections from a single client application on a single client computer and multiple connections from multiple client applications or multiple client computers; they are atomic. One connection can block another connection, regardless of the source client.

> **Note**
>
> **This article is focused on SQL Server instances, including Azure SQL Managed Instances.** For information specific to troubleshooting blocking in Azure SQL Database, see **Understand and resolve Azure SQL Database blocking problems**.

What is blocking

Blocking is an unavoidable and by-design characteristic of any relational database management system (RDBMS) with lock-based concurrency. As mentioned previously, in SQL Server, blocking occurs when one session holds a lock on a specific resource and a second SPID attempts to acquire a conflicting lock type on the same resource. Typically, the time frame for which the first SPID locks the resource is small. When the owning session releases the lock, the second connection is then free to acquire its own lock on the resource and continue processing. Blocking as described here is normal behavior and may happen many times throughout the course of a day with no noticeable effect on system performance.

The duration and transaction context of a query determine how long its locks are held and, thereby, their effect on other queries. If the query is not executed within a transaction (and no lock hints are used), the locks for SELECT statements will only be held on a resource at the time it is actually being read, not during the query. For INSERT, UPDATE, and DELETE statements, the locks are held during the query, both for data consistency and to allow the query to be rolled back if necessary.

For queries executed within a transaction, the duration for which the locks are held are determined by the type of query, the transaction isolation level, and whether lock hints are used in the query. For a description of locking, lock hints, and transaction isolation levels, see the following articles:

- Locking in the Database Engine
- Customizing Locking and Row Versioning
- Lock Modes
- Lock Compatibility
- Row Versioning-based Isolation Levels in the Database Engine
- Transactions

When locking and blocking persists to the point where there is a detrimental effect on system performance, it is due to one of the following reasons:

- A SPID holds locks on a set of resources for an extended period of time before releasing them. This type of blocking resolves itself over time but can cause

performance degradation.

- A SPID holds locks on a set of resources and never releases them. This type of blocking does not resolve itself and prevents access to the affected resources indefinitely.

In the first scenario, the situation can be very fluid as different SPIDs cause blocking on different resources over time, creating a moving target. These situations are difficult to troubleshoot using SQL Server Management Studio to narrow down the issue to individual queries. In contrast, the second situation results in a consistent state that can be easier to diagnose.

Applications and blocking

There may be a tendency to focus on server-side tuning and platform issues when facing a blocking problem. However, attention paid only to the database may not lead to a resolution, and can absorb time and energy better directed at examining the client application and the queries it submits. No matter what level of visibility the application exposes regarding the database calls being made, a blocking problem nonetheless frequently requires both the inspection of the exact SQL statements submitted by the application and the application's exact behavior regarding query cancellation, connection management, fetching all result rows, and so on. If the development tool does not allow explicit control over connection management, query cancellation, query timeout, result fetching, and so on, blocking problems may not be resolvable. This potential should be closely examined before selecting an application development tool for SQL Server, especially for performance sensitive OLTP environments.

Pay attention to database performance during the design and construction phase of the database and application. In particular, the resource consumption, isolation level, and transaction path length should be evaluated for each query. Each query and transaction should be as lightweight as possible. Good connection management discipline must be exercised, without it, the application may appear to have acceptable performance at low numbers of users, but the performance may degrade significantly as the number of users scales upward.

With proper application and query design, SQL Server is capable of supporting many thousands of simultaneous users on a single server, with little blocking.

Troubleshoot blocking

Regardless of which blocking situation we are in, the methodology for troubleshooting locking is the same. These logical separations are what will dictate the rest of the

composition of this article. The concept is to find the head blocker and identify what that query is doing and why it is blocking. Once the problematic query is identified (that is, what is holding locks for the prolonged period), the next step is to analyze and determine why the blocking happening. After we understand why, we can then make changes by redesigning the query and the transaction.

Steps in troubleshooting:

1. Identify the main blocking session (head blocker)
2. Find the query and transaction that is causing the blocking (what is holding locks for a prolonged period)
3. Analyze/understand why the prolonged blocking occurs
4. Resolve blocking issue by redesigning query and transaction

Now let's dive in to discuss how to pinpoint the main blocking session with an appropriate data capture.

Gather blocking information

To counteract the difficulty of troubleshooting blocking problems, a database administrator can use SQL scripts that constantly monitor the state of locking and blocking on SQL Server. To gather this data, there are two complimentary methods.

The first is to query dynamic management objects (DMOs) and store the results for comparison over time. Some objects referenced in this article are dynamic management views (DMVs) and some are dynamic management functions (DMFs).

The second is to use Extended Events(XEvents) or SQL Profiler Traces to capture what is executing. Since SQL Trace and SQL Server Profiler are deprecated, this troubleshooting guide will focus on XEvents.

