Blocking and deadlocks

Last updated by | Peter Hewitt | Oct 19, 2022 at 12:02 AM PDT

Contents

Common solution

This is the common solution article related to blocking and deadlock issues that's displayed to customers in the Azure portal when creating a support ticket.

## Resolve blocking and deadlock issues with Azure SQL Database

Blocking and deadlocks are related. Their symptoms are overlapping to a degree, but they are still separate issues. Use the following description to distinguish between these issues:

- **Blocking** occurs in a database when one session holds a lock on a specific resource and a second session attempts to acquire a conflicting lock type on the same resource. When the owning session releases the lock, the second connection is then free to acquire its own lock on the resource and can continue processing.
- A **deadlock** occurs when two or more sessions are waiting on the same resource, and each process is waiting on the other process to complete before moving forward. These sessions are blocking each other, but they can't resolve the deadlock situation on their own. The database engine has to kill one session and will report error message 1205, "Your transaction (process ID #52) was deadlocked on lock resources", to the deadlock victim.

Review the sections below for each scenario to get troubleshooting steps and details that will help resolve your issues.

Section: Blocking - Overview

Blocking is an unavoidable characteristic of any relational database management system with lock-based concurrency. Blocked queries can cause poor performance, including slow-running or long-running queries that contribute to excessive resource consumption. While the concepts of blocking are the same for SQL Server and Azure SQL Database, the default isolation level is different. For Azure SQL Databases, READ_COMMITTED_SNAPSHOT ⧉ is enabled by default, thus avoiding a wide scope of causes for blocking by default.

In addition to the following sections, also review and work through Understand and resolve Azure SQL Database blocking problems ⧉. This complete troubleshooting article describes blocking in full detail and demonstrates step-by-step instructions on how to troubleshoot and resolve the most common blocking scenarios.

Section: Blocking - Diagnostics

We're running checks on your database to identify causes of blocking. This will take approximately 30 seconds or less to complete.
<Diagnostics results displayed to customer>

Section: Blocking - Collect blocking details

The first troubleshooting task is to identify the main blocking session (head blocker). The second task is to then find the query and transaction that is causing the blocking and holding the blocking locks.

The following SQL queries will help you with these tasks. Run the queries in the affected database where you see the blocking; you can also copy the queries into the same script file and run them in one batch to capture a snapshot of the blocking situation.

Step 1

While the blocking issue is still occurring, run the following query to capture details about the head blocker and the blocked sessions.

```sql
SELECT
    [HeadBlocker] =
        CASE
            -- session has an active request, is blocked, but is blocking others
            -- or session is idle but has an open transaction and is blocking others
            WHEN r2.session_id IS NOT NULL AND (r.blocking_session_id = 0 OR r.session_id IS NULL) THEN '1'
            -- session is either not blocking someone, or is blocking someone but is blocked by another party
            ELSE ''
        END,
    [SessionID] = s.session_id,
    [Login] = s.login_name,
    [Database] = db_name(p.dbid),
    [BlockedBy] = w.blocking_session_id,
    [OpenTransactions] = r.open_transaction_count,
    [Status] = s.status,
    [WaitType] = w.wait_type,
    [WaitTime_ms] = w.wait_duration_ms,
    [WaitResource] = r.wait_resource,
    [WaitResourceDesc] = w.resource_description,
    [Command] = r.command,
    [Application] = s.program_name,
    [TotalCPU_ms] = s.cpu_time,
    [TotalPhysicalIO_MB] = (s.reads + s.writes) * 8 / 1024,
    [MemoryUse_KB] = s.memory_usage * 8192 / 1024,
    [LoginTime] = s.login_time,
    [LastRequestStartTime] = s.last_request_start_time,
    [HostName] = s.host_name,
    [QueryHash] = r.query_hash,
    [BlockerQuery_or_MostRecentQuery] = txt.text
FROM sys.dm_exec_sessions s
LEFT OUTER JOIN sys.dm_exec_connections c ON (s.session_id = c.session_id)
LEFT OUTER JOIN sys.dm_exec_requests r ON (s.session_id = r.session_id)
LEFT OUTER JOIN sys.dm_os_tasks t ON (r.session_id = t.session_id AND r.request_id = t.request_id)
LEFT OUTER JOIN
(
    SELECT *, ROW_NUMBER() OVER (PARTITION BY waiting_task_address ORDER BY wait_duration_ms DESC) AS row_num
    FROM sys.dm_os_waiting_tasks
) w ON (t.task_address = w.waiting_task_address) AND w.row_num = 1
LEFT OUTER JOIN sys.dm_exec_requests r2 ON (s.session_id = r2.blocking_session_id)
LEFT OUTER JOIN sys.sysprocesses p ON (s.session_id = p.spid)
OUTER APPLY sys.dm_exec_sql_text (ISNULL(r.[sql_handle], c.most_recent_sql_handle)) AS txt
WHERE s.is_user_process = 1
AND (r2.session_id IS NOT NULL AND (r.blocking_session_id = 0 OR r.session_id IS NULL))
OR blocked > 0
ORDER BY [HeadBlocker] desc, s.session_id;
```

