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.

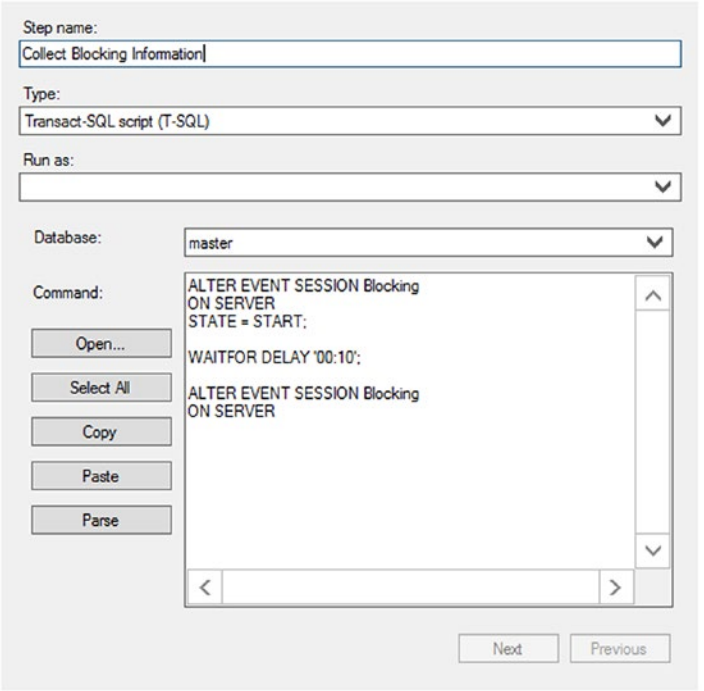

Figure 21-17. *Entering the command to run the blocker script*

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.

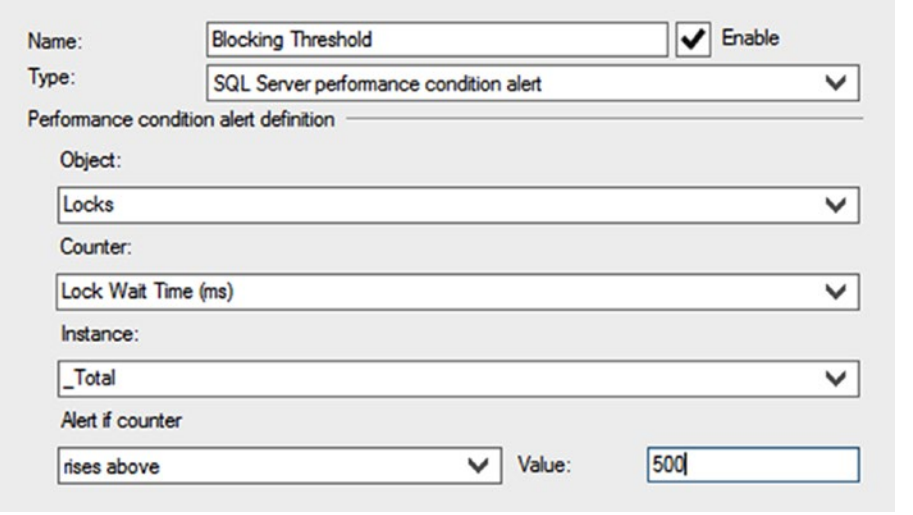

Figure 21-18. *Entering the alert name and other details*

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.

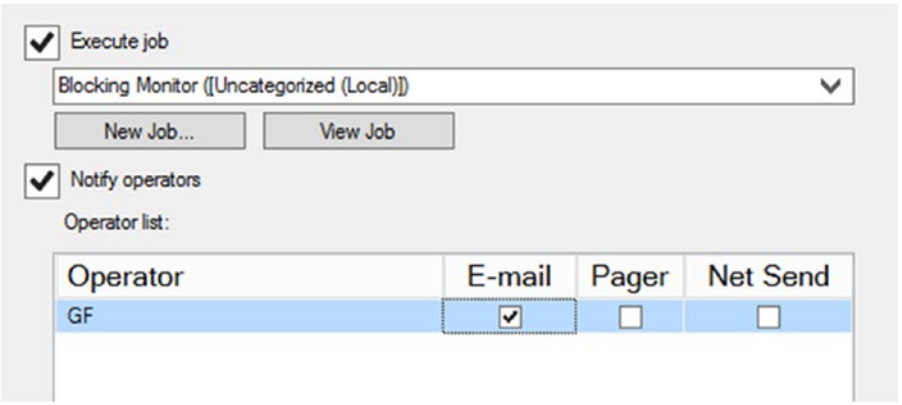

Figure 21-19. *Entering the actions to be performed when the alert is triggered*

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

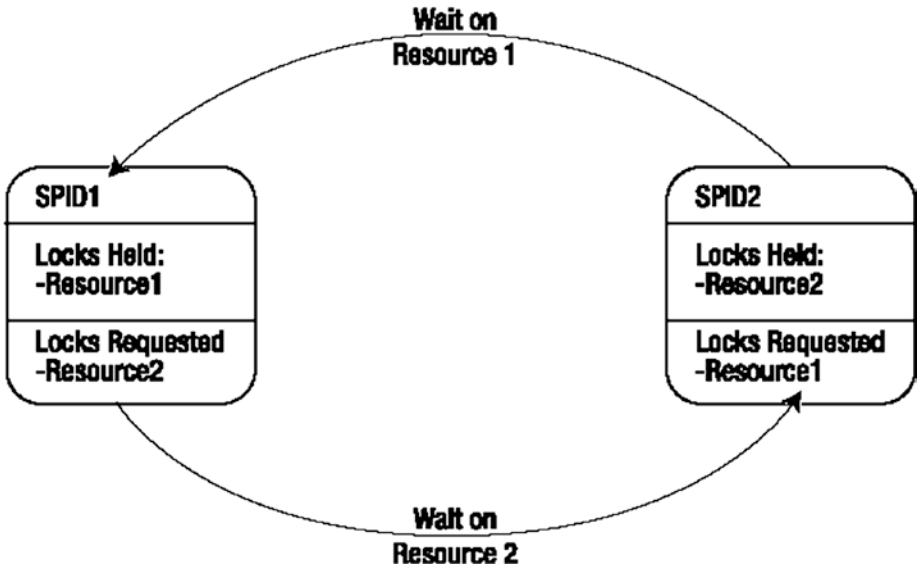

Figure 22-1. *A deadlock scenario*

Deadlocks also frequently occur when two processes attempt to escalate their locking mechanisms on the same resource. In this case, each of the two processes has a shared lock on a resource, such as an `RID`, and each attempts to promote the lock from shared to exclusive; however, neither can do so until the other releases its shared lock. This too leads to one of the processes being chosen as a deadlock victim.

Finally, it is possible for a single process to get a deadlock during parallel operations. During parallel operations, it's possible for a thread to be holding a lock on one resource, A, while waiting for another resource, B; at the same time, another thread can have a lock on B while waiting for A. This is as much a deadlock situation as when multiple processes are involved but instead involves multiple threads from one process. This is a rare event, but it is possible and is generally considered a bug that has probably been fixed by a Cumulative Update.

Deadlocks are an especially nasty type of blocking because a deadlock cannot resolve on its own, even if given an unlimited period of time. A deadlock requires an external process to break the circular blocking.

SQL Server has a deadlock detection routine, called a *lock monitor,* that regularly checks for the presence of deadlocks in SQL Server. Once a deadlock condition is detected, SQL Server selects one of the sessions participating in the deadlock as a *victim* to break the circular blocking. The victim is usually the process with the lowest estimated cost since this implies that process will be the easiest one for SQL Server to roll back. This

operation involves withdrawing all the resources held by the victim session. SQL Server does so by rolling back the uncommitted transaction of the session picked as a victim.

Deadlocks are a performance issue and, like any performance issue, need to be dealt with. Like other performance issues, there is a general threshold of pain. The occasional rare deadlock is not a cause for alarm. However, frequent and consistent deadlocks certainly are. Just as you may get a query that on rare occasions runs a little long and doesn't need a lot of tuning attention, you may run into deadlock situations that also don't need your focus. Be sure you're working on the most painful parts of your system.