Gather information from DMVs

Referencing DMVs to troubleshoot blocking has the goal of identifying the SPID (session ID) at the head of the blocking chain and the SQL Statement. Look for victim SPIDs that are being blocked. If any SPID is being blocked by another SPID, then investigate the SPID owning the resource (the blocking SPID). Is that owner SPID being blocked as well? You can walk the chain to find the head blocker then investigate why it is maintaining its lock.

To do this, you can use one of the following methods:

- In SQL Server Management Studio (SSMS) Object Explorer, right-click the top-level server object, **expand Reports**, expand **Standard Reports**, and then select **Activity – All Blocking Transactions**. This report shows current transactions at the head of a blocking chain. If you expand the transaction, the report will show the transactions that are blocked by the head transaction. This report will also show the **Blocking SQL Statement** and the **Blocked SQL Statement**.

- Open Activity Monitor in SSMS and refer to the Blocked By column. Find more information about Activity Monitor here.

More detailed query-based methods are also available using DMVs:

- The `sp_who` and `sp_who2` commands are older commands to show all current sessions. The DMV `sys.dm_exec_sessions` returns more data in a result set that is easier to query and filter. You will find `sys.dm_exec_sessions` at the core of other queries.

- If you already have a particular session identified, you can use `DBCC INPUTBUFFER(<session_id>)` to find the last statement that was submitted by a session. Similar results can be returned with the `sys.dm_exec_input_buffer` dynamic management function (DMF), in a result set that is easier to query and filter, providing the session_id and the request_id. For example, to return the most recent query submitted by session_id 66 and request_id 0:

SQL Copy

```
SELECT * FROM sys.dm_exec_input_buffer (66,0);
```

- Refer to the `sys.dm_exec_requests` and reference the `blocking_session_id` column. When `blocking_session_id` = 0, a session is not being blocked. While `sys.dm_exec_requests` lists only requests currently executing, any connection (active or not) will be listed in `sys.dm_exec_sessions`. Build on this common join between `sys.dm_exec_requests` and `sys.dm_exec_sessions` in the next query. Keep in mind to be returned by `sys.dm_exec_requests`, the query must be actively executing with SQL Server.

- Run this sample query to find the actively executing queries and their current SQL batch text or input buffer text, using the sys.dm_exec_sql_text or sys.dm_exec_input_buffer DMVs. If the data returned by the `text` column of `sys.dm_exec_sql_text` is NULL, the query is not currently executing. In that case, the `event_info` column of `sys.dm_exec_input_buffer` will contain the last command string passed to the SQL engine. This query can also be used to identify

sessions blocking other sessions, including a list of session_ids blocked per session_id.

SQL Copy

```
WITH cteBL (session_id, blocking_these) AS
(SELECT s.session_id, blocking_these = x.blocking_these FROM
sys.dm_exec_sessions s
CROSS APPLY    (SELECT isnull(convert(varchar(6), er.session_id),'') + ', '
                FROM sys.dm_exec_requests as er
                WHERE er.blocking_session_id = isnull(s.session_id ,0)
                AND er.blocking_session_id <> 0
                FOR XML PATH('') ) AS x (blocking_these)
)
SELECT s.session_id, blocked_by = r.blocking_session_id, bl.blocking_these
, batch_text = t.text, input_buffer = ib.event_info, *
FROM sys.dm_exec_sessions s
LEFT OUTER JOIN sys.dm_exec_requests r on r.session_id = s.session_id
INNER JOIN cteBL as bl on s.session_id = bl.session_id
OUTER APPLY sys.dm_exec_sql_text (r.sql_handle) t
OUTER APPLY sys.dm_exec_input_buffer(s.session_id, NULL) AS ib
WHERE blocking_these is not null or r.blocking_session_id > 0
ORDER BY len(bl.blocking_these) desc, r.blocking_session_id desc,
r.session_id;
```

- Run this more elaborate sample query, provided by Microsoft Support, to identify the head of a multiple session blocking chain, including the query text of the sessions involved in a blocking chain.