◀ ━━━ ▶

Step 2

To catch long-running or uncommitted transactions, use the following query for identifying current open transactions. It will also indicate which databases are involved in each transaction.

```sql
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];
```

Step 3

Use the following query to return status information about open transactions. Consider a transaction's current state, its `transaction_begin_time`, and other situational data to evaluate how it could contribute to the blocking situation.

```sql
SELECT tst.session_id, [database_name] = db_name(s.database_id)
, tat.transaction_begin_time
, transaction_duration_s = datediff(s, tat.transaction_begin_time, sysdatetime())
, 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 'Distributed 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 initialized yet.'
            WHEN 1 THEN 'The transaction has been initialized but has not started.'
            WHEN 2 THEN 'The transaction is active - has not been committed 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
, azure_dtc_state = CASE tat.dtc_state
                    WHEN 1 THEN 'ACTIVE'
                    WHEN 2 THEN 'PREPARED'
                    WHEN 3 THEN 'COMMITTED'
                    WHEN 4 THEN 'ABORTED'
                    WHEN 5 THEN 'RECOVERED' END
, 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;
```

Step 4

This query looks at the issue from a different angle. It starts with the locks that are involved in the blocking, then shows the affected table, the involved session IDs, and the wait status. You can filter on the table name to reduce the result further.

```sql
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>';
```

Section: Blocking - Troubleshoot and resolve blocking scenarios

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

- The solution to this type of blocking problem is to look for ways to optimize the head blocker, blocking query. The issue is usually a performance problem. If you've confirmed this to be the cause, troubleshoot the specific slow-running query. The performance tools in the Azure portal for Azure SQL Database will be a good starting point, especially the Query Performance Insight tool.
- If you have a long-running query that is blocking other users and can't be optimized, consider moving it from an OLTP environment to a dedicated reporting system.

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

- This type of blocking can often be identified by a session that's sleeping or awaiting a command, yet whose transaction nesting level (`SELECT @@TRANCOUNT` , `open_transaction_count` in `sys.dm_exec_requests`) is greater than zero. This can occur if the application experiences a query time-out or issues a cancel without also issuing the required number of ROLLBACK and/or COMMIT statements. A time-out or cancellation doesn't automatically roll back or commit the transaction. The application is responsible for this, as Azure SQL Database can't assume that an entire transaction must be rolled back due to a single query being canceled.
- See [Detailed blocking scenarios](#) ☒ for further details about resolving this type of issue.

3. Blocking caused by a SPID whose corresponding client application didn't 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 doesn't fetch all result rows, locks can be left on the tables, blocking other users. If the application can't be configured to do so, you may be unable to resolve the blocking problem.
- The impact of this scenario is reduced when *read committed snapshot* is enabled on the database, which is the default configuration in Azure SQL Database.
- To avoid the problem, you can restrict poorly behaved applications to a reporting or a decision-support database, separate from the main OLTP database.