Choosing the Deadlock Victim

SQL Server determines the session to be a deadlock victim by evaluating the cost of undoing the transaction of the participating sessions, and it selects the one with the least estimated cost. You can exercise some control over the session to be chosen as a victim by setting the deadlock priority of its connection to `LOW`.

```
SET DEADLOCK_PRIORITY LOW;
```

This steers SQL Server toward choosing this particular session as a victim in the event of a deadlock. You can reset the deadlock priority of the connection to its normal value by executing the following `SET` statement:

```
SET DEADLOCK_PRIORITY NORMAL;
```

The `SET` statement allows you to mark a session as a `HIGH` deadlock priority, too. This won't prevent deadlocks on a given session, but it will reduce the likelihood of a given session being picked as the victim. You can even set the priority level to a number value from –10 for the lowest priority up to 10 for the highest.

Caution Setting the deadlock priority is not something that should be applied promiscuously. You could accidently set the priority on a report that causes mission-critical processes to be chosen as a victim. Careful testing is necessary with this setting.

In the event of a tie, one of the processes is chosen as a victim and rolled back as if it had the least cost. Some processes are invulnerable to being picked as a deadlock victim. These processes are marked as such in the deadlock graph and will never be chosen as

a deadlock victim. The most common example that I've seen occurs when processes are already involved in a rollback.

Using Error Handling to Catch a Deadlock

When SQL Server chooses a session as a victim, it raises an error with the error number. You can use the TRY/CATCH construct within T-SQL to handle the error. SQL Server ensures the consistency of the database by automatically rolling back the transaction of the victim session. The rollback ensures that the session is returned to the same state it was in before the start of its transaction. On determining a deadlock situation in the error handler, it is possible to attempt to restart the transaction within T-SQL a number of times before returning the error to the application.

Take the following T-SQL statement as an example of one method for handling a deadlock error:

```
DECLARE @retry AS TINYINT = 1,
        @retrymax AS TINYINT = 2,
        @retrycount AS TINYINT = 0;
WHILE @retry = 1 AND @retrycount <= @retrymax
BEGIN
    SET @retry = 0;

    BEGIN TRY
        UPDATE HumanResources.Employee
        SET LoginID = '54321'
        WHERE BusinessEntityID = 100;
    END TRY
    BEGIN CATCH
        IF (ERROR_NUMBER() = 1205)
        BEGIN
            SET @retrycount = @retrycount + 1;
            SET @retry = 1;
        END
    END CATCH
END
```

The TRY/CATCH methodology allows you to capture errors. You can then check the error number using the ERROR_NUMBER() function to determine whether you have a deadlock. Once a deadlock is established, it's possible to try restarting the transaction a set number of times—two, in this case. Using error trapping will help your application deal with intermittent or occasional deadlocks, but the best approach is to analyze the cause of the deadlock and resolve it, if possible.

Deadlock Analysis

You can sometimes prevent a deadlock from happening by analyzing the causes. You need the following information to do this:

- The sessions participating in the deadlock
- The resources involved in the deadlock
- The queries executed by the sessions

Collecting Deadlock Information

You have four ways to collect the deadlock information.

- Use Extended Events.
- Set trace flag 1222.
- Set trace flag 1204.
- Use trace events.

Trace flags are used to customize certain SQL Server behavior such as, in this case, generating the deadlock information. But, they're an older way to capture this information. Within SQL Server, on every instance since 2008, there is an Extended Events session called system_health. This session runs automatically, and one of the events it gathers by default is the deadlock graph. This is the easiest way to get immediate access to deadlock information without having to modify your server in any way. The system_health session is also how you get deadlock information from an Azure SQL Database.

The system_health session writes to disk by default. The files are limited in size and number, so depending the activity on your system, you may find that the deadlock information is missing if the deadlock you're investigating occurred some time in the past. If you need to gather information for longer periods of time and ensure that you capture as many events as possible, Extended Events provides several ways to gather the deadlock information. This is probably the best method you can apply to your server for collecting deadlock information. You can use these options:

- lock_deadlock: Displays basic information about a deadlock occurrence
- lock_deadlock_chain: Captures information from each participant in a deadlock
- xml:deadlock_report: Displays an XML deadlock graph with the cause of the deadlock

The deadlock graph generates XML output. After Extended Events captures the deadlock event, you can view the deadlock graph within SSMS either by using the event viewer or by opening the XML file if you output your event results there. While similar information is displayed in all three events, for basic deadlock information, the easiest to understand is the xml:deadlock_report. When specifically monitoring for deadlocks, in a situation where you're attempting to deal with one in particular, I recommend also capturing the lock_deadlock_chain so that you have more detailed information about the individual sessions involved in the deadlock if you need it. For most situations, the deadlock graph should provide the information you need.

To retrieve the graph directly from the system_health session, you can query the output like this:

```
DECLARE @path NVARCHAR(260)
--to retrieve the local path of system_health files
SELECT @path = dosdlc.path
FROM sys.dm_os_server_diagnostics_log_configurations AS dosdlc;

SELECT @path = @path + N'system_health_*';

WITH fxd
AS (SELECT CAST(fx.event_data AS XML) AS Event_Data
    FROM sys.fn_xe_file_target_read_file(@path,
```

```
                                        NULL,
                                        NULL,
                                        NULL) AS fx )
SELECT dl.deadlockgraph
FROM
(   SELECT dl.query('.') AS deadlockgraph
    FROM fxd
        CROSS APPLY event_data.nodes('(/event/data/value/deadlock)') AS
d(dl) ) AS dl;
```

You can open the deadlock graph in Management Studio. You can search the XML, but the deadlock graph generated from the XML works almost like an execution plan for deadlocks, as shown in Figure 22-2.

Figure 22-2. *A deadlock graph as displayed in the Profiler*

I'll show you how to use this in the "Analyzing the Deadlock" section later in this chapter.

The two trace flags that generate deadlock information can be used individually or together to generate different sets of information. Usually people will prefer to run one or the other because they write a lot of information into the error log of SQL Server. The trace flags write the information gathered into the log file on the server where the deadlock event occurred. Trace flag 1222 provides the most detailed information on the deadlock.

Trace flag 1204 provides deadlock information that helps you analyze the cause of a deadlock. It sorts the information by each of the nodes involved in the deadlock. Trace flag 1222 provides detailed deadlock information, but it breaks the information down differently. Trace flag 1222 sorts the information by resource and processes, and it provides even more information. Both sets of data will be discussed in the "Analyzing the Deadlock" section.

The `DBCC TRACEON` statement is used to turn on (or enable) the trace flags. A trace flag remains enabled until it is disabled using the `DBCC TRACEOFF` statement. If the server is

restarted, this trace flag will be cleared. You can determine the status of a trace flag using the `DBCC TRACESTATUS` statement. Setting both of the deadlock trace flags looks like this:

```
DBCC TRACEON (1222, -1);
DBCC TRACEON (1204, -1);
```

To ensure that the trace flags are always set, it is possible to make them part of the SQL Server startup in the SQL Server Configuration Manager by following these steps:

1. Open the Properties dialog box of the instance of SQL Server.
2. Switch to the Startup Parameters tab of the Properties dialog box, as shown in Figure 22-3.

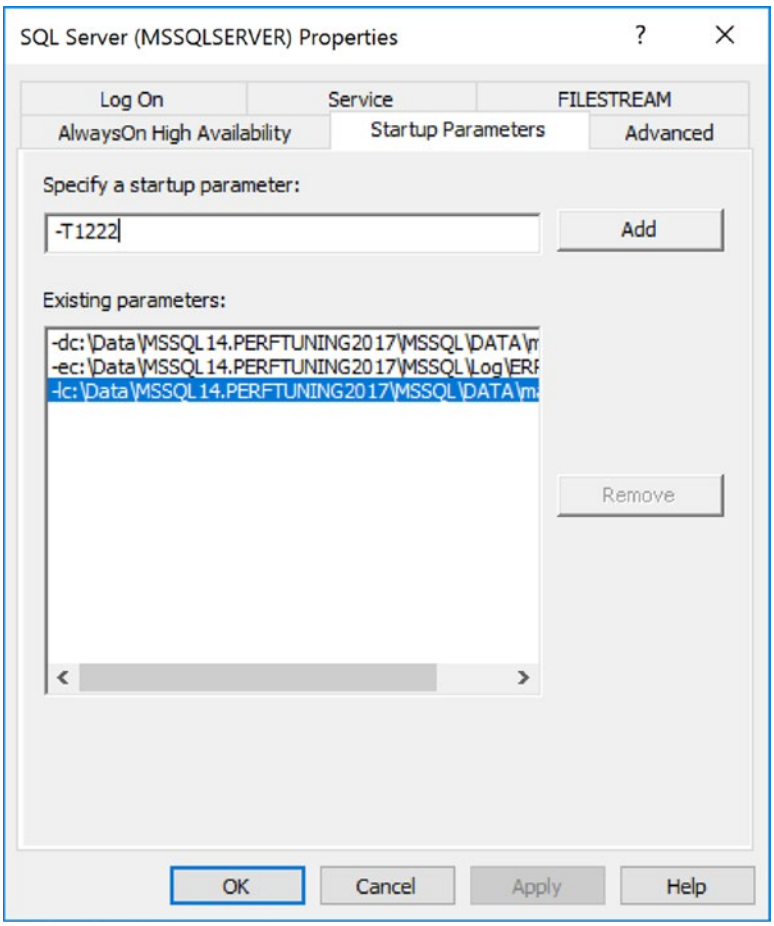

Figure 22-3. *A SQL Server instance's Properties dialog box showing the Startup Parameters tab*

3. Type **-T1222** in the "Specify a startup parameter" text box, and click Add to add trace flag 1222.
4. Click the OK button to close all the dialog boxes.