SQL Copy

```
WITH cteHead (
session_id,request_id,wait_type,wait_resource,last_wait_type,is_user_process
,request_cpu_time
,request_logical_reads,request_reads,request_writes,wait_time,blocking_sessi
on_id,memory_usage
,session_cpu_time,session_reads,session_writes,session_logical_reads
,percent_complete,est_completion_time,request_start_time,request_status,comm
and
,plan_handle,sql_handle,statement_start_offset,statement_end_offset,most_rec
ent_sql_handle
,session_status,group_id,query_hash,query_plan_hash)
AS ( SELECT sess.session_id, req.request_id, LEFT (ISNULL (req.wait_type,
''), 50) AS 'wait_type'
    , LEFT (ISNULL (req.wait_resource, ''), 40) AS 'wait_resource', LEFT
(req.last_wait_type, 50) AS 'last_wait_type'
    , sess.is_user_process, req.cpu_time AS 'request_cpu_time',
req.logical_reads AS 'request_logical_reads'
    , req.reads AS 'request_reads', req.writes AS 'request_writes',
req.wait_time, req.blocking_session_id,sess.memory_usage
    , sess.cpu_time AS 'session_cpu_time', sess.reads AS 'session_reads',
sess.writes AS 'session_writes', sess.logical_reads AS
'session_logical_reads'
    , CONVERT (decimal(5,2), req.percent_complete) AS 'percent_complete',
```

```
req.estimated_completion_time AS 'est_completion_time'
    , req.start_time AS 'request_start_time', LEFT (req.status, 15) AS
'request_status', req.command
    , req.plan_handle, req.[sql_handle], req.statement_start_offset,
req.statement_end_offset, conn.most_recent_sql_handle
    , LEFT (sess.status, 15) AS 'session_status', sess.group_id,
req.query_hash, req.query_plan_hash
    FROM sys.dm_exec_sessions AS sess
    LEFT OUTER JOIN sys.dm_exec_requests AS req ON sess.session_id =
req.session_id
    LEFT OUTER JOIN sys.dm_exec_connections AS conn on conn.session_id =
sess.session_id
    )
, cteBlockingHierarchy (head_blocker_session_id, session_id,
blocking_session_id, wait_type, wait_duration_ms,
wait_resource, statement_start_offset, statement_end_offset, plan_handle,
sql_handle, most_recent_sql_handle, [Level])
AS ( SELECT head.session_id AS head_blocker_session_id, head.session_id AS
session_id, head.blocking_session_id
    , head.wait_type, head.wait_time, head.wait_resource,
head.statement_start_offset, head.statement_end_offset
    , head.plan_handle, head.sql_handle, head.most_recent_sql_handle, 0 AS
[Level]
    FROM cteHead AS head
    WHERE (head.blocking_session_id IS NULL OR head.blocking_session_id = 0)
    AND head.session_id IN (SELECT DISTINCT blocking_session_id FROM cteHead
WHERE blocking_session_id != 0)
    UNION ALL
    SELECT h.head_blocker_session_id, blocked.session_id,
blocked.blocking_session_id, blocked.wait_type,
    blocked.wait_time, blocked.wait_resource, h.statement_start_offset,
h.statement_end_offset,
    h.plan_handle, h.sql_handle, h.most_recent_sql_handle, [Level] + 1
    FROM cteHead AS blocked
    INNER JOIN cteBlockingHierarchy AS h ON h.session_id =
blocked.blocking_session_id and h.session_id!=blocked.session_id --avoid in-
finite recursion for latch type of blocking
    WHERE h.wait_type COLLATE Latin1_General_BIN NOT IN ('EXCHANGE', 'CX-
PACKET') or h.wait_type is null
    )
SELECT bh.*, txt.text AS blocker_query_or_most_recent_query
FROM cteBlockingHierarchy AS bh
OUTER APPLY sys.dm_exec_sql_text (ISNULL ([sql_handle],
most_recent_sql_handle)) AS txt;
```

- To catch long-running or uncommitted transactions, use another set of DMVs for viewing current open transactions, including sys.dm_tran_database_transactions, sys.dm_tran_session_transactions, sys.dm_exec_connections, and `sys.dm_exec_sql_text`. There are several DMVs associated with tracking transactions, see more DMVs on transactions here.

SQL Copy

```
SELECT [s_tst].[session_id],
[database_name] = DB_NAME (s_tdt.database_id),
[s_tdt].[database_transaction_begin_time],
[sql_text] = [s_est].[text]
FROM sys.dm_tran_database_transactions [s_tdt]
INNER JOIN sys.dm_tran_session_transactions [s_tst] ON [s_tst].
[transaction_id] = [s_tdt].[transaction_id]
INNER JOIN sys.dm_exec_connections [s_ec] ON [s_ec].[session_id] = [s_tst].
[session_id]
CROSS APPLY sys.dm_exec_sql_text ([s_ec].[most_recent_sql_handle]) AS
[s_est];
```

- Reference sys.dm_os_waiting_tasks that is at the thread/task layer of SQL Server. This returns information about what SQL wait type the request is currently experiencing. Like `sys.dm_exec_requests`, only active requests are returned by `sys.dm_os_waiting_tasks`.

> **Note**
>
> For much more on wait types including aggregated wait stats over time, see the DMV **sys.dm_db_wait_stats**.

- Use the sys.dm_tran_locks DMV for more granular information on what locks have been placed by queries. This DMV can return large amounts of data on a production SQL Server instance, and is useful for diagnosing what locks are currently held.

