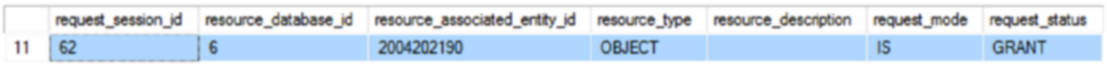
```
FROM    dbo.MyEmployees WITH (HOLDLOCK)
WHERE   GroupID = 10;
```

A clustered index is available on the GroupID column, allowing SQL Server to acquire a (RangeS-S) lock on the row to be accessed and the next row in the correct order.

If the index on the GroupID column is removed, then SQL Server cannot determine the rows on which the range locks should be acquired since the order of the rows is no longer guaranteed. Consequently, the SELECT statement acquires an (IS) lock at the table level instead of acquiring lower-granularity locks at the row level, as shown in Figure 21-13.



	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
11	62	6	2004202190	OBJECT		IS	GRANT

Figure 21-13. *Output from sys.dm_tran_locks showing the locks granted to a SELECT statement with no index on the WHERE clause column*

By failing to have an index on the filter column, you significantly increase the degree of blocking caused by the Serializable isolation level. This is another good reason to have an index on the WHERE clause columns.

Capturing Blocking Information

Although blocking is necessary to isolate a transaction from other concurrent transactions, sometimes it may rise to excessive levels, adversely affecting database concurrency. In the simplest blocking scenario, the lock acquired by a session on a resource blocks another session requesting an incompatible lock on the resource. To improve concurrency, it is important to analyze the cause of blocking and apply the appropriate resolution.

In a blocking scenario, you need the following information to have a clear understanding of the cause of the blocking:

- *The connection information of the blocking and blocked sessions*: You can obtain this information from the sys.dm_os_waiting_tasks dynamic management view.
- *The lock information of the blocking and blocked sessions*: You can obtain this information from the sys.dm_tran_locks DMO.

- *The SQL statements last executed by the blocking and blocked sessions:* You can use the `sys.dm_exec_requests` DMV combined with `sys.dm_exec_sql_text` and `sys.dm_exec_queryplan` or Extended Events to obtain this information.

You can also obtain the following information from SQL Server Management Studio by running the Activity Monitor. The Processes page provides connection information of all SPIDs. This shows blocked SPIDS, the process blocking them, and the head of any blocking chain with details on how long the process has been running, its SPID, and other information. It is possible to put Extended Events to work using the blocking report to gather a lot of the same information. For immediate checks on locking, use the DMOs; for extended monitoring and historical tracking, you'll want to use Extended Events. You can find more on this in the "Extended Events and the blocked_process_report Event" section.

To provide more power and flexibility to the process of collecting blocking information, a SQL Server administrator can use SQL scripts to provide the relevant information listed here.

Capturing Blocking Information with SQL

To arrive at enough information about blocked and blocking processes, you can bring several dynamic management views into play. This query will show information necessary to identify blocked processes based on those that are waiting. You can easily add filtering to access only those processes blocked for a certain period of time or only within certain databases, among other options.

```
SELECT  dtl.request_session_id AS WaitingSessionID,
        der.blocking_session_id AS BlockingSessionID,
        dowt.resource_description,
        der.wait_type,
        dowt.wait_duration_ms,
        DB_NAME(dtl.resource_database_id) AS DatabaseName,
        dtl.resource_associated_entity_id AS WaitingAssociatedEntity,
        dtl.resource_type AS WaitingResourceType,
        dtl.request_type AS WaitingRequestType,
        dest.[text] AS WaitingTSql,
        dtlbl.request_type BlockingRequestType,
        destbl.[text] AS BlockingTsql
FROM    sys.dm_tran_locks AS dtl
JOIN    sys.dm_os_waiting_tasks AS dowt
        ON dtl.lock_owner_address = dowt.resource_address
JOIN    sys.dm_exec_requests AS der
        ON der.session_id = dtl.request_session_id
CROSS APPLY sys.dm_exec_sql_text(der.sql_handle) AS dest
LEFT JOIN sys.dm_exec_requests derbl
        ON derbl.session_id = dowt.blocking_session_id
OUTER APPLY sys.dm_exec_sql_text(derbl.sql_handle) AS destbl
LEFT JOIN sys.dm_tran_locks AS dtlbl
        ON derbl.session_id = dtlbl.request_session_id;
```