These trace flag settings will be in effect after you restart your SQL Server instance.

For most systems, using the `system_health` session is an easier and more efficient mechanism. It's installed and enabled by default. You don't have to do anything to get it running. The `system_health` session doesn't add noise to your servers error log, making it cleaner and easier to deal with as well. The trace flags are still available for use, and older systems may find they're necessary. However, more modern systems just won't need them.

Analyzing the Deadlock

To analyze the cause of a deadlock, let's consider a straightforward little example. I'm going to use the `system_health` session to show the deadlock information.

In one connection, execute this script:

```
BEGIN TRAN
UPDATE Purchasing.PurchaseOrderHeader
SET Freight = Freight * 0.9 -- 10% discount on shipping
WHERE PurchaseOrderID = 1255;
```

In a second connection, execute this script:

```
BEGIN TRANSACTION
UPDATE Purchasing.PurchaseOrderDetail
SET OrderQty = 4
WHERE ProductID = 448
      AND PurchaseOrderID = 1255;
```

Each of these scripts opens a transaction and manipulates data, but neither commits or rolls back the transaction. Switch back to the first transaction and run this additional query:

```
UPDATE  Purchasing.PurchaseOrderDetail
SET     OrderQty = 2
WHERE   ProductID = 448
        AND PurchaseOrderID = 1255;
```

Unfortunately, after possibly a few seconds, the first connection faces a deadlock.

```
Msg 1205, Level 13, State 51, Line 1
Transaction (Process ID 52) was deadlocked on lock resources with another
process and has been chosen as the deadlock victim. Rerun the transaction.
```

Any idea what's wrong here?

Let's analyze the deadlock by examining the deadlock graph collected through the trace event. There is a separate tab in the event explorer window for the `xml:deadlock_report` event. Opening that tab will show you the deadlock graph (see Figure 22-4).

Figure 22-4. *A deadlock graph displayed in the Profiler tool*

From the deadlock graph displayed in Figure 22-4, it's fairly clear that two processes were involved: session 53 on the left and session 63 on the right. Session 53, the one with the big *X* crossing it out (blue on the deadlock graph screen), was chosen as the deadlock victim. Two different keys were in question. The top key was owned by session 53, as indicated by the arrow pointing to the session object, named `Owner Mode`, and marked with an `X` for exclusive. Session 63 was attempting to request the same key for an update. The other key was owned by session 63, with session 53 requesting an update, indicated by the `U`. You can see the exact HoBt ID, object ID, object name, and index name for the objects in question for the deadlock. For a classic, simple deadlock like this, you have most of the information you need. The last piece would be the queries running from each process. These are available if you over the mouse over each session, as shown in Figure 22-5.

Figure 22-5. *The T-SQL statement for the deadlock victim*

The T-SQL statement for each side of the deadlock can be read in this manner so that you can focus exactly where the information is contained.

This visual representation of the deadlock can do the job. However, you may need to drill down into the underlying XML to examine some details of the deadlock, such as the isolation level of the processes involved. If you open that XML file directly from the extended event value, you can find a lot more information available than the simple set displayed for you in the graphical deadlock graph. Take a look at Figure 22-6.

```
<deadlock>
 <victim-list>
  <victimProcess id="process179b3f3b468" />
 </victim-list>
 <process-list>
  <process id="process179b3f3b468" taskpriority="0" logused="400" wa
   <executionStack>
    <frame procname="adhoc" line="1" stmtend="240" sqlhandle="0x0200
unknown    </frame>
    <frame procname="adhoc" line="1" stmtend="240" sqlhandle="0x0200
unknown    </frame>
   </executionStack>
   <inputbuf>
UPDATE  Purchasing.PurchaseOrderDetail
SET     OrderQty = 2
WHERE   ProductID = 448
        AND PurchaseOrderID = 1255;
   </inputbuf>
  </process>
  <process id="process179b7b63468" taskpriority="0" logused="9800" w
```

Figure 22-6. *The XML information that defines the deadlock graph*

If you look through this, you can see some of the information on display in the deadlock graph, but you also see a whole lot more. For example, part of this deadlock actually involves code that I did not write or execute as part of the example. There's a trigger on the table called `uPurchaseOrderDetail`. You can also see the code I used to generate the deadlock. All this information can help you identify exactly which pieces of code lead to the deadlock. You also get information such as the `sqlhandle`, which you can then use in combination with DMOs to pull statements and execution plans out of the cache or out of the Query Store. Because the plan is created before the query is run, it will be available for you even for the queries that were chosen as the deadlock victim.

It's worth taking some time to explore this XML in a little more detail. Table 22-1 shows some of the elements from the extended event and the information it represents.

Table 22-1. *XML Deadlock Graph Data*

Entry in Log	Description
`<deadlock>` `<victim-list>`	The beginning of the deadlock information. It starts laying out the victim processes.
`<victimProcess id="process179b3f3b468" />`	Physical memory address of the process picked to be the deadlock victim.
`<process-list>`	Processes that define the deadlock victim. There may be more than one.

`<process179b3f3b468" />`
`</victim-list>`
`<process-list>`
`<process id="process179b3f3b468" taskpriority="0" logused="400" waitresource="KEY: 6:72057594050904064 (4ab5f0d47ad5)" waittime="3703" ownerId="179351993" transactionname="user_transaction" lasttranstarted="2018-03-25T11:28:18.140" XDES="0x179b4bdc490" lockMode="U" schedulerid="1" kpid="2168" status="suspended" spid="53" sbid="0" ecid="0" priority="0" trancount="2" lastbatchstarted="2018-03-25T11:29:05.377" lastbatchcompleted="2018-03-25T11:29:05.363" lastattention="1900-01-01T00:00:00.363" clientapp="Microsoft SQL Server Management Studio - Query" hostname="WIN-8A2LQANSO51" hostpid="7028" loginname="WIN-8A2LQANSO51\Administrator"` **`isolationlevel="read committed (2)"`** `xactid="179351993" currentdb="6" lockTimeout="4294967295" clientoption1="671090784" clientoption2="390200">`
	All the information about the session picked as the deadlock victim. Note the highlighted isolation level, which is a key for helping identify the root cause of a deadlock.

(*continued*)

Table 22-1. (*continued*)

Entry in Log	Description
<executionStack> <frame procname="adhoc" line="1" stmtend="240" sqlhandle="0x02000000d0c7f31a30fb1ad425c34357fe8ef6326793e7aa0000000000000000000000000000000000000000000000"> unknown </frame> <frame procname="adhoc" line="1" stmtend="240" sqlhandle="0x02000000e7794d32ae3080d4a3217fdd3d1499f2e322d46e0000000000000000000000000000000000000000000000"> unknown </frame> </executionStack> <inputbuf> UPDATE Purchasing.PurchaseOrderDetail SET OrderQty = 2 WHERE ProductID = 448 AND PurchaseOrderID = 1255; </inputbuf> </process>	
<process id="process179b7b63468" taskpriority="0" logused="9800" waitresource="KEY: 6:72057594050969600 (4bc08edebc6b)" waittime="44833" ownerId="179352664" transactionname="user_transaction" lasttranstarted="2018-03-25T11:28:24.163" XDES="0x179bc2a8490" lockMode="U" schedulerid="1" kpid="3784" status="suspended" spid="63" sbid="0" ecid="0" priority="0" trancount="2" lastbatchstarted="2018-03-25T11:28:23.960" lastbatchcompleted="2018-03-25T11:28:23.920" lastattention="1900-01-01T00:00:00.920" clientapp="Microsoft SQL Server Management Studio - Query" hostname="WIN-8A2LQANSO51" hostpid="7028" loginname="WIN-8A2LQANSO51\Administrator" **isolationlevel="read committed (2)"** xactid="179352664" currentdb="6" lockTimeout="4294967295" clientoption1="673319008" clientoption2="390200">	The second process defined.