Due to the INNER JOIN on `sys.dm_os_waiting_tasks`, the following query restricts the output from `sys.dm_tran_locks` only to currently blocked requests, their wait status, and their locks:

SQL Copy

```
SELECT table_name = schema_name(o.schema_id) + '.' + o.name
, wt.wait_duration_ms, wt.wait_type, wt.blocking_session_id,
wt.resource_description
, tm.resource_type, tm.request_status, tm.request_mode,
tm.request_session_id
FROM sys.dm_tran_locks AS tm
INNER JOIN sys.dm_os_waiting_tasks as wt ON tm.lock_owner_address =
wt.resource_address
LEFT OUTER JOIN sys.partitions AS p on p.hobt_id =
tm.resource_associated_entity_id
LEFT OUTER JOIN sys.objects o on o.object_id = p.object_id or
tm.resource_associated_entity_id = o.object_id
WHERE resource_database_id = DB_ID()
AND object_name(p.object_id) = '<table_name>';
```

With DMVs, storing the query results over time will provide data points that will allow you to review blocking over a specified time interval to identify persisted blocking or trends. The go-to tool for CSS to troubleshoot such issues is using the PSSDiag data collector. This tool uses the "SQL Server Perf Stats" to collect resultsets from DMVs referenced above, over time. As this tool is constantly evolving, review the latest public version of DiagManager on GitHub .

Gather information from extended events

In addition to the above information, it is often necessary to capture a trace of the activities on the server to thoroughly investigate a blocking problem in SQL Server. For example, if a session executes multiple statements within a transaction, only the last statement that was submitted will be represented. However, one of the earlier statements may be the reason locks are still being held. A trace will enable you to see all the commands executed by a session within the current transaction.

There are two ways to capture traces in SQL Server; Extended Events (XEvents) and Profiler Traces. However, SQL traces using the SQL Server Profiler are deprecated. XEvents are the newer, superior tracing platform that allows more versatility and less impact to the observed system, and its interface is integrated into SSMS.

There are pre-made Extended Event sessions ready to start in SSMS, listed in Object Explorer under the menu for XEvent Profiler. For more information, see Xevent Profiler. You can also create your own custom Extended Event sessions in SSMS, see Extended Events New Session Wizard. For troubleshooting blocking issues, we typically will capture:

- Category Errors:
  - Attention
  - Blocked_process_report**
  - Error_reported (Channel Admin)
  - Exchange_spill
  - Execution_warning

**To configure the threshold and frequency at which blocked process reports are generated, use the sp_configure command to configure the blocked process threshold option, which can be set in seconds. By default, no blocked process reports are produced.

- Category Warnings:
- Hash_warning
- Missing_column_statistics

- Missing_join_predicate
- Sort_warning
- Category Execution:
  - Rpc_completed
  - Rpc_starting
  - Sql_batch_completed
  - Sql_batch_starting
- Category Lock
  - Lock_deadlock
- Category Session
  - Existing_connection
  - Login
  - Logout

Identify and resolve common blocking scenarios

By examining the above information, you can determine the cause of most blocking problems. The rest of this article is a discussion of how to use this information to identify and resolve some common blocking scenarios. This discussion assumes you have used the blocking scripts (referenced earlier) to capture information on the blocking SPIDs and have captured application activity using an XEvent session.

Analyze blocking data

- Examine the output of the DMVs `sys.dm_exec_requests` and `sys.dm_exec_sessions` to determine the heads of the blocking chains, using `blocking_these` and `session_id`. This will most clearly identify which requests are blocked and which are blocking. Look further into the sessions that are blocked and blocking. Is there a common or root to the blocking chain? They likely share a common table, and one or more of the sessions involved in a blocking chain is performing a write operation.
- Examine the output of the DMVs `sys.dm_exec_requests` and `sys.dm_exec_sessions` for information on the SPIDs at the head of the blocking chain. Look for the following columns:
  - `sys.dm_exec_requests.status`

    This column shows the status of a particular request. Typically, a sleeping status indicates that the SPID has completed execution and is waiting for the

application to submit another query or batch. A runnable or running status indicates that the SPID is currently processing a query. The following table gives brief explanations of the various status values.

Status	Meaning
Background	The SPID is running a background task, such as deadlock detection, log writer, or checkpoint.
Sleeping	The SPID is not currently executing. This usually indicates that the SPID is awaiting a command from the application.
Running	The SPID is currently running on a scheduler.
Runnable	The SPID is in the runnable queue of a scheduler and waiting to get scheduler time.
Suspended	The SPID is waiting for a resource, such as a lock or a latch.

- `sys.dm_exec_sessions.open_transaction_count`
  This column tells you the number of open transactions in this session. If this value is greater than 0, the SPID is within an open transaction and may be holding locks acquired by any statement within the transaction.