To understand how to analyze a blocking scenario and the relevant information provided by the blocker script, consider the following example. First, create a test table.

```
DROP TABLE IF EXISTS dbo.BlockTest;
GO

CREATE TABLE dbo.BlockTest (C1 INT,
                            C2 INT,
                            C3 DATETIME);

INSERT INTO dbo.BlockTest
VALUES (11, 12, GETDATE()),
       (21, 22, GETDATE());
```

Now open three connections and run the following two queries concurrently. Once you run them, use the blocker script in the third connection. Execute the following code in one connection:

```
BEGIN TRAN User1
UPDATE  dbo.BlockTest
SET     C3 = GETDATE();
```

Next, execute this code while the `User1` transaction is executing:

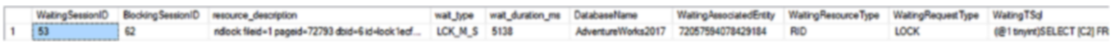
```
BEGIN TRAN User2
SELECT  C2
FROM    dbo.BlockTest
WHERE   C1 = 11;
COMMIT
```

This creates a simple blocking scenario where the `User1` transaction blocks the `User2` transaction.

The output of the blocker script provides information immediately useful to begin resolving blocking issues. First, you can identify the specific session information, including the session ID of both the blocking and waiting sessions. You get an immediate resource description from the waiting resource, the wait type, and the length of time in milliseconds that the process has been waiting. It's that value that allows you to provide a filter to eliminate short-term blocks, which are part of normal processing.

The database name is supplied because blocking can occur anywhere in the system, not just in AdventureWorks2017. You'll want to identify it where it occurs. The resources and types from the basic locking information are retrieved for the waiting process.

The blocking request type is displayed, and both the waiting T-SQL and blocking T-SQL, if available, are displayed. Once you have the object where the block is occurring, having the T-SQL so that you can understand exactly where and how the process is either blocking or being blocked is a vital part of the process of eliminating or reducing the amount of blocking. All this information is available from one simple query. Figure 21-14 shows the sample output from the earlier blocked process.



	WaitingSessionID	BlockingSessionID	resource_description	wait_type	wait_duration_ms	DatabaseName	WaitingAssociatedEntity	WaitingResourceType	WaitingRequestType	WaitingTSql
1	53	62	ridlock fileid=1 pageid=72793 dbid=6 id=lock1ecf...	LCK_M_S	5138	AdventureWorks2017	72057594078429184	RID	LOCK	(@1 tinyint)SELECT [C2] FR

Figure 21-14. *Output from the blocker script*

Be sure to go back to Connection 1 and commit or roll back the transaction.

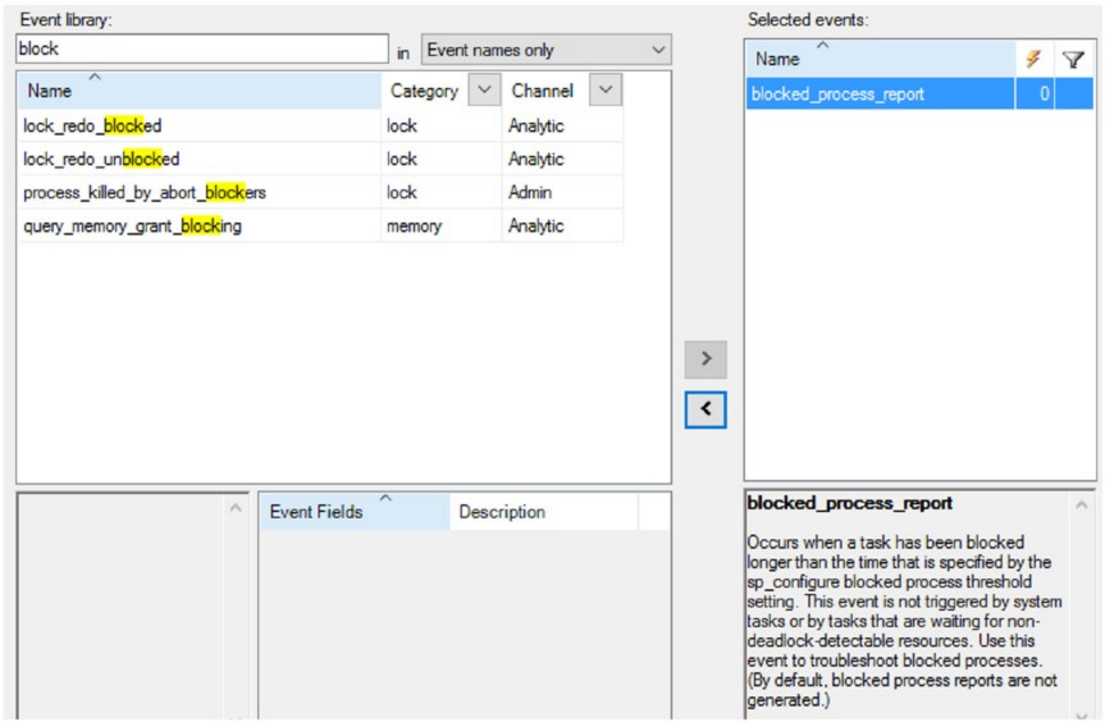
Extended Events and the blocked_process_report Event