(*continued*)

Table 22-1. (*continued*)

Entry in Log	Description
<frame procname="AdventureWorks2017.Purchasing.uPurchaseOrderDetail" line="39" stmtstart="2732" stmtend="3830" **sqlhandle="0x0300060025999f1142d8ef0019a800000000000000000000000000000000000000000000000000000000000000000000"**> UPDATE [Purchasing].[PurchaseOrderHeader] SET [Purchasing].[PurchaseOrderHeader].[SubTotal] = (SELECT SUM([Purchasing].[PurchaseOrderDetail].[LineTotal]) FROM [Purchasing].[PurchaseOrderDetail] WHERE [Purchasing].[PurchaseOrderHeader].[PurchaseOrderID] = [Purchasing].[PurchaseOrderDetail].[PurchaseOrderID]) WHERE [Purchasing].[PurchaseOrderHeader].[PurchaseOrderID] IN (SELECT inserted.[PurchaseOrderID] FROM inserted </frame>	You can see that this is a trigger, referred to as a procname, uPurchaseOrderDetail. It has the sqlhandle, highlighted, so that you can retrieve it from the cache or the Query Store. It also shows the code of the trigger.
<frame procname="adhoc" line="2" stmtstart="38" stmtend="278" sqlhandle="0x02000000352f5b347ab7d87fc940e4f04e534f1c825a28b4000000000000000000000000000000000000000000000000">unknown </frame> </executionStack> <inputbuf> BEGIN TRANSACTION UPDATE Purchasing.PurchaseOrderDetail SET OrderQty = 4 WHERE ProductID = 448 AND PurchaseOrderID = 1255; </inputbuf>	The next statement in the batch and the code being called.

(*continued*)

Table 22-1. (*continued*)

Entry in Log	Description
<resource-list> <keylock hobtid="72057594050904064" dbid="6" objectname="AdventureWorks2017. Purchasing.PurchaseOrderDetail" indexname="PK_ PurchaseOrderDetail_PurchaseOrderID_ PurchaseOrderDetailID" id="lock17992a41a00" mode="X" associatedObjectId="72057594050904064"> <owner-list> <owner id="process179b7b63468" mode="X" /> </owner-list> <waiter-list> <waiter id="process179b3f3b468" mode="U" requestType="wait" /> </waiter-list> </keylock> **<keylock hobtid="72057594050969600" dbid="6" objectname="AdventureWorks2017. Purchasing.PurchaseOrderHeader" indexname="PK_PurchaseOrderHeader_ PurchaseOrderID" id="lock179b7a1a880" mode="X" associatedObjectId="72057594050969600">** <owner-list> <owner id="process179b3f3b468" mode="X" /> </owner-list> <waiter-list> <waiter id="process179b7b63468" mode="U" requestType="wait" /> </waiter-list> </keylock> </resource-list>	The objects that caused the conflict. Within this is the definition of the primary key from the `Purchasing.PurchaseOrderDetail` table. You can see which process from the earlier code owned which resource. You can also see the information defining the processes that were waiting. This is everything you need to discern where the issue exists.

This information is a bit more difficult to read through than the clean set of data provided by the graphical deadlock graph. However, it is a similar set of information, just more detailed. You can see, highlighted in bold near the bottom, the definition of one of the keys associated with the deadlock. You can also see, just before it, that the text of the execution plans is available through the Extended Events tool's XML output, just like the deadlock graph. You get everything you need to isolate the cause of the deadlock either way.

The information gathered by trace flag 1222 is almost identical to the XML data in every regard. The main differences are the formatting and location. The output from 1222 is located in the SQL Server error log, and it's in text format instead of nice, clean XML. The information collected by trace flag 1204 is completely different from either of the other two sets of data and doesn't provide nearly as much detail. Trace flag 1204 is also much more difficult to interpret. For all these reasons, I suggest you stick to using Extended Events if you can—or trace flag 1222 if you can't—to capture deadlock data. You also have the `system_health` session that captures a number of events by default, including deadlocks. It's a great resource if you are unprepared for capturing this information. Just remember that it keeps only four 5MB files online. As these fill, the data in the oldest file is lost. Depending on the number of transactions in your system and the number of deadlocks or other events that could fill these files, you may have only recent data available. Further, as mentioned earlier, since the `system_health` session uses the ring buffer to capture events, you can expect some event loss, so your deadlock events could go missing.

This example demonstrated a classic circular reference. Although not immediately obvious, the deadlock was caused by a trigger on the `Purchasing.PurchaseOrderDetail` table. When `Quantity` is updated on the `Purchasing.PurchaseOrderDetail` table, it attempts to update the `Purchasing.PurchaseOrderHeader` table. When the first two queries are run, each within an open transaction, it's just a blocking situation. The second query is waiting on the first to clear so that it can also update the `Purchasing.PurchaseOrderHeader` table. But when the third query (that is, the second within the first transaction) is introduced, a circular reference is created. The only way to resolve it is to kill one of the processes.

Before proceeding, be sure to roll back any open transactions.

Here's the obvious question at this stage: can you avoid this deadlock? If the answer is "yes," then how?

Avoiding Deadlocks

The methods for avoiding a deadlock scenario depend upon the nature of the deadlock. The following are some of the techniques you can use to avoid a deadlock:

- Access resources in the same physical order.
- Decrease the number of resources accessed.
- Minimize lock contention.
- Tune queries.

Accessing Resources in the Same Physical Order

One of the most commonly adopted techniques for avoiding a deadlock is to ensure that every transaction accesses the resources in the same physical order. For instance, suppose that two transactions need to access two resources. If each transaction accesses the resources in the same physical order, then the first transaction will successfully acquire locks on the resources without being blocked by the second transaction. The second transaction will be blocked by the first while trying to acquire a lock on the first resource. This will cause a typical blocking scenario without leading to a circular blocking and a deadlock.

If the resources are not accessed in the same physical order (as demonstrated in the earlier deadlock analysis example), this can cause a circular blocking between the two transactions.

- Transaction 1:
  - Access Resource 1
  - Access Resource 2
- Transaction 2:
  - Access Resource 2
  - Access Resource 1

In the current deadlock scenario, the following resources are involved in the deadlock:

- Resource 1, `hobtid=72057594046578688`: This is the index row within index `PK_ PurchaseOrderDetail_PurchaseOrderId_ PurchaseOrderDetailId` on the `Purchasing.PurchaseOrderDetail` table.
- Resource 2, `hobtid=72057594046644224`: This is the row within clustered index `PK_ PurchaseOrderHeader_PurchaseOrderId` on the `Purchasing.PurchaseOrderHeader` table.

Both sessions attempt to access the resource; unfortunately, the order in which they access the key is different.

It's common with some of the generated code produced by tools such as nHibernate and Entity Framework to see objects being referenced in a different order in different queries. You'll have to work with your development team to see that type of issue eliminated within the generated code.

Decreasing the Number of Resources Accessed

A deadlock involves at least two resources. A session holds the first resource and then requests the second resource. The other session holds the second resource and requests the first resource. If you can prevent the sessions (or at least one of them) from accessing one of the resources involved in the deadlock, then you can prevent the deadlock. You can achieve this by redesigning the application, which is a solution highly resisted by developers late in the project. However, you can consider using the following features of SQL Server without changing the application design:

- Convert a nonclustered index to a clustered index.
- Use a covering index for a `SELECT` statement.

Convert a Nonclustered Index to a Clustered Index

As you know, the leaf pages of a nonclustered index are separate from the data pages of the heap or the clustered index. Therefore, a nonclustered index takes two locks: one for the base (either the cluster or the heap) and one for the nonclustered index. However, in the case of a clustered index, the leaf pages of the index and the data pages of the table

are the same; it requires one lock, and that one lock protects both the clustered index and the table because the leaf pages and the data pages are the same. This decreases the number of resources to be accessed by the same query, compared to a nonclustered index. But, it is completely dependent on this being an appropriate clustered index. There's nothing magical about the clustered index that simply applying it to any column would help. You still need to assess whether it's appropriate.

Use a Covering Index for a SELECT Statement

You can also use a covering index to decrease the number of resources accessed by a `SELECT` statement. Since a `SELECT` statement can get everything from the covering index itself, it doesn't need to access the base table. Otherwise, the `SELECT` statement needs to access both the index and the base table to retrieve all the required column values. Using a covering index stops the `SELECT` statement from accessing the base table, leaving the base table free to be locked by another session.