- `sys.dm_exec_requests.open_transaction_count`
  Similarly, this column tells you the number of open transactions in this request. If this value is greater than 0, the SPID is within an open transaction and may be holding locks acquired by any statement within the transaction.

- `sys.dm_exec_requests.wait_type`, `wait_time`, and `last_wait_type`
  If the `sys.dm_exec_requests.wait_type` is NULL, the request is not currently waiting for anything and the `last_wait_type` value indicates the last `wait_type` that the request encountered. For more information about `sys.dm_os_wait_stats` and a description of the most common wait types, see sys.dm_os_wait_stats. The `wait_time` value can be used to determine if the request is making progress. When a query against the `sys.dm_exec_requests` table returns a value in the `wait_time` column that is less than the `wait_time` value from a previous query of `sys.dm_exec_requests`, this indicates that the prior lock was acquired and released and is now waiting on a new lock (assuming non-zero `wait_time`). This can be verified by comparing the `wait_resource` between `sys.dm_exec_requests` output, which displays the resource for which the request is waiting.

- `sys.dm_exec_requests.wait_resource` This column indicates the resource that a blocked request is waiting on. The following table lists common `wait_resource` formats and their meaning:

Resource	Format	Example	Explanat
Table	DatabaseID:ObjectID:IndexID	TAB: 5:261575970:1	In this ca ID 5 is th sample d `object_i` 2615759 titles tab the cluste
Page	DatabaseID:FileID:PageID	PAGE: 5:1:104	In this ca ID 5 is pu is the pri file, and page bel the titles identify t the page use the d manager function sys.dm_d passing i Database PageId fr `wait_res`
Key	DatabaseID:Hobt_id (Hash value for index key)	KEY: 5:72057594044284928 (3300a4f361aa)	In this ca ID 5 is Pu 7205759 correspo index_id object_id (titles tab `sys.part` catalog v associate `hobt_id` particula and `obje` is no way the index a specific

Resource	Format	Example	Explanat
Row	DatabaseID:FileID:PageID:Slot(row)	RID: 5:1:104:3	In this ca ID 5 is pu is the pri file, page page bel the titles slot 3 inc row's pos page.
Compile	DatabaseID:FileID:PageID:Slot(row)	RID: 5:1:104:3	In this ca ID 5 is pu is the pri file, page page bel the titles slot 3 inc row's pos page.

- `sys.dm_tran_active_transactions` The sys.dm_tran_active_transactions DMV contains data about open transactions that can be joined to other DMVs for a complete picture of transactions awaiting commit or rollback. Use the following query to return information on open transactions, joined to other DMVs including sys.dm_tran_session_transactions. Consider a transaction's current state, `transaction_begin_time`, and other situational data to evaluate whether it could be a source of blocking.

SQL Copy

```
SELECT tst.session_id, [database_name] = db_name(s.database_id)
, tat.transaction_begin_time
, transaction_duration_s = datediff(s, tat.transaction_begin_time, sys-
datetime())
, transaction_type = CASE tat.transaction_type  WHEN 1 THEN 'Read/write
transaction'
                                                WHEN 2 THEN 'Read-only
transaction'
                                                WHEN 3 THEN 'System
transaction'
                                                WHEN 4 THEN 'Distrib-
uted transaction' END
, input_buffer = ib.event_info, tat.transaction_uow
, transaction_state  = CASE tat.transaction_state
            WHEN 0 THEN 'The transaction has not been completely ini-
tialized yet.'
            WHEN 1 THEN 'The transaction has been initialized but has
```

```
not started.'
            WHEN 2 THEN 'The transaction is active - has not been com-
mitted or rolled back.'
            WHEN 3 THEN 'The transaction has ended. This is used for
read-only transactions.'
            WHEN 4 THEN 'The commit process has been initiated on the
distributed transaction.'
            WHEN 5 THEN 'The transaction is in a prepared state and
waiting resolution.'
            WHEN 6 THEN 'The transaction has been committed.'
            WHEN 7 THEN 'The transaction is being rolled back.'
            WHEN 8 THEN 'The transaction has been rolled back.' END
, transaction_name = tat.name, request_status = r.status
, tst.is_user_transaction, tst.is_local
, session_open_transaction_count = tst.open_transaction_count
, s.host_name, s.program_name, s.client_interface_name, s.login_name,
s.is_user_process
FROM sys.dm_tran_active_transactions tat
INNER JOIN sys.dm_tran_session_transactions tst  on tat.transaction_id
= tst.transaction_id
INNER JOIN Sys.dm_exec_sessions s on s.session_id = tst.session_id
LEFT OUTER JOIN sys.dm_exec_requests r on r.session_id = s.session_id
CROSS APPLY sys.dm_exec_input_buffer(s.session_id, null) AS ib;
```

- Other columns

  The remaining columns in sys.dm_exec_sessions and sys.dm_exec_request can provide insight into the root of a problem as well. Their usefulness varies depending on the circumstances of the problem. For example, you can determine if the problem happens only from certain clients (`hostname`), on certain network libraries (`client_interface_name`), when the last batch submitted by a SPID was `last_request_start_time` in `sys.dm_exec_sessions`, how long a request had been running using `start_time` in `sys.dm_exec_requests`, and so on.