Extended Events provides an event called blocked_process_report. This event works off the blocked process threshold that you need to provide to the system configuration. This script sets the threshold to five seconds:

```
EXEC sp_configure 'show advanced option', '1';
RECONFIGURE;
EXEC sp_configure
    'blocked process threshold',
    5;
RECONFIGURE;
```

This would normally be a very low value in most systems. If you have an established performance service level agreement (SLA), you could use that as the threshold. Once the value is set, you can configure alerts so that e-mails, tweets, or instant messages are sent if any process is blocked longer than the value you set. It also acts as a trigger for the extended event. The default value for the blocked process threshold is zero, meaning that it never actually fires. If you are going to use Extended Events to track blocked processes, you will want to adjust this value from the default.

To set up a session that captures the blocked_process_report, first open the Extended Events session properties window. (Although you should use scripts to set up this event in a production environment, I'll show how to use the GUI.) Provide the session with a name and then navigate to the Events page. Type **block** into the "Event library" text box, which will find the blocked_process_report event. Select that event by clicking the right arrow. You should see something similar to Figure 21-15.


Event library:
block in Event names only

Name	Category	Channel
lock_redo_blocked	lock	Analytic
lock_redo_unblocked	lock	Analytic
process_killed_by_abort_blockers	lock	Admin
query_memory_grant_blocking	memory	Analytic

Event Fields Description

Selected events:

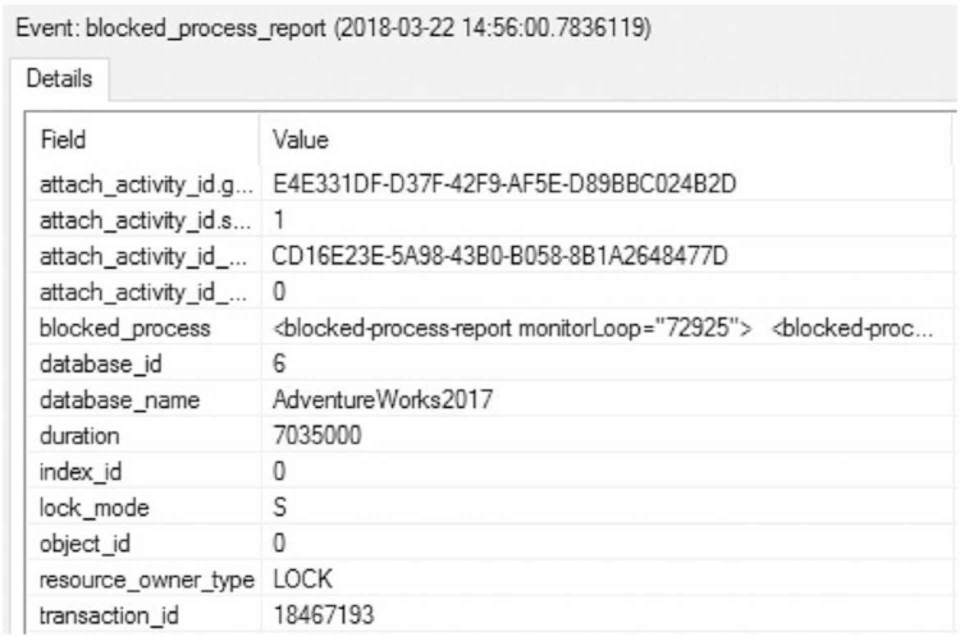
Name		
blocked_process_report	0	

blocked_process_report

Occurs when a task has been blocked longer than the time that is specified by the sp_configure blocked process threshold setting. This event is not triggered by system tasks or by tasks that are waiting for non-deadlock-detectable resources. Use this event to troubleshoot blocked processes. (By default, blocked process reports are not generated.)