Minimizing Lock Contention

You can also resolve a deadlock by avoiding the lock request on one of the contended resources. You can do this when the resource is accessed only for reading data. Modifying a resource will always acquire an exclusive (X) lock on the resource to maintain the consistency of the resource; therefore, in a deadlock situation, identify the resource accesses that are read-only and try to avoid their corresponding lock requests by using the dirty read feature, if possible. You can use the following techniques to avoid the lock request on a contended resource:

- Implement row versioning.
- Decrease the isolation level.
- Use locking hints.

Implement Row Versioning

Instead of attempting to prevent access to resources using a more stringent locking scheme, you could implement row versioning through the `READ_COMMITTED_SNAPSHOT` isolation level or through the `SNAPSHOT` isolation level. The row versioning isolation levels are used to reduce blocking, as outlined in Chapter 21. Because they reduce blocking,

which is the root cause of deadlocks, they can also help with deadlocks. By introducing READ_COMMITTED_SNAPSHOT with the following T-SQL, you can have a version of the rows available in tempdb, thus potentially eliminating the contention caused by the lock in the preceding deadlock scenario:

```
ALTER DATABASE AdventureWorks2017
SET READ_COMMITTED_SNAPSHOT ON;
```

This will allow any necessary reads without causing lock contention since the reads are on a different version of the data. There is overhead associated with row versioning, especially in tempdb and when marshaling data from multiple resources instead of just the table or indexes used in the query. But that trade-off of increased tempdb overhead versus the benefit of reduced deadlocking and increased concurrency may be worth the cost.

Decrease the Isolation Level

Sometimes the (S) lock requested by a SELECT statement contributes to the formation of circular blocking. You can avoid this type of circular blocking by reducing the isolation level of the transaction containing the SELECT statement to READ COMMITTED SNAPSHOT. This will allow the SELECT statement to read the data without requesting an (S) lock and thereby avoid the circular blocking. You may also see issues of this type around cursors because they tend to have pessimistic concurrency.

Also check to see whether the connections are setting themselves to be SERIALIZABLE. Sometimes online connection string generators will include this option, and developers will use it completely by accident. MSDTC will use serializable by default, but it can be changed.

Use Locking Hints

I absolutely do not recommend this approach. However, you can potentially resolve the deadlock presented in the preceding technique using the following locking hints:

- NOLOCK
- READUNCOMMITTED

Like the `READ UNCOMMITTED` isolation level, the `NOLOCK` or `READUNCOMMITTED` locking hint will avoid the (S) locks requested by a given session, thereby preventing the formation of circular blocking.

The effect of the locking hint is at a query level and is limited to the table (and its indexes) on which it is applied. The `NOLOCK` and `READUNCOMMITTED` locking hints are allowed only in `SELECT` statements and the data selection part of the `INSERT`, `DELETE`, and `UPDATE` statements.

The resolution techniques of minimizing lock contention introduce the side effect of a dirty read, which may not be acceptable in every transaction. A dirty read can involve missing rows or extra rows because of page splits and rearranging pages. Therefore, use these resolution techniques only in situations in which a low quality of data is acceptable.

Tune the Queries

At its root, deadlocking is about performance. If all the queries complete execution before resource contention is possible, then you can completely avoid the issue entirely.

Summary

As you learned in this chapter, a deadlock is the result of conflicting blocking between processes and is reported to an application with the error number 1205. You can analyze the cause of a deadlock by collecting the deadlock information using various resources, but the extended event `xml:deadlock_report` is probably the best.

You can use a number of techniques to avoid a deadlock; which technique is applicable depends upon the type of queries executed by the participating sessions, the locks held and requested on the involved resources, and the business rules governing the degree of isolation required. Generally, you can resolve a deadlock by reconfiguring the indexes and the transaction isolation levels. However, at times you may need to redesign the application or automatically reexecute the transaction on a deadlock. Just remember, at its core, deadlocks are a performance problem, and anything you can do to make the queries run faster will help to mitigate, if not eliminate, deadlocks in your queries.

In the next chapter, I cover the performance aspects of cursors and how to optimize the cost overhead of using cursors.

CHAPTER 23

Row-by-Row Processing

It is common to find database applications that use cursors to process one row at a time. Developers tend to think about processing data in a row-by-row fashion. Oracle even uses something called *cursors* as a high-speed data access mechanism. Cursors in SQL Server are different. Because data manipulation through a cursor in SQL Server incurs significant additional overhead, database applications should avoid using cursors. T-SQL and SQL Server are designed to work best with sets of data, not one row at a time. Jeff Moden famously termed this type of processing RBAR (pronounced, "ree-bar"), meaning row by agonizing row. However, if a cursor must be used, then use a cursor with the least cost.

In this chapter, I cover the following topics:

- The fundamentals of cursors
- A cost analysis of different characteristics of cursors
- The benefits and drawbacks of a default result set over cursors
- Recommendations to minimize the cost overhead of cursors

Cursor Fundamentals

When a query is executed by an application, SQL Server returns a set of data consisting of rows. Generally, applications can't process multiple rows together; instead, they process one row at a time by walking through the result set returned by SQL Server. This functionality is provided by a *cursor*, which is a mechanism to work with one row at a time out of a multirow result set.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_23

T-SQL cursor processing usually involves the following steps:

1. Declare the cursor to associate it with a SELECT statement and define the characteristics of the cursor.
2. Open the cursor to access the result set returned by the SELECT statement.
3. Retrieve a row from the cursor. Optionally, modify the row through the cursor.
4. Move to additional rows in the result set.
5. Once all the rows in the result set are processed, close the cursor and release the resources assigned to the cursor.

You can create cursors using T-SQL statements or the data access layers used to connect to SQL Server. Cursors created using data access layers are commonly referred to as *client* cursors. Cursors written in T-SQL are referred to as *server* cursors. The following is an example of a server cursor processing query results from a table:

```
--Associate a SELECT statement to a cursor and define the
--cursor's characteristics
USE AdventureWorks2017;
GO
SET NOCOUNT ON
DECLARE MyCursor CURSOR /*<cursor characteristics>*/
FOR
SELECT adt.AddressTypeID,
       adt.Name,
       adt.ModifiedDate
FROM Person.AddressType AS adt;

--Open the cursor to access the result set returned by the
--SELECT statement
OPEN MyCursor;

--Retrieve one row at a time from the result set returned by
--the SELECT statement
```

```
DECLARE @AddressTypeId INT,
        @Name VARCHAR(50),
        @ModifiedDate DATETIME;

FETCH NEXT FROM MyCursor
INTO @AddressTypeId,
     @Name,
     @ModifiedDate;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'NAME =   ' + @Name;

    --Optionally, modify the row through the cursor
    UPDATE Person.AddressType
    SET Name = Name + 'z'
    WHERE CURRENT OF MyCursor;

    --Move through to additional rows in the data set
    FETCH NEXT FROM MyCursor
    INTO @AddressTypeId,
         @Name,
         @ModifiedDate;
END

--Close the cursor and release all resources assigned to the
--cursor
CLOSE MyCursor;
DEALLOCATE MyCursor;
```

Part of the overhead of the cursor depends on the cursor characteristics. The characteristics of the cursors provided by SQL Server and the data access layers can be broadly classified into three categories.

- *Cursor location*: Defines the location of the cursor creation
- *Cursor concurrency*: Defines the degree of isolation and synchronization of a cursor with the underlying content
- *Cursor type*: Defines the specific characteristics of a cursor

Before looking at the costs of cursors, I'll take a few pages to introduce the various characteristics of cursors. You can undo the changes to the `Person.AddressType` table with this query:

```
UPDATE Person.AddressType
SET Name = LEFT(Name, LEN(Name) - 1);
```

Cursor Location

Based on the location of its creation, a cursor can be classified into one of two categories.

- Client-side cursors
- Server-side cursors

The T-SQL cursors are always created on SQL Server. However, the database API cursors can be created on either the client or server side.

Client-Side Cursors

As its name signifies, a *client-side cursor is* created on the machine running the application, whether the app is a service, a data access layer, or the front end for the user. It has the following characteristics:

- It is created on the client machine.
- The cursor metadata is maintained on the client machine.
- It is created using the data access layers.
- It works against most of the data access layers (OLEDB providers and ODBC drivers).
- It can be a forward-only or static cursor.

Note Cursor types, including forward-only and static cursor types, are described later in the chapter in the "Cursor Types" section.