Common blocking scenarios

The table below maps common symptoms to their probable causes.

The `wait_type`, `open_transaction_count`, and `status` columns refer to information returned by sys.dm_exec_request, other columns may be returned by sys.dm_exec_sessions. The "Resolves?" column indicates whether or not the blocking will resolve on its own, or whether the session should be killed via the `KILL` command. For more information, see KILL (Transact-SQL).

Scenario	Wait_type	Open_Tran	Status	Resolves?	Other Symptoms
1	NOT NULL	>= 0	runnable	Yes, when query	In `sys.dm_exec_sess reads, cpu_time, ar`

Scenario	Wait_type	Open_Tran	Status	Resolves?	Other symptoms
				finishes.	`memory_usage` colun will increase over ti Duration for the qu will be high when completed.
2	NULL	>0	sleeping	No, but SPID can be killed.	An attention signal be seen in the Exter Event session for th SPID, indicating a q timeout or cancel h occurred.
3	NULL	>= 0	runnable	No. Will not resolve until client fetches all rows or closes connection. SPID can be killed, but it may take up to 30 seconds.	If open_transaction_c = 0, and the SPID h locks while the transaction isolatior level is default (REA COMMMITTED), thi likely cause.
4	Varies	>= 0	runnable	No. Will not resolve until client cancels queries or closes connections. SPIDs can be killed, but may take up to 30 seconds.	The `hostname` colun `sys.dm_exec_sessio` for the SPID at the l of a blocking chain be the same as one the SPID it is blocki
5	NULL	>0	rollback	Yes.	An attention signal be seen in the Exter Events session for t SPID, indicating a q timeout or cancel h occurred, or simply

Scenario	Wait_type	Open_Tran	Status	Resolves?	Other Symptoms
					rollback statement been issued.
6	NULL	>0	sleeping	Eventually. When Windows NT determines the session is no longer active, the connection will be broken.	The `last_request_start` value in `sys.dm_exec_sessio` much earlier than tl current time.

Detailed blocking scenarios

1. Blocking caused by a normally running query with a long execution time

   Resolution: The solution to this type of blocking problem is to look for ways to optimize the query. Actually, this class of blocking problem may just be a performance problem, and require you to pursue it as such. For information on troubleshooting a specific slow-running query, see How to troubleshoot slow-running queries on SQL Server. For more information, see Monitor and Tune for Performance.

   Reports built-in to SSMS from the Query Store (introduced in SQL Server 2016) are also a highly recommended and valuable tool for identifying the most costly queries, suboptimal execution plans.

   If you have a long-running query that is blocking other users and cannot be optimized, consider moving it from an OLTP environment to a dedicated reporting system, or use AlwaysOn availability groups to synchronize a read-only replica of the database.

2. Blocking caused by a sleeping SPID that has an uncommitted transaction

This type of blocking can often be identified by a SPID that is sleeping or awaiting a command, yet whose transaction nesting level (`@@TRANCOUNT`, `open_transaction_count` from `sys.dm_exec_requests`) is greater than zero. This can occur if the application experiences a query timeout, or issues a cancel without also issuing the required number of ROLLBACK and/or COMMIT statements. When a SPID receives a query timeout or a cancel, it will terminate the current query and batch, but does not automatically roll back or commit the transaction. The application is responsible for this, as SQL Server cannot assume that an entire transaction must be rolled back due to a single query being canceled. The query timeout or cancel will appear as an ATTENTION signal event for the SPID in the Extended Event session.

To demonstrate an uncommitted explicit transaction, issue the following query:

SQL Copy

```
CREATE TABLE #test (col1 INT);
INSERT INTO #test SELECT 1;
BEGIN TRAN
UPDATE #test SET col1 = 2 where col1 = 1;
```

Then, execute this query in the same window:

SQL Copy

```
SELECT @@TRANCOUNT;
ROLLBACK TRAN
DROP TABLE #test;
```

The output of the second query indicates that the transaction nesting level is one. All the locks acquired in the transaction are still be held until the transaction was committed or rolled back. If applications explicitly open and commit transactions, a communication or other error could leave the session and its transaction in an open state.

Use the script earlier in this article based on `sys.dm_tran_active_transactions` to identify currently uncommitted transactions across the instance.

Resolutions:

- Additionally, this class of blocking problem may also be a performance problem, and require you to pursue it as such. If the query execution time can be diminished, the query timeout or cancel would not occur. It is important that the application is able to handle the timeout or cancel

scenarios should they arise, but you may also benefit from examining the performance of the query.