Figure 21-15. *The blocked process report event selected in the Extended Events window*

The event fields are all preselected for you. If you still have the queries running from the previous section that created the block, all you need to do now is click the Run button to capture the event. Otherwise, go back to the queries we used to generate the blocked process report in the previous section and run them in two different connections. After the blocked process threshold is passed, you'll see the event fire…and fire. It will fire every five seconds if that's how you've configured it and you're leaving the connections running. The output in the live data stream looks like Figure 21-16.


Event: blocked_process_report (2018-03-22 14:56:00.7836119)

Details

Field	Value
attach_activity_id.g...	E4E331DF-D37F-42F9-AF5E-D89BBC024B2D
attach_activity_id.s...	1
attach_activity_id_...	CD16E23E-5A98-43B0-B058-8B1A2648477D
attach_activity_id_...	0
blocked_process	<blocked-process-report monitorLoop="72925"> <blocked-proc...
database_id	6
database_name	AdventureWorks2017
duration	7035000
index_id	0
lock_mode	S
object_id	0
resource_owner_type	LOCK
transaction_id	18467193

Figure 21-16. *Output from the blocked_process_report event*

Some of the information is self-explanatory; to get into the details, you need to look at the XML generated in the blocked_process field.

```
<blocked-process-report monitorLoop="72925">
 <blocked-process>
  <process id="process1edc0b0c108" taskpriority="0" logused="0"
   waitresource="RID: 6:1:72793:0" waittime="7035" ownerId="18467193"
   transactionname="User2" lasttranstarted="2018-03-22T14:55:53.743"
   XDES="0x1edf1bc0490" lockMode="S" schedulerid="1" kpid="14036"
   status="suspended" spid="53" sbid="0" ecid="0" priority="0" trancount="1"
   lastbatchstarted="2018-03-22T14:55:53.743" lastbatchcompleted="2018-
   03-22T14:55:53.740" lastattention="1900-01-01T00:00:00.740"
   clientapp="Microsoft SQL Server Management Studio - Query" hostname="WIN-
   8A2LQANSO51" hostpid="5540" loginname="WIN-8A2LQANSO51\Administrator"
   isolationlevel="read committed (2)" xactid="18467193" currentdb="6"
   lockTimeout="4294967295" clientoption1="671090784" clientoption2="390200">
   <executionStack>
    <frame line="2" stmtstart="24" stmtend="118" sqlhandle="0x02000000ccf3e60
     45e680885750c3f36d7cc549d8ff0136800000000000000000000000000000000000000000"/>
    <frame line="2" stmtstart="36" stmtend="134" sqlhandle="0x0200000063e12d309
     fa7874804b7b56c7be7beecf2a0255b000000000000000000000000000000000000000000"/>
   </executionStack>
   <inputbuf>
BEGIN TRAN User2
SELECT  C2
FROM    dbo.BlockTest
WHERE   C1 = 11;
COMMIT   </inputbuf>
  </process>
 </blocked-process>
 <blocking-process>
  <process status="sleeping" spid="62" sbid="0" ecid="0" priority="0"
   trancount="1" lastbatchstarted="2018-03-22T14:55:50.923"
   lastbatchcompleted="2018-03-22T14:55:50.927" lastattention="1900-01-
   01T00:00:00.927" clientapp="Microsoft SQL Server Management Studio -
   Query" hostname="WIN-8A2LQANSO51" hostpid="5540" loginname="WIN-
   8A2LQANSO51\Administrator" isolationlevel="read committed
   (2)" xactid="18467189" currentdb="6" lockTimeout="4294967295"
   clientoption1="671090784" clientoption2="390200">
   <executionStack />
   <inputbuf>
BEGIN TRAN User1
UPDATE  dbo.BlockTest
SET     C3 = GETDATE();   </inputbuf>
  </process>
 </blocking-process>
</blocked-process-report>
```

The elements are clear if you look through this XML. `<blocked-process>` shows information about the process that was blocked, including familiar information such as the session ID (labeled with the old-fashioned SPID here), the database ID, and so on. You can see the query in the `<inputbuf>` element. Details such as the `lockMode` are available within the `<process>` element. Note that the XML doesn't include some of the other information that you can easily get from T-SQL queries, such as the query string of the blocked and waiting processes. But with the SPID available, you can get them from the cache, if available, or you can combine the Blocked Process report with other events such as `rpc_starting` to show the query information. However, doing so will add to the overhead of using those events long term within your database. If you know you have a blocking problem, this can be part of a short-term monitoring project to capture the necessary blocking information.