Server-Side Cursors

A *server-side cursor* is created on the SQL Server machine. It has the following characteristics:

- It is created on the server machine.
- The cursor metadata is maintained on the server machine.
- It is created using either data access layers or T-SQL statements.
- A server-side cursor created using T-SQL statements is tightly integrated with SQL Server.
- It can be any type of cursor. (Cursor types are explained later in the chapter.)

Note The cost comparison between client-side and server-side cursors is covered later in the chapter in the "Cost Comparison on Cursor Type" section.

Cursor Concurrency

Depending on the required degree of isolation and synchronization with the underlying content, cursors can be classified into the following concurrency models:

- *Read-only*: A nonupdatable cursor
- *Optimistic*: An updatable cursor that uses the optimistic concurrency model (no locks retained on the underlying data rows)
- *Scroll locks*: An updatable cursor that holds a lock on any data row to be updated

Read-Only

A read-only cursor is nonupdatable; no locks are held on the base tables. While fetching a cursor row, whether an (S) lock will be acquired on the underlying row depends upon the isolation level of the connection and any locking hints used in the `SELECT` statement

for the cursor. However, once the row is fetched, by default the locks are released. The following T-SQL statement creates a read-only T-SQL cursor:

```
DECLARE MyCursor CURSOR READ_ONLY FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Using as minimal locking overhead as possible makes the read-only type of cursor faster and safer. Just remember that you cannot manipulate data through the read-only cursor, which is the sacrifice you make for improved performance.

Optimistic

The optimistic with values concurrency model makes a cursor updatable. No locks are held on the underlying data. The factors governing whether an (S) lock will be acquired on the underlying row are the same as for a read-only cursor.

The optimistic concurrency model uses row versioning to determine whether a row has been modified since it was read into the cursor, instead of locking the row while it is read into the cursor. Version-based optimistic concurrency requires a `ROWVERSION` column in the underlying user table on which the cursor is created. The `ROWVERSION` data type is a binary number that indicates the relative sequence of modifications on a row. Each time a row with a `ROWVERSION` column is modified, SQL Server stores the current value of the global `ROWVERSION` value, `@@DBTS`, in the `ROWVERSION` column; it then increments the `@@DBTS` value.

Before applying a modification through the optimistic cursor, SQL Server determines whether the current `ROWVERSION` column value for the row matches the `ROWVERSION` column value for the row when it was read into the cursor. The underlying row is modified only if the `ROWVERSION` values match, indicating that the row hasn't been modified by another user in the meantime. Otherwise, an error is raised. In case of an error, refresh the cursor with the updated data.

If the underlying table doesn't contain a `ROWVERSION` column, then the cursor defaults to value-based optimistic concurrency, which requires matching the current value of the row with the value when the row was read into the cursor. The version-based concurrency control is more efficient than the value-based concurrency control since it requires less processing to determine the modification of the underlying row. Therefore,

for the best performance of a cursor with the optimistic concurrency model, ensure that the underlying table has a ROWVERSION column.

The following T-SQL statement creates an optimistic T-SQL cursor:

```
DECLARE MyCursor CURSOR OPTIMISTIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

A cursor with scroll locks concurrency holds a (U) lock on the underlying row until another cursor row is fetched or the cursor is closed. This prevents other users from modifying the underlying row when the cursor fetches it. The scroll locks concurrency model makes the cursor updatable.

The following T-SQL statement creates a T-SQL cursor with the scroll locks concurrency model:

```
DECLARE MyCursor CURSOR SCROLL_LOCKS FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Since locks are held on a row being referenced (until another cursor row is fetched or the cursor is closed), it blocks all the other users trying to modify the row during that period. This hurts database concurrency but ensures that you won't get errors if you're modifying data through the cursor.

Cursor Types

Cursors can be classified into the following four types:

- Forward-only cursors
- Static cursors
- Keyset-driven cursors
- Dynamic cursors

Let's take a closer look at these four types in the sections that follow.

Forward-Only Cursors

These are the characteristics of forward-only cursors:

- They operate directly on the base tables.
- Rows from the underlying tables are usually not retrieved until the cursor rows are fetched using the cursor `FETCH` operation. However, the database API forward-only cursor type, with the following additional characteristics, retrieves all the rows from the underlying table first:
  - Client-side cursor location
  - Server-side cursor location and read-only cursor concurrency
- They support forward scrolling only (`FETCH NEXT`) through the cursor.
- They allow all changes (`INSERT`, `UPDATE`, and `DELETE`) through the cursor. Also, these cursors reflect all changes made to the underlying tables.

The forward-only characteristic is implemented differently by the database API cursors and the T-SQL cursor. The data access layers implement the forward-only cursor characteristic as one of the four previously listed cursor types. But the T-SQL cursor doesn't implement the forward-only cursor characteristic as a cursor type; rather, it implements it as a property that defines the scrollable behavior of the cursor. Thus, for a T-SQL cursor, the forward-only characteristic can be used to define the scrollable behavior of one of the remaining three cursor types.

The T-SQL syntax provides a specific cursor type option, `FAST_FORWARD`, to create a fast-forward-only cursor. The nickname for the `FAST_FORWARD` cursor is the *fire hose* because it is the fastest way to move data through a cursor and because all the information flows one way. However, don't be surprised when the "firehose" is still not as fast as traditional set-based operations. The following T-SQL statement creates a fast-forward-only T-SQL cursor:

```
DECLARE MyCursor CURSOR FAST_FORWARD FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

The FAST_FORWARD property specifies a forward-only, read-only cursor with performance optimizations enabled.

Static Cursors

These are the characteristics of static cursors:

- They create a snapshot of cursor results in the tempdb database when the cursor is opened. Thereafter, static cursors operate on the snapshot in the tempdb database.
- Data is retrieved from the underlying tables when the cursor is opened.
- Static cursors support all scrolling options: FETCH FIRST, FETCH NEXT, FETCH PRIOR, FETCH LAST, FETCH ABSOLUTE n, and FETCH RELATIVE n.
- Static cursors are always read-only; data modifications are not allowed through static cursors. Also, changes (INSERT, UPDATE, and DELETE) made to the underlying tables are not reflected in the cursor.

The following T-SQL statement creates a static T-SQL cursor:

```
DECLARE MyCursor CURSOR STATIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Some tests show that a static cursor can perform as well as—and sometimes faster than—a forward-only cursor. Be sure to test this behavior on your own system in situations where you must use a cursor.

Keyset-Driven Cursors

These are the characteristics of keyset-driven cursors:

- Keyset cursors are controlled by a set of unique identifiers (or keys) known as a *keyset*. The keyset is built from a set of columns that uniquely identify the rows in the result set.
- These cursors create the keyset of rows in the tempdb database when the cursor is opened.

- Membership of rows in the cursor is limited to the keyset of rows created in the tempdb database when the cursor is opened.
- On fetching a cursor row, the database engine first looks at the keyset of rows in tempdb and then navigates to the corresponding data row in the underlying tables to retrieve the remaining columns.
- They support all scrolling options.
- Keyset cursors allow all changes through the cursor. An `INSERT` performed outside the cursor is not reflected in the cursor since the membership of rows in the cursor is limited to the keyset of rows created in the tempdb database on opening the cursor. An `INSERT` through the cursor appears at the end of the cursor. A `DELETE` performed on the underlying tables raises an error when the cursor navigation reaches the deleted row. An `UPDATE` on the nonkeyset columns of the underlying tables is reflected in the cursor. An `UPDATE` on the keyset columns is treated like a `DELETE` of an old key value and the `INSERT` of a new key value. If a change disqualifies a row for membership or affects the order of a row, then the row does not disappear or move unless the cursor is closed and reopened.

The following T-SQL statement creates a keyset-driven T-SQL cursor:

```
DECLARE MyCursor CURSOR KEYSET FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Dynamic Cursors

These are the characteristics of dynamic cursors:

- Dynamic cursors operate directly on the base tables.
- The membership of rows in the cursor is not fixed since they operate directly on the base tables.
- As with forward-only cursors, rows from the underlying tables are not retrieved until the cursor rows are fetched using a cursor `FETCH` operation.

- Dynamic cursors support all scrolling options except `FETCH ABSOLUTE n`, since the membership of rows in the cursor is not fixed.
- These cursors allow all changes through the cursor. Also, all changes made to the underlying tables are reflected in the cursor.
- Dynamic cursors don't support all properties and methods implemented by the database API cursors. Properties such as `AbsolutePosition`, `Bookmark`, and `RecordCount`, as well as methods such as `clone` and `Resync`, are not supported by dynamic cursors. Instead, they are supported by keyset-driven cursors.