4. Blocking caused by an orphaned connection

- If the client application crashes, disconnects from the network, or if the client workstation is restarted, the network session to the server may not be immediately canceled under some conditions. From the Azure SQL Database perspective, the client still appears to be present, and any locks acquired may still be retained.
- **Resolution**: If the client application has disconnected without appropriately cleaning up its resources, you can terminate the session by using the *KILL* command. The KILL command takes the session ID value as input. For example, to kill session ID 99, issue the SQL command: `KILL 99` .

Section: Deadlock - Overview

A deadlock occurs when two or more tasks permanently block one another because each task has a lock on a resource that the other task is trying to lock. A deadlock is also called a cyclic dependency: in the case of a two-task deadlock, transaction A has a dependency on transaction B, and transaction B closes the circle by having a dependency on transaction A. Complex deadlock scenarios can consist of three and many more tasks in the deadlock cycle.

For troubleshooting, in addition to the following sections below, also review and work through [Analyze and prevent deadlocks in Azure SQL Database](#) ⧉. This article shows you how to identify deadlocks in Azure SQL Database, use deadlock graphs and Query Store to identify the queries in the deadlock, and plan and test changes to prevent deadlocks from reoccurring.

Section: Deadlock - Deadlocks on your database for the past 24 hours

The following chart shows the number of deadlocks on your database for the past 24 hours.

To come back to this screen, select the **X** at the upper-right corner for closing the metrics page.
<Chart displayed to customer>

Section: Deadlock - Capture deadlock details

Step 1

The following query provides you with an overview list of the recent deadlocks at the server:

```
-- run in database master
WITH CTE AS (
      SELECT CAST(event_data AS XML)  AS [target_data_XML]
      FROM sys.fn_xe_telemetry_blob_target_read_file('dl', null, null, null)
)
SELECT
    target_data_XML.value('(/event/@timestamp)[1]', 'DateTime2') AS Timestamp,
    target_data_XML.query('/event/data[@name=''xml_report'']/value/deadlock') AS deadlock_xml,
    target_data_XML.query('/event/data[@name=''database_name'']/value').value('(/value)[1]', 'nvarchar(100)')
FROM CTE
```

If only a subset of the deadlocks is important to you, then the result will help with narrowing down on a specific database and point in time when the deadlocks were occurring. The `deadlock_xml` column contains all the relevant information, including the SQL statements, processes, tables, locks, and indexes that were involved in the deadlock.

Use the following steps to see a graphical representation of a deadlock:

- Copy the `deadlock_xml` column results from the previous query and load them into a text file. If more than one row is returned, be sure to load each row result separately.
- Save the file as a `.xdl` extension, for example, `deadlock.xdl`. This file can be viewed in tools such as [SQL Server Management Studio (SSMS)](#) ⧉ as a deadlock report or graph.

Step 2

If you need to analyze many deadlocks, the following query will help by providing you with the SQL query text, the resource it's waiting on, and a count of how often each one has occurred. This will give you a trend of what specific queries and objects are causing deadlocks within each database.

```
-- run in database master
WITH CTE AS (
    SELECT CAST(event_data AS XML) AS [target_data_XML]
    FROM sys.fn_xe_telemetry_blob_target_read_file('dl', null, null, null)
)
SELECT [db_name], [query_text], [wait_resource], COUNT(*) as [number_of_deadlocks]
FROM (
    SELECT LTRIM(RTRIM(REPLACE(REPLACE(input_buf.value('.', 'nvarchar(250)'), CHAR(10),' '), CHAR(13),' '))) as
    process_node.value('@waitresource', 'nvarchar(250)') AS [wait_resource],
    target_data_XML.query('/event/data[@name=''database_name'']/value').value('(/value)[1]', 'nvarchar(250)') A
    FROM CTE
    CROSS APPLY target_data_XML.nodes('(/event/data/value/deadlock/process-list/process/inputbuf)') AS T(input_
    CROSS APPLY target_data_XML.nodes('(/event/data/value/deadlock/process-list/process)') AS Q(process_node)
    ) deadlock
GROUP BY [query_text], [wait_resource], [db_name]
ORDER BY [number_of_deadlocks] DESC;
```