Blocking Resolutions

Once you've analyzed the cause of a block, the next step is to determine any possible resolutions. Here are a few techniques you can use to do this:

- Optimize the queries executed by blocking and blocked SPIDs.
- Decrease the isolation level.
- Partition the contended data.
- Use a covering index on the contended data.

Note A detailed list of recommendations to avoid blocking appears later in the chapter in the "Recommendations to Reduce Blocking" section.

To understand these resolution techniques, let's apply them in turn to the preceding blocking scenario.

Optimize the Queries

Optimizing the queries executed by the blocking and blocked processes helps reduce the blocking duration. In the blocking scenario, the queries executed by the processes participating in the blocking are as follows:

- Blocking process:

```
BEGIN TRAN User1
UPDATE  dbo.BlockTest
SET     C3 = GETDATE();
```

- Blocked process:

  ```
  BEGIN TRAN User2
  SELECT  C2
  FROM    dbo.BlockTest
  WHERE   C1 = 11;
  COMMIT
  ```

Note that beyond the missing COMMIT for the first query, running UPDATE without a WHERE clause is certainly potentially problematic and will not perform well. It will get worse over time as the data scales. However, it is just a test for demonstration purposes.

Next, let's analyze the individual SQL statements executed by the blocking and blocked SPIDs to optimize their performance.

- The UPDATE statement of the blocking SPID accesses the data without a WHERE clause. This makes the query inherently costly on a large table. If possible, break the action of the UPDATE statement into multiple batches using appropriate WHERE clauses. Remember to try to use set-based operations such as a TOP statement to limit the rows. If the individual UPDATE statements of the batch are executed in separate transactions, then fewer locks will be held on the resource within one transaction and for shorter time periods. This could also help reduce or avoid lock escalation.
- The SELECT statement executed by the blocked SPID has a WHERE clause on the C1 column. From the index structure on the test table, you can see that there is no index on this column. To optimize the SELECT statement, you could create a clustered index on the C1 column.

  ```
  CREATE CLUSTERED INDEX i1 ON dbo.BlockTest(C1);
  ```

Note Since the `example` table fits within one page, adding the clustered index won't make much difference to the query performance. However, as the number of rows in the table increases, the beneficial effect of the index will become more pronounced.

Optimizing the queries reduces the duration for which the locks are held by the processes. The query optimization reduces the impact of blocking, but it doesn't prevent the blocking completely. However, as long as the optimized queries execute within acceptable performance limits, a small amount of blocking may be ignored.

Decrease the Isolation Level

Another approach to resolve blocking can be to use a lower isolation level, if possible. The `SELECT` statement of the `User2` transaction gets blocked while requesting an (S) lock on the data row. The isolation level of this transaction can be mitigated by taking advantage of `SNAPSHOT` isolation level Read Committed Snapshot so that the (S) lock is not requested by the `SELECT` statement. The Read Committed Snapshot isolation level can be configured for the connection using the `SET` statement.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
GO
BEGIN TRAN User2
SELECT  C2
FROM    dbo.BlockTest
WHERE   C1 = 11;
COMMIT
GO
--Back to default
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
GO
```

This example shows the utility of reducing the isolation level. Using this `SNAPSHOT` isolation is radically preferred over using any of the methods that produce dirty reads that could lead to incorrect data or missing or extra rows.

Partition the Contended Data

When dealing with large data sets or data that can be discretely stored, it is possible to apply table partitioning to the data. Partitioned data is split horizontally, that is, by certain values (such as splitting sales data up by month, for example). This allows the transactions to execute concurrently on the individual partitions, without blocking each other. These separate partitions are treated as a single unit for querying, updating, and inserting; only the storage and access are separated by SQL Server. It should be noted that partitioning is available only in the Developer and Enterprise editions of SQL Server.

In the preceding blocking scenario, the data could be separated by date. This would entail setting up multiple filegroups if you're concerned with performance (or just put everything on PRIMARY if you're worried about management) and splitting the data per a defined rule. Once the UPDATE statement gets a WHERE clause, then it and the original SELECT statement will be able to execute concurrently on two separate partitions. This does require that the WHERE clause filters only on the partition key column. As soon as you get other conditions in the mix, you're unlikely to benefit from partition elimination, which means performance could be much worse, not better.

Note Partitioning, if done properly, can improve both performance and concurrency on large data sets. But, partitioning is almost exclusively a data management solution, not a performance tuning option.

In a blocking scenario, you should analyze whether the query of the blocking or the blocked process can be fully satisfied using a covering index. If the query of one of the processes can be satisfied using a covering index, then it will prevent the process from requesting locks on the contended resource. Also, if the other process doesn't need a lock on the covering index (to maintain data integrity), then both processes will be able to execute concurrently without blocking each other.

For instance, in the preceding blocking scenario, the SELECT statement by the blocked process can be fully satisfied by a covering index on the C1 and C2 columns.