The following T-SQL statement creates a dynamic T-SQL cursor:

```
DECLARE MyCursor CURSOR DYNAMIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

The dynamic cursor is absolutely the slowest possible cursor to use in all situations. It takes more locks and holds them longer, which radically increases its poor performance. Take this into account when designing your system.

Cursor Cost Comparison

Now that you've seen the different cursor flavors, let's look at their costs. If you must use a cursor, you should always use the lightest-weight cursor that meets the requirements of your application. The cost comparisons among the different characteristics of the cursors are detailed next.

Cost Comparison on Cursor Location

The client-side and server-side cursors have their own cost benefits and overhead, as explained in the sections that follow.

Client-Side Cursors

Client-side cursors have the following cost benefits compared to server-side cursors:

- *Higher scalability*: Since the cursor metadata is maintained on the individual client machines connected to the server, the overhead of maintaining the cursor metadata is taken up by the client machines. Consequently, the ability to serve a larger number of users is not limited by the server resources.
- *Fewer network round-trips*: Since the result set returned by the `SELECT` statement is passed to the client where the cursor is maintained, extra network round-trips to the server are not required while retrieving rows from the cursor.
- *Faster scrolling*: Since the cursor is maintained locally on the client machine, it's potentially faster to walk through the rows of the cursor.
- *Highly portable*: Since the cursor is implemented using data access layers, it works across a large range of databases: SQL Server, Oracle, Sybase, and so forth.

Client-side cursors have the following cost overhead or drawbacks:

- *Higher pressure on client resources*: Since the cursor is managed at the client side, it increases pressure on the client resources. But it may not be all that bad, considering that most of the time the client applications are web applications and scaling out web applications (or web servers) is quite easy using standard load-balancing solutions. On the other hand, scaling out a transactional SQL Server database is still an art!
- *Support for limited cursor types*: Dynamic and keyset-driven cursors are not supported.

- *Only one active cursor-based statement on one connection*: As many rows of the result set as the client network can buffer are arranged in the form of network packets and sent to the client application. Therefore, until all the cursor's rows are fetched by the application, the database connection remains busy, pushing the rows to the client. During this period, other cursor-based statements cannot use the connection. This is negated by taking advantage of multiple active result sets (MARS), which would allow a connection to have a second active cursor.

Server-Side Cursors

Server-side cursors have the following cost benefits:

- *Multiple active cursor-based statements on one connection*: While using server-side cursors, no results are left outstanding on the connection between the cursor operations. This frees the connection, allowing the use of multiple cursor-based statements on one connection at the same time. In the case of client-side cursors, as explained previously, the connection remains busy until all the cursor rows are fetched by the application. This means they cannot be used simultaneously by multiple cursor-based statements.
- *Row processing near the data*: If the row processing involves joining with other tables and a considerable amount of set operations, then it is advantageous to perform the row processing near the data using a server-side cursor.
- *Less pressure on client resources*: It reduces pressure on the client resources. But this may not be that desirable because, if the server resources are maxed out (instead of the client resources), then it will require scaling out the database, which is a difficult proposition.
- *Support for all cursor types*: Client-side cursors have limitations on which types of cursors can be supported. There are no limits on the server-side cursors.

Server-side cursors have the following cost overhead or disadvantages:

- *Lower scalability*: They make the server less scalable since server resources are consumed to manage the cursor.
- *More network round-trips*: They increase network round-trips if the cursor row processing is done in the client application. The number of network round-trips can be optimized by processing the cursor rows in the stored procedure or by using the cache size feature of the data access layer.
- *Less portable*: Server-side cursors implemented using T-SQL cursors are not readily portable to other databases because the syntax of the database code managing the cursor is different across databases.

Cost Comparison on Cursor Concurrency

As expected, cursors with a higher concurrency model create the least amount of blocking in the database and support higher scalability, as explained in the following sections.

Read-Only

The read-only concurrency model provides the following cost benefits:

- *Lowest locking overhead*: The read-only concurrency model introduces the least locking and synchronization overhead on the database. Since (S) locks are not held on the underlying row after a cursor row is fetched, other users are not blocked from accessing the row. Furthermore, the (S) lock acquired on the underlying row while fetching the cursor row can be avoided by using the `NO_LOCK` locking hint in the `SELECT` statement of the cursor, but only if you don't care about what kind of data you get back because of dirty reads.
- *Highest concurrency*: Since additional locks are not held on the underlying rows, the read-only cursor doesn't block other users from accessing the underlying tables. The shared lock is still acquired.

The main drawback of the read-only cursor is as follows:

- *Nonupdatable*: The content of underlying tables cannot be modified through the cursor.

Optimistic

The optimistic concurrency model provides the following benefits:

- *Low locking overhead*: Similar to the read-only model, the optimistic concurrency model doesn't hold an (S) lock on the cursor row after the row is fetched. To further improve concurrency, the `NOLOCK` locking hint can also be used, as in the case of the read-only concurrency model. But, please know that `NOLOCK` can absolutely lead to incorrect data or missing or extra rows, so its use requires careful planning. Modification through the cursor to an underlying row requires exclusive rights on the row as required by an action query.
- *High concurrency*: Since only a shared lock is used on the underlying rows, the cursor doesn't block other users from accessing the underlying tables. But the modification through the cursor to an underlying row will block other users from accessing the row during the modification.

The following examples detail the cost overhead of the optimistic concurrency model:

- *Row versioning*: Since the optimistic concurrency model allows the cursor to be updatable, an additional cost is incurred to ensure that the current underlying row is first compared (using either version-based or value-based concurrency control) with the original cursor row fetched before applying a modification through the cursor. This prevents the modification through the cursor from accidentally overwriting the modification made by another user after the cursor row is fetched.

- *Concurrency control without a ROWVERSION column*: As explained previously, a `ROWVERSION` column in the underlying table allows the cursor to perform an efficient version-based concurrency control. In case the underlying table doesn't contain a `ROWVERSION` column, the cursor resorts to value-based concurrency control, which requires matching the current value of the row to the value when the row was read into the cursor. This increases the cost of the concurrency control. Both forms of concurrency control will cause additional overhead in tempdb.

Scroll Locks

The major benefit of the scroll locks concurrency model is as follows:

- *Simple concurrency control*: By locking the underlying row corresponding to the last fetched row from the cursor, the cursor assures that the underlying row can't be modified by another user. This eliminates the versioning overhead of optimistic locking. Also, since the row cannot be modified by another user, the application is relieved from checking for a row-mismatch error.

The scroll locks concurrency model incurs the following cost overhead:

- *Highest locking overhead*: The scroll locks concurrency model introduces a pessimistic locking characteristic. A (U) lock is held on the last cursor row fetched, until another cursor row is fetched or the cursor is closed.
- *Lowest concurrency*: Since a (U) lock is held on the underlying row, all other users requesting a (U) or an (X) lock on the underlying row will be blocked. This can significantly hurt concurrency. Therefore, please avoid using this cursor concurrency model unless absolutely necessary.

Cost Comparison on Cursor Type

Each of the basic four cursor types mentioned in the "Cursor Fundamentals" section earlier in the chapter incurs a different cost overhead on the server. Choosing an incorrect cursor type can hurt database performance. Besides the four basic cursor types, a fast-forward-only cursor (a variation of the forward-only cursor) is provided to enhance performance. The cost overhead of these cursor types is explained in the sections that follow.

Forward-Only Cursors

These are the cost benefits of forward-only cursors:

- *Lower cursor open cost than static and keyset-driven cursors*: Since the cursor rows are not retrieved from the underlying tables and are not copied into the tempdb database during cursor open, the forward-only T-SQL cursor opens quickly. Similarly, the forward-only, server-side API cursors with optimistic/scroll locks concurrency open quickly since they do not retrieve the rows during cursor open.
- *Lower scroll overhead*: Since only `FETCH NEXT` can be performed on this cursor type, it requires less overhead to support different scroll operations.
- *Lower impact on the tempdb database than static and keyset-driven cursors*: Since the forward-only T-SQL cursor doesn't copy the rows from the underlying tables into the tempdb database, no additional pressure is created on the database.