◀ ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ ▶

Step 3

This third query helps with filtering for the database, wait resource, and time the deadlock occurred. It also returns the SQL query statements and the deadlock XML graph for further analysis.

```
-- run in database master
WITH CTE AS (
    SELECT CAST(event_data AS XML) AS [target_data_XML]
    FROM sys.fn_xe_telemetry_blob_target_read_file('dl', null, null, null)
)
SELECT [Timestamp], [db_name], [query_text], [wait_resource], [deadlock_xml] FROM (
    SELECT
    target_data_XML.value('(/event/@timestamp)[1]', 'DateTime2') AS Timestamp,
    target_data_XML.query('/event/data[@name=''database_name'']/value').value('(/value)[1]', 'nvarchar(250)') A
    LTRIM(RTRIM(REPLACE(REPLACE(input_buf.value('.', 'nvarchar(250)'), CHAR(10),' '), CHAR(13),' '))) as [query
    process_node.value('@waitresource', 'nvarchar(250)') AS [wait_resource],
    deadlock_node.query('.') as [deadlock_xml]
    FROM CTE
    CROSS APPLY target_data_XML.nodes('(/event/data/value/deadlock)') AS T(deadlock_node)
    CROSS APPLY target_data_XML.nodes('(/event/data/value/deadlock/process-list/process)') AS U(process_node)
    CROSS APPLY target_data_XML.nodes('(/event/data/value/deadlock/process-list/process/inputbuf)') AS Q(input_
    ) deadlock
WHERE
    [db_name] = '<database name>'
--   AND [wait_resource] = '<wait resource>'
--   AND [Timestamp] = '<timestamp>'
```

◀ ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ ▶

Section: Deadlock - Use Extended Events to customize the captured deadlock events

If you need to customize the events that you capture when a deadlock occurs, create your own Extended Events Session ⧉ with the following events for a deadlock:

- `deadlock_report_xml` : This returns the deadlock graph.
- `Lock_Deadlock` : Occurs when an attempt to acquire a lock is canceled for the deadlock victim.
- `Lock_deadlock_chain` : Occurs when an attempt to acquire a lock generates a deadlock. This event is raised for each participant in the deadlock.

Here is a sample script for capturing and viewing deadlock-related Extended Events:

```
-- Create the event session
CREATE EVENT SESSION [deadlocks] ON DATABASE
    ADD EVENT database_xml_deadlock_report
    ADD TARGET package0.ring_buffer(SET max_memory=(3072))
    WITH (STARTUP_STATE=OFF);
GO

-- Start the event collection
ALTER EVENT SESSION [deadlocks] ON DATABASE STATE = START;
GO

-- View the deadlock graph after the deadlock has occurred
DECLARE @x XML;
SELECT @x = CAST(st.target_data AS XML)
FROM sys.dm_xe_database_sessions AS se
INNER JOIN sys.dm_xe_database_session_targets AS st   ON se.address like st.event_session_address
WHERE se.name = 'deadlocks';

SELECT @x;
GO

-- Stop and drop the event session if it is no longer needed
ALTER EVENT SESSION [deadlocks] ON DATABASE STATE = STOP;
DROP EVENT SESSION [deadlocks] ON DATABASE;
GO
```

Section: Deadlock - Handling deadlocks

When an instance of the SQL Server Database engine chooses a transaction as a deadlock victim, it terminates the current batch, rolls back the transaction, and returns error message 1205 to the application:

> Your transaction (process ID #52) was deadlocked on {lock | communication buffer | thread} resources with another process and has been chosen as the deadlock victim. Rerun your transaction.

Applications should have an error handler that can trap error message 1205. If the error remains unhandled, the application can proceed unaware that its transaction has been rolled back, causing data and application errors later. The error handler can take remedial action, like automatically resubmitting the query that was involved in the deadlock. By resubmitting the query automatically, the user doesn't need to know that a deadlock occurred.

The application should pause briefly before resubmitting its query. This gives the other transaction involved in the deadlock a chance to complete and release its locks that formed part of the deadlock cycle. This minimizes the likelihood of the deadlock reoccurring when the resubmitted query requests its locks.