```
CREATE NONCLUSTERED INDEX iAvoidBlocking ON dbo.BlockTest(C1,  C2) ;
```

The transaction of the blocking process need not acquire a lock on the covering index since it accesses only the C3 column of the table. The covering index will allow the SELECT statement to get the values for the C1 and C2 columns without accessing the base table. Thus, the SELECT statement of the blocked process can acquire an (S) lock on the covering-index row without being blocked by the (X) lock on the data row acquired by the blocking process. This allows both transactions to execute concurrently without any blocking.

Consider a covering index as a mechanism to "duplicate" part of the table data in which consistency is automatically maintained by SQL Server. This covering index, if mostly read-only, can allow some transactions to be served from the "duplicate" data while the base table (and other indexes) can continue to serve other transactions. The trade-offs to this approach are the need for additional storage and the potential for additional overhead during data modification.

Recommendations to Reduce Blocking

Single-user performance and the ability to scale with multiple users are both important for a database application. In a multiuser environment, it is important to ensure that the database operations don't hold database resources for a long time. This allows the database to support a large number of operations (or database users) concurrently without serious performance degradation. The following is a list of tips to reduce/avoid database blocking:

- Keep transactions short.
    - Perform the minimum steps/logic within a transaction.
    - Do not perform costly external activity within a transaction, such as sending an acknowledgment e-mail or performing activities driven by the end user.
    - Optimize queries.
    - Create indexes as required to ensure optimal performance of the queries within the system.
    - Avoid a clustered index on frequently updated columns. Updates to clustered index key columns require locks on the clustered index and all nonclustered indexes (since their row locator contains the clustered index key).
    - Consider using a covering index to serve the blocked `SELECT` statements.

- Use query timeouts or a resource governor to control runaway queries. For more on the resource governor, consult Books Online: http://bit.ly/1jiPhfS.
    - Avoid losing control over the scope of the transactions because of poor error-handling routines or application logic.
    - Use SET XACT_ABORT ON to avoid a transaction being left open on an error condition within the transaction.
    - Execute the following SQL statement from a client error handler (TRY/CATCH) after executing a SQL batch or stored procedure containing a transaction.

      ```
      IF @@TRANCOUNT > 0 ROLLBACK
      ```
- Use the lowest isolation level required.
    - Consider using row versioning, one of the SNAPSHOT isolation levels, to help reduce contention.

Automation to Detect and Collect Blocking Information

In addition to capturing information using extended events, you can automate the process of detecting a blocking condition and collecting the relevant information using SQL Server Agent. SQL Server provides the Performance Monitor counters shown in Table 21-2 to track the amount of wait time.

Table 21-2. *Performance Monitor Counters*

Object	Counter	Instance	Description
SQLServer:Locks (for SQL Server named instance MSSQL$<InstanceName>:Locks)	Average Wait Time(ms)	_Total	Average amount of wait time for each lock that resulted in a wait
	Lock Wait Time (ms)	_Total	Total wait time for locks in the last second

You can create a combination of SQL Server alerts and jobs to automate the following process:

1. Determine when the average amount of wait time exceeds an acceptable amount of blocking using the `Average Wait Time (ms)` counter. Based on your preferences, you can use the `Lock Wait Time (ms)` counter instead.

2. Once you've established the minimum wait, set `Blocked Process Threshold`. When the average wait time exceeds the limit, notify the SQL Server DBA of the blocking situation through e-mail.

3. Automatically collect the blocking information using the blocker script or a trace that relies on the Blocked Process report for a certain period of time.

To set up the Blocked Process report to run automatically, first create the SQL Server job, called Blocking Analysis, so that it can be used by the SQL Server alert you'll create later. You can create this SQL Server job from SQL Server Management Studio to collect blocking information by following these steps:

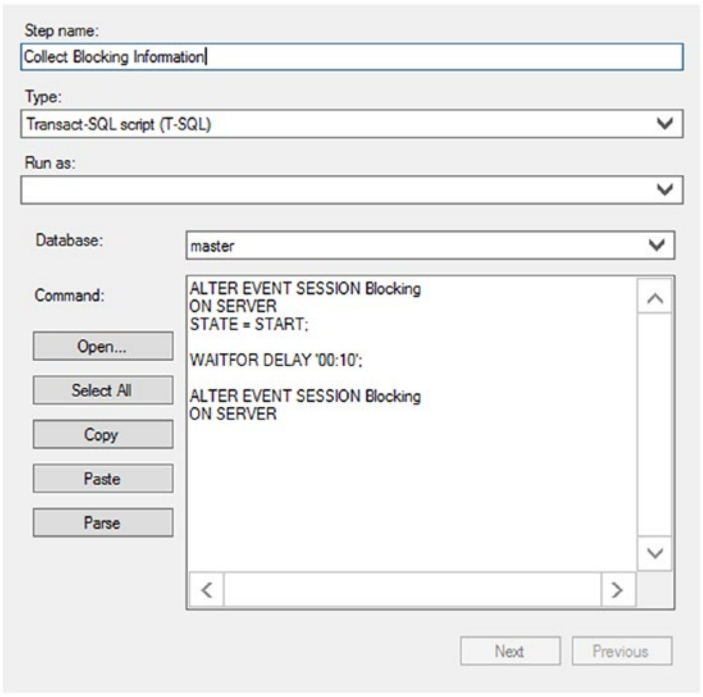
1. Generate an Extended Events script (as detailed in Chapter 6) using the `blocked_process_report` event.

2. Run the script to create the session on the server, but don't start it yet.

3. In Management Studio, expand the server by selecting <*ServerName*> ➤ SQL Server Agent ➤ Jobs. Finally, right-click and select New Job.

4. On the General page of the New Job dialog box, enter the job name and other details.

5. On the Steps page, click New and enter the command to start and stop the session through T-SQL, as shown in Figure 21-17.


Step name: Collect Blocking Information

Type: Transact-SQL script (T-SQL)

Run as:

Database: master

Command:

```
ALTER EVENT SESSION Blocking
ON SERVER
STATE = START;

WAITFOR DELAY '00:10';

ALTER EVENT SESSION Blocking
ON SERVER
```

Figure 21-17. *Entering the command to run the blocker script*

You can do this using the following command:

```
ALTER EVENT SESSION Blocking
ON SERVER
STATE = START;

WAITFOR DELAY '00:10';

ALTER EVENT SESSION Blocking
ON SERVER
STATE = STOP;
```

The output of the session is determined by how you defined the target or targets when you created it.

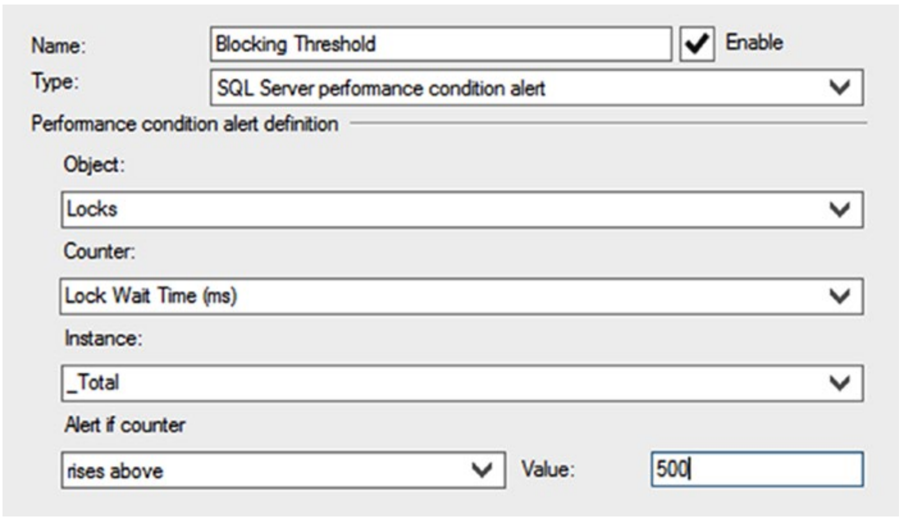
1. Return to the New Job dialog box by clicking OK.
2. Click OK to create the SQL Server job. The SQL Server job will be created with an enabled and runnable state to collect blocking information for ten minutes using the trace script.

You can create a SQL Server alert to automate the following tasks:

- Inform the DBA via e-mail, SMS text, or pager.
- Execute the Blocking Analysis job to collect blocking information for ten minutes.

You can create the SQL Server alert from SQL Server Enterprise Manager by following these steps:

1. In Management Studio, while still in the SQL Agent area of the Object Explorer, right-click Alerts and select New Alert.
2. On the General page of the new alert's Properties dialog box, enter the alert name and other details, as shown in Figure 21-18. The specific object you need to capture information from for your instance is `Locks` (`MSSQL$GF2008:Locks` in Figure 21-18). I chose `500ms` as an example of a stringent SLA that wants to know when queries extend beyond that value.



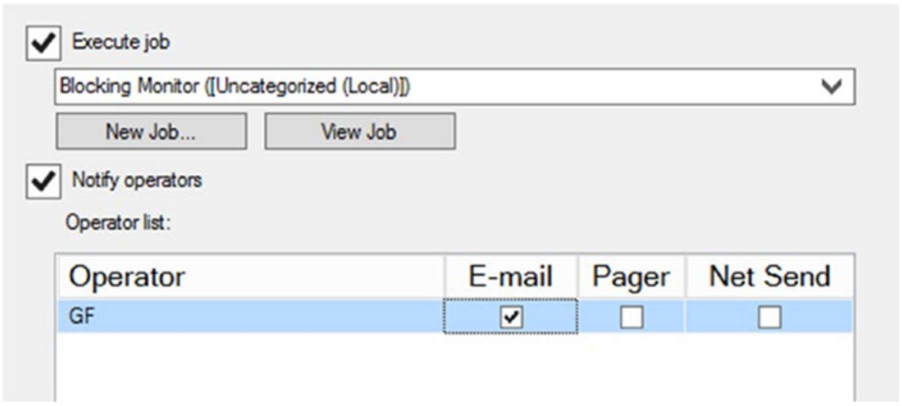
Name:	Blocking Threshold [✓ Enable]
Type:	SQL Server performance condition alert

Performance condition alert definition

Object:	Locks
Counter:	Lock Wait Time (ms)
Instance:	_Total
Alert if counter	rises above
Value:	500

Figure 21-18. *Entering the alert name and other details*

1. On the Response page, define the response you think appropriate, such as alerting an operator.
2. Return to the new alert's Properties dialog box by clicking OK.
3. On the Response page, enter the remaining information shown in Figure 21-19.



✓ Execute job

Blocking Monitor ([Uncategorized (Local)])

✓ Notify operators

Operator list:

Operator	E-mail	Pager	Net Send
GF	✓	☐	☐

Figure 21-19. *Entering the actions to be performed when the alert is triggered*

4. The Blocking Analysis job is selected to automatically collect the blocking information.
5. Once you've finished entering all the information, click OK to create the SQL Server alert. The SQL Server alert will be created in the enabled state to perform the intended tasks.
6. Ensure that the SQL Server Agent is running.