The forward-only cursor type has the following drawbacks:

- *Lower concurrency*: Every time a cursor row is fetched, the corresponding underlying row is accessed with a lock request depending on the cursor concurrency model (as noted earlier in the discussion about concurrency). It can block other users from accessing the resource.
- *No backward scrolling*: Applications requiring two-way scrolling can't use this cursor type. But if the applications are designed properly, then it isn't difficult to live without backward scrolling.

Fast-Forward-Only Cursor

The fast-forward-only cursor is the fastest and least expensive cursor type. This forward-only and read-only cursor is specially optimized for performance. Because of this, you should always prefer it to the other SQL Server cursor types.

Furthermore, the data access layer provides a fast-forward-only cursor on the client side. That type of cursor uses a so-called *default result set* to make cursor overhead almost disappear.

Note The default result set is explained later in the chapter in the "Default Result Set" section.

Static Cursors

These are the cost benefits of static cursors:

- *Lower fetch cost than other cursor types*: Since a snapshot is created in the tempdb database from the underlying rows on opening the cursor, the cursor row fetch is targeted to the snapshot instead of the underlying rows. This avoids the lock overhead that would otherwise be required to fetch the cursor rows.
- *No blocking on underlying rows*: Since the snapshot is created in the tempdb database, other users trying to access the underlying rows are not blocked.

On the downside, the static cursor has the following cost overhead:

- *Higher open cost than other cursor types*: The cursor open operation of the static cursor is slower than that of other cursor types since all the rows of the result set have to be retrieved from the underlying tables and the snapshot has to be created in the tempdb database during the cursor open.
- *Higher impact on tempdb than other cursor types*: There can be significant impact on server resources for creating, populating, and cleaning up the snapshot in the tempdb database.

Keyset-Driven Cursors

These are the cost benefits of keyset-driven cursors:

- *Lower open cost than the static cursor*: Since only the keyset, not the complete snapshot, is created in the tempdb database, the keyset-driven cursor opens faster than the static cursor. SQL Server populates the keyset of a large keyset-driven cursor asynchronously, which shortens the time between when the cursor is opened and when the first cursor row is fetched.
- *Lower impact on tempdb than that with the static cursor*: Because the keyset-driven cursor is smaller, it uses less space in tempdb.

The cost overhead of keyset-driven cursors is as follows:

- *Higher open cost than forward-only and dynamic cursors*: Populating the keyset in the tempdb database makes the cursor open operation of the keyset-driven cursor costlier than that of forward-only (with the exceptions mentioned earlier) and dynamic cursors.
- *Higher fetch cost than other cursor types*: For every cursor row fetch, the key in the keyset has to be accessed first, and then the corresponding underlying row in the user database can be accessed. Accessing both the tempdb and the user database for every cursor row fetch makes the fetch operation costlier than that of other cursor types.
- *Higher impact on tempdb than forward-only and dynamic cursors*: Creating, populating, and cleaning up the keyset in tempdb impacts server resources.
- *Higher lock overhead and blocking than the static cursor*: Since row fetch from the cursor retrieves rows from the underlying table, it acquires an (S) lock on the underlying row (unless the `NOLOCK` locking hint is used) during the row fetch operation.

Dynamic Cursor

The dynamic cursor has the following cost benefits:

- *Lower open cost than static and keyset-driven cursors*: Since the cursor is opened directly on the underlying rows without copying anything to the tempdb database, the dynamic cursor opens faster than the static and keyset-driven cursors.
- *Lower impact on tempdb than static and keyset-driven cursors*: Since nothing is copied into tempdb, the dynamic cursor places far less strain on tempdb than the other cursor types.

The dynamic cursor has the following cost overhead:

- *Higher lock overhead and blocking than the static cursor*: Every cursor row fetch in a dynamic cursor requeries the underlying tables involved in the `SELECT` statement of the cursor. The dynamic fetches are generally expensive because the original select condition might have to be reexecuted.

For a summary of the different cursors, their positives and negatives, please refer to Table 23-1.

Table 23-1. *Comparing Cursors*

Cursor Type	Positives	Negatives
Forward-only	Lower cost, lower scroll overhead, lower impact on tempdb	Lower concurrency, no backward scrolling
Fast-forward-only	Fastest cursor, lowest cost, lowest impact	No backward scrolling, no concurrency
Static	Lower fetch cost, no blocking, forward and backward scrolling	Higher open cost, higher impact on tempdb, no concurrency
Keyset-driven	Lower open cost, lower impact on tempdb, forward and backward scrolling, concurrency	Higher open cost, highest fetch cost, highest impact on tempdb, higher locking costs
Dynamic	Lower open cost, lower impact on tempdb, forward and backward scrolling, concurrency	Highest locking costs

Default Result Set

The default cursor type for the data access layers (ADO, OLEDB, and ODBC) is forward-only and read-only. The default cursor type created by the data access layers isn't a true cursor but a stream of data from the server to the client, generally referred to as the *default result set* or *fast-forward-only cursor* (created by the data access layer). In ADO.NET, the `DataReader` control has the forward-only and read-only properties, and it can be considered as the default result set in the ADO.NET environment. SQL Server uses this type of result set processing under the following conditions:

- The application, using the data access layers (ADO, OLEDB, ODBC), leaves all the cursor characteristics at the default settings, which requests a forward-only and read-only cursor.
- The application executes a `SELECT` statement instead of executing a `DECLARE CURSOR` statement.

Note Because SQL Server is designed to work with sets of data, not to walk through records one by one, the default result set is always faster than any other type of cursor.

The only request sent from the client to SQL Server is the SQL statement associated with the default cursor. SQL Server executes the query, organizes the rows of the result set in network packets (filling the packets as best it can), and then sends the packets to the client. These network packets are cached in the network buffers of the client. SQL Server sends as many rows of the result set to the client as the client-network buffers can cache. As the client application requests one row at a time, the data access layer on the client machine pulls the row from the client-network buffers and transfers it to the client application.

The following sections outline the benefits and drawbacks of the default result set.

Benefits

The default result set is generally the best and most efficient way of returning rows from SQL Server for the following reasons:

- *Minimum network round-trips between the client and SQL Server*: Since the result set returned by SQL Server is cached in the client-network buffers, the client doesn't have to make a request across the network to get the individual rows. SQL Server puts most of the rows that it can in the network buffer and sends to the client as much as the client-network buffer can cache.
- *Minimum server overhead*: Since SQL Server doesn't have to store data on the server, this reduces server resource utilization.

Multiple Active Result Sets

SQL Server 2005 introduced the concept of multiple active result sets, wherein a single connection can have more than one batch running at any given moment. In prior versions, a single result set had to be processed or closed out prior to submitting the next request. MARS allows multiple requests to be submitted at the same time through the same connection. MARS is enabled on SQL Server all the time. It is not enabled by a connection unless that connection explicitly calls for it. Transactions must be handled at the client level and have to be explicitly declared and committed or rolled back. With MARS in action, if a transaction is not committed on a given statement and the connection is closed, all other transactions that were part of that single connection will be rolled back. MARS is enabled through application connection properties.

Drawbacks

While there are advantages to the default result set, there are drawbacks as well. Using the default result set requires some special conditions for maximum performance:

- *It doesn't support all properties and methods*: Properties such as `AbsolutePosition`, `Bookmark`, and `RecordCount`, as well as methods such as `Clone`, `MoveLast`, `MovePrevious`, and `Resync`, are not supported.
- *Locks may be held on the underlying resource*: SQL Server sends as many rows of the result set to the client as the client-network buffers can cache. If the size of the result set is large, then the client-network buffers may not be able to receive all the rows. SQL Server then holds a lock on the next page of the underlying tables, which has not been sent to the client.

To demonstrate these concepts, consider the following test table:

```
USE AdventureWorks2017;
GO
DROP TABLE IF EXISTS dbo.Test1;
GO

CREATE TABLE dbo.Test1 (C1 INT,
                        C2 CHAR(996));

CREATE CLUSTERED INDEX Test1Index ON dbo.Test1 (C1);

INSERT INTO dbo.Test1
VALUES (1, '1'),
       (2, '2');
GO
```

Now consider this PowerShell script, which accesses the rows of the test table using ADO with OLEDB and the default cursor type for the database API cursor (`ADODB.Recordset` object) as follows:

```
$AdoConn = New-Object -comobject ADODB.Connection
$AdoRecordset = New-Object -comobject ADODB.Recordset
```

```
##Change the Data Source to your server
$AdoConn.Open("Provider= SQLOLEDB; Data Source=DOJO\RANDORI; Initial
Catalog=AdventureWorks2017; Integrated Security=SSPI")
$AdoRecordset.Open("SELECT * FROM dbo.Test1", $AdoConn)

do {
    $C1 = $AdoRecordset.Fields.Item("C1").Value
    $C2 = $AdoRecordset