Section: Deadlock - Troubleshooting and resolving deadlocks

Although deadlocks can't be completely avoided, following certain coding conventions can minimize the chance of generating a deadlock. Minimizing deadlocks can increase transaction throughput and reduce system overhead because fewer transactions are:

- Rolled back, undoing all the work performed by the transaction.
- Resubmitted by applications because they were rolled back when deadlocked.

Deadlocks need to be addressed from the application development side and the following topics will help minimize the likelihood and impact of deadlocks.

1. Access objects in the same order

If an application needs to update several tables within a transaction and updates the same tables through different code paths, it's important to access the objects in the same order through all code paths. Concurrent transactions might then still block each other. But if they maintain the same order, they are less likely to run into deadlocks. If the accessing order is random, it's more likely that concurrent transactions request locks from each other that are already held by the other transaction.

Here is an example for application code that increases the likelihood of deadlocks. Note the reversed order when updating the tables.

```
-- connection 1
BEGIN TRANSACTION
UPDATE table1 SET col1 = 'newdata' WHERE ID = 123
UPDATE table2 SET col2 = 'newdata' WHERE ID = 321
...

-- connection 2
BEGIN TRANSACTION
UPDATE table2 SET col2 = 'newdata' WHERE ID = 321
UPDATE table1 SET col1 = 'newdata' WHERE ID = 123
...
```

2. Avoid user interaction in transactions

Avoid writing transactions that include user interaction, because the speed of batches running without user intervention is much faster than the speed at which a user must manually respond to queries, such as replying to a prompt for a parameter requested by an application. This degrades system throughput because any locks held by the transaction are released only when the transaction is committed or rolled back. Even if a deadlock situation doesn't arise, other transactions accessing the same resources are blocked while waiting for the transaction to be completed.

3. Keep transactions short and in one batch

A deadlock typically occurs when several long-running transactions execute concurrently in the same database. The longer the transaction, the longer the exclusive or update locks are held, blocking other activity, and leading to possible deadlock situations. Keeping transactions in one batch minimizes network roundtrips during a transaction, reducing possible delays in completing the transaction and releasing locks.

4. Use a lower isolation level

Determine whether a transaction can run at a lower isolation level to reduce lock contention. Using a lower isolation level, such as read committed, holds shared locks for a shorter duration than a higher isolation level, such as serializable.

5. Use a row versioning-based isolation level

When possible for the application, set the `READ_COMMITTED_SNAPSHOT` database option to `ON`. READ_COMMITTED_SNAPSHOT ON is the default on Azure SQL Database, but it might have been customized. With `READ_COMMITTED_SNAPSHOT` on, a transaction uses row versioning rather than shared locks during read operations. Locks aren't used to protect the data from updates by other transactions, thus decreasing the probability of blocking and deadlocks.

To confirm that the `READ_COMMITTED_SNAPSHOT` option and/or snapshot isolation are enabled, connect to your Azure SQL Database and run the following query:

```
SELECT name, is_read_committed_snapshot_on, snapshot_isolation_state_desc
FROM sys.databases
WHERE name = DB_NAME();
```

If `READ_COMMITTED_SNAPSHOT` is on, the `is_read_committed_snapshot_on` column will return the value `1`. If snapshot isolation is enabled, the `snapshot_isolation_state_desc` column will return the value `ON`.

Further details about these items are discussed in the [Minimizing deadlocks](#) ⧉ paragraph of the [Transaction locking and row versioning guide](#) ⧉.

Section: Resources

- [Understand and resolve blocking](#) ⧉
- [Find blocking queries](#) ⧉
- [Deadlock analysis for SQL Azure Database](#) ⧉
- Learn about [Deadlocks, including interpreting Deadlock Graphs](#) ⧉
- [Snapshot Isolation in SQL Server](#) ⧉
- [Learn more about Read Committed Snapshot Isolation in the Transaction Locking and Row Versioning Guide](#) ⧉
- [Transaction locking and row versioning guide](#) ⧉
- [Monitoring and performance tuning](#) ⧉

How good have you found this content?