- Applications must properly manage transaction nesting levels, or they may cause a blocking problem following the cancellation of the query in this manner. Consider the following:
  - In the error handler of the client application, execute `IF @@TRANCOUNT > 0 ROLLBACK TRAN` following any error, even if the client application does not believe a transaction is open. Checking for open transactions is required, because a stored procedure called during the batch could have started a transaction without the client application's knowledge. Certain conditions, such as canceling the query, prevent the procedure from executing past the current statement, so even if the procedure has logic to check `IF @@ERROR <> 0` and abort the transaction, this rollback code will not be executed in such cases.
  - If connection pooling is being used in an application that opens the connection and runs a small number of queries before releasing the connection back to the pool, such as a Web-based application, temporarily disabling connection pooling may help alleviate the problem until the client application is modified to handle the errors appropriately. By disabling connection pooling, releasing the connection will cause a physical disconnect of the SQL Server connection, resulting in the server rolling back any open transactions.
  - Use `SET XACT_ABORT ON` for the connection, or in any stored procedures that begin transactions and are not cleaning up following an error. In the event of a run-time error, this setting will abort any open transactions and return control to the client. For more information, review SET XACT_ABORT (Transact-SQL).

> **Note**
>
> The connection is not reset until it is reused from the connection pool, so it is possible that a user could open a transaction and then release the connection to the connection pool, but it might not be reused for several seconds, during which time the transaction would remain open. If the connection is not reused, the transaction will be aborted when the connection times out and is removed from the connection pool. Thus, it is optimal for the client application to abort transactions in their error handler or use `SET XACT_ABORT ON` to avoid this potential delay.

> **⊗ Caution**
>
> Following `SET XACT_ABORT ON`, T-SQL statements following a statement that causes an error will not be executed. This could affect the intended flow of existing code.

3. Blocking caused by a SPID whose corresponding client application did not fetch all result rows to completion

   After sending a query to the server, all applications must immediately fetch all result rows to completion. If an application does not fetch all result rows, locks can be left on the tables, blocking other users. If you are using an application that transparently submits SQL statements to the server, the application must fetch all result rows. If it does not (and if it cannot be configured to do so), you may be unable to resolve the blocking problem. To avoid the problem, you can restrict poorly behaved applications to a reporting or a decision-support database, separate from the main OLTP database.

   Resolution:

   The application must be rewritten to fetch all rows of the result to completion. This does not rule out the use of OFFSET and FETCH in the ORDER BY clause of a query to perform server-side paging.

4. Blocking caused by a distributed client/server deadlock

   Unlike a conventional deadlock, a distributed deadlock is not detectable using the RDBMS lock manager. This is because only one of the resources involved in the deadlock is a SQL Server lock. The other side of the deadlock is at the client application level, over which SQL Server has no control. The following are two examples of how this can happen, and possible ways the application can avoid it.

   A. Client/Server Distributed Deadlock with a Single Client Thread

   If the client has multiple open connections, and a single thread of execution, the following distributed deadlock may occur. For brevity, the term `dbproc` used here refers to the client connection structure.

Console | Copy

```
SPID1------blocked on lock------->SPID2
  /\ (waiting to write results
  | back to client)
  | |
  | | Server side
```

```
| ================================|================================
| <-- single thread --> | Client side
| \/
dbproc1 <------------------- dbproc2
(waiting to fetch (effectively blocked on dbproc1, awaiting
next row) single thread of execution to run)
```

In the case shown above, a single client application thread has two open connections. It asynchronously submits a SQL operation on dbproc1. This means it does not wait on the call to return before proceeding. The application then submits another SQL operation on dbproc2, and awaits the results to start processing the returned data. When data starts coming back (whichever dbproc first responds--assume this is dbproc1), it processes to completion all the data returned on that dbproc. It fetches results from dbproc1 until SPID1 gets blocked on a lock held by SPID2 (because the two queries are running asynchronously on the server). At this point, dbproc1 will wait indefinitely for more data. SPID2 is not blocked on a lock, but tries to send data to its client, dbproc2. However, dbproc2 is effectively blocked on dbproc1 at the application layer as the single thread of execution for the application is in use by dbproc1. This results in a deadlock that SQL Server cannot detect or resolve because only one of the resources involved is a SQL Server resource.

B. Client/Server Distributed Deadlock with a Thread per Connection

Even if a separate thread exists for each connection on the client, a variation of this distributed deadlock may still occur as shown by the following.

