Deadlocks

Last updated by | Peter Hewitt | Nov 22, 2022 at 1:23 AM PST

Contents

Introduction

A deadlock is a circular blocking chain. It occurs when two different transactions are waiting to acquire a lock on a resource that is being held by the other transaction. Since both transactions are waiting on each other, one of them needs to be killed so the other can proceed.

Investigation

The tools and methods to investigate deadlocks.

1. ASC

ASC is the first place to check for details of recent deadlocks. From ASC, generate a *Troubleshooting Report* covering the time period of the issue.

ASC >> Tools >> SQL Troubleshooter >> Create Report

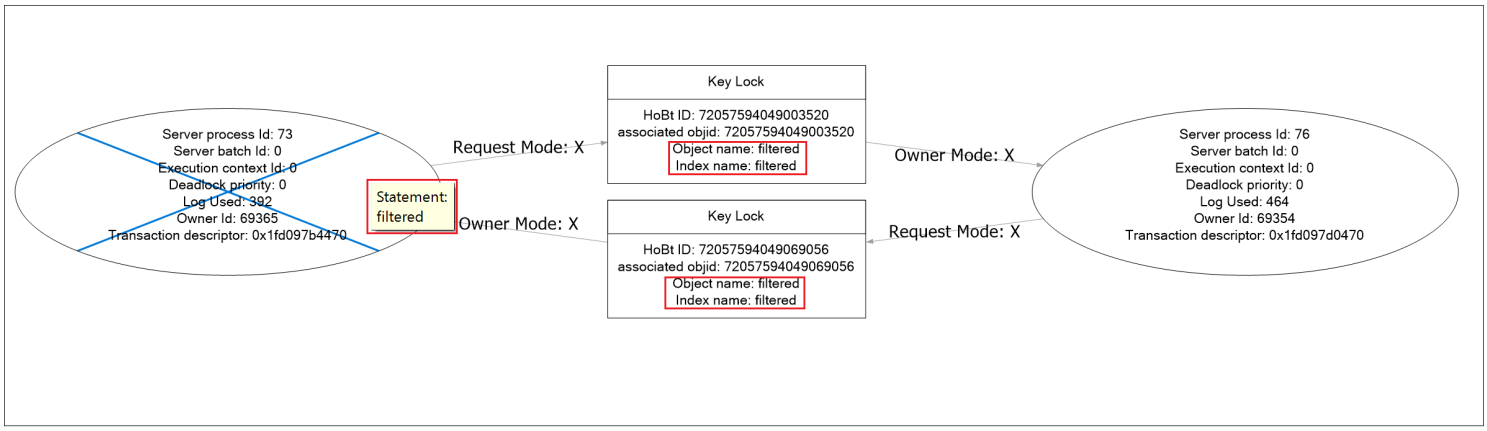
From the Performance section of the *Troubleshooting Report* click on the *Blocking & Deadlocking* tab. This contains details such as:

- Long Running Transactions Summary
- Number of Blocked Sessions and Deadlocks
- Blocking Sessions Summary
- Blocking Sessions Detail
- Top 10 Deadlocks
- Row and Page Lock Status

Of these, the *Top 10 Deadlocks* section contains the deadlock timestamp, deadlock XML report, associated node name, query hashes, query plan hashes and if a complete deadlock graph is available. To view the deadlock graph:

- Save the content from the *xml_report_filtered* column into an .xdl format, for example, deadlock.xdl.
- Open the .xdl file in SSMS to analyse the deadlock graph.

Note - To adhere to privacy restrictions the deadlock graph from ASC contains filtered information such as object names, index names and the T-SQL statements:



2. T-SQL

The T-SQL queries below provided deadlock details.

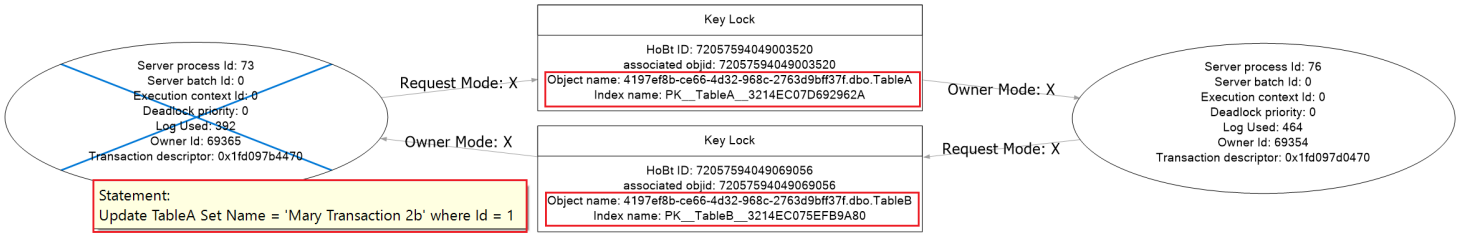
Step 1
The following query returns a list of recent deadlocks for all databases on the server:

```
-- 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
```

The *deadlock_xml* column contains the relevant information, including the T-SQL statements, processes, tables, locks, and indexes that were involved in the deadlock.

To view the deadlock graph:

- Save the content from the *deadlock_xml* column into an .xdl format, for example, deadlock.xdl.
- Open the .xdl file in SSMS to analyse the deadlock graph.



Note: - The information that was filtered when viewing the deadlock graph in ASC is visible in the deadlock graph obtained by running the T-SQL direct on the database.

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>'
```

3. Kusto queries

The Kusto table for analyzing deadlocks is `MonDeadlockReportsFiltered`. Some example queries:

```
MonDeadlockReportsFiltered
| where SubscriptionId =~ "Input SubscriptionId Here"
| where LogicalServerName =~ "server_name" and database_name contains "db_name"
| project TIMESTAMP, xml_report_filtered
//| limit 1000

MonDeadlockReportsFiltered
| where SubscriptionId =~ "Input SubscriptionId Here"
| where LogicalServerName =~ "server_name"
| where xml_report_filtered contains 'spid="spid_id"'
| project TIMESTAMP, xml_report_filtered
```

4. Configure 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
```

Handling deadlocks

When the Lock Monitor performs a deadlock search and detects that one or more sessions are embraced in a deadlock, one of the sessions is selected as a deadlock victim and its current transaction is rolled back. When this occurs, all of the locks held by the victim's session are released, allowing any previously blocked other sessions to continue processing. Once the rollback completes, the victim's session is terminated, returning a 1205 error message to the originating client.

> Error 1205: Transaction (Process ID %d) was deadlocked on %.*ls resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

SQL Server selects the deadlock victim based on the following criteria:

- Deadlock priority – the assigned DEADLOCK_PRIORITY of a given session determines the relative importance of it completing its transactions, if that session is involved in a deadlock. The session with the lowest priority will always be chosen as the deadlock victim. Deadlock priority is covered in more detail later in this article.
- Rollback cost – if two or more sessions involved in a deadlock have the same deadlock priority, then SQL Server will choose as the deadlock victim the session that has lowest estimated cost to roll back.

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.

Mitigation

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.

Public Doc Reference

Lesson Learned #19: How to obtain the deadlocks of your Azure SQL Database
Deadlock analysis for SQL Azure Database
Learn about Deadlocks, including interpreting Deadlock Graphs
Snapshot Isolation in SQL Server
Monitoring and performance tuning

Internal Reference

Workflow for blocking and deadlocks
Deadlocks - types and solutions

Root Cause Classification

Cases resolved by this TSG should be coded to the following root cause:
/Performance/Waits/Locking/Blocking

How good have you found this content?