Together, the SQL Server alert and the job will automate the blocking detection and the information collection process. This automatic collection of the blocking information will ensure that a good amount of the blocking information will be available whenever the system gets into a massive blocking state.

Summary

Even though blocking is inevitable and is in fact essential to maintain isolation among transactions, it can sometimes adversely affect database concurrency. In a multiuser database application, you must minimize blocking among concurrent transactions.

SQL Server provides different techniques to avoid/reduce blocking, and a database application should take advantage of these techniques to scale linearly as the number of database users increases. When an application faces a high degree of blocking, you can collect the relevant blocking information using various tools to understand the root cause of the blocking. The next step is to use an appropriate technique to either avoid or reduce blocking.

Blocking not only can hurt concurrency but can lead to an abrupt termination of a database request in the case of mutual blocking between processes or even within a process. We will cover this event, known as a *deadlock*, in the next chapter.

CHAPTER 22

Causes and Solutions for Deadlocks

In the preceding chapter, I discussed how blocking works. Blocking is one of the primary causes of poor performance. Blocking can lead to a special situation referred to as a *deadlock*, which in turn means that deadlocks are fundamentally a performance problem. When a deadlock occurs between two or more transactions, SQL Server allows one transaction to complete and terminates the other transaction, rolling back the transaction. SQL Server then returns an error to the corresponding application, notifying the user that he has been chosen as a deadlock victim. This leaves the application with only two options: resubmit the transaction or apologize to the end user. To successfully complete a transaction and avoid the apologies, it is important to understand what might cause a deadlock and the ways to handle a deadlock.

In this chapter, I cover the following topics:

- Deadlock fundamentals
- Error handling to catch a deadlock
- Ways to analyze the cause of a deadlock
- Techniques to resolve a deadlock

Deadlock Fundamentals

A *deadlock* is a special blocking scenario in which two processes get blocked by each other. Each process, while holding its own resources, attempts to access a resource that is locked by the other process. This will lead to a blocking scenario known as a *deadly embrace,* as illustrated in Figure 22-1.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_22