Console Copy

```
SPID1------blocked on lock-------->SPID2
  /\ (waiting on net write) Server side
  | |
  | |
  | INSERT |SELECT
  | ================================|================================
  | <-- thread per dbproc --> | Client side
  | \/
  dbproc1 <-----data row------- dbproc2
  (waiting on (blocked on dbproc1, waiting for it
  insert) to read the row from its buffer)
```

This case is similar to Example A, except dbproc2 and SPID2 are running a `SELECT` statement with the intention of performing row-at-a-time processing and handing each row through a buffer to dbproc1 for an `INSERT`, `UPDATE`, or `DELETE` statement on the same table. Eventually, SPID1 (performing the `INSERT`, `UPDATE`, or `DELETE`) becomes blocked on a lock held by SPID2 (performing the `SELECT`). SPID2 writes a

result row to the client dbproc2. Dbproc2 then tries to pass the row in a buffer to dbproc1, but finds dbproc1 is busy (it is blocked waiting on SPID1 to finish the current `INSERT`, which is blocked on SPID2). At this point, dbproc2 is blocked at the application layer by dbproc1 whose SPID (SPID1) is blocked at the database level by SPID2. Again, this results in a deadlock that SQL Server cannot detect or resolve because only one of the resources involved is a SQL Server resource.

Both examples A and B are fundamental issues that application developers must be aware of. They must code applications to handle these cases appropriately.

Resolutions:

When a query timeout has been provided, if the distributed deadlock occurs, it will be broken when timeout happens. Reference your connection provider documentation for more information on using a query timeout.

5. Blocking caused by a session in a rollback state

   A data modification query that is KILLed, or canceled outside of a user-defined transaction, will be rolled back. This can also occur as a side effect of the client network session disconnecting, or when a request is selected as the deadlock victim. This can often be identified by observing the output of `sys.dm_exec_requests`, which may indicate the ROLLBACK `command`, and the `percent_complete` column may show progress.

   A data modification query that is KILLed, or canceled outside of a user-defined transaction, will be rolled back. This can also occur as a side effect of the client computer restarting and its network session disconnecting. Likewise, a query selected as the deadlock victim will be rolled back. A data modification query often cannot be rolled back any faster than the changes were initially applied. For example, if a `DELETE`, `INSERT`, or `UPDATE` statement had been running for an hour, it could take at least an hour to roll back. This is expected behavior, because the changes made must be rolled back, or transactional and physical integrity in the database would be compromised. Because this must happen, SQL Server marks the SPID in a golden or rollback state (which means it cannot be KILLed or selected as a deadlock victim). This can often be identified by observing the output of `sp_who`, which may indicate the ROLLBACK command. The `status` column of `sys.dm_exec_sessions` will indicate a ROLLBACK status.

   > **Note**
   >
   > Lengthy rollbacks are rare when the **Accelerated Database Recovery feature** is enabled. This feature was introduced in SQL Server 2019.

Resolution:

You must wait for the session to finish rolling back the changes that were made.

If the instance is shut down in the middle of this operation, the database will be in recovery mode upon restarting, and it will be inaccessible until all open transactions are processed. Startup recovery takes essentially the same amount of time per transaction as run-time recovery, and the database is inaccessible during this period. Thus, forcing the server down to fix a SPID in a rollback state will often be counterproductive. In SQL Server 2019 with Accelerated Database Recovery enabled, this should not occur.

To avoid this situation, do not perform large batch write operations or index creation or maintenance operations during busy hours on OLTP systems. If possible, perform such operations during periods of low activity.

6. Blocking caused by an orphaned connection

   This is a common problem scenario. If the client application stops or the client workstation is restarted, or there is a batch-aborting error, the network session to the server may not be immediately canceled under some conditions. This can occur if the application does not rollback the transaction in the application's CATCH or FINALLY blocks.

   In this scenario, while the execution of a SQL batch has been canceled, the SQL connection and transaction are left open by the application. From the SQL Server instance's perspective, the client still appears to be present, and any locks acquired may still be retained.

   Resolution:

   The best way to prevent this condition is by improving application error handling, especially for unexpected terminations. Consider also the use of `SET XACT_ABORT ON` for the connection, or in any stored procedures that begin transactions and are not cleaning up following an error. In the event of a run-time error, this setting will abort any open transactions and return control to the client. For more information, review SET XACT_ABORT (Transact-SQL).

   To resolve an orphaned connection of a client application that has disconnected without appropriately cleaning up its resources, you can terminate the SPID by using the `KILL` command. For reference, see KILL (Transact-SQL).

   The `KILL` command takes the SPID value as input. For example, to kill SPID 9, issue the following command:

SQL Copy

```
KILL 99
```

> **Note**
>
> The `KILL` command may take up to 30 seconds to complete, due to the interval between checks for the `KILL` command.

See also

- Monitoring performance by using the Query Store
- Transaction Locking and Row Versioning Guide
- SET TRANSACTION ISOLATION LEVEL
- Quickstart: Extended events in SQL Server

Is this page helpful?

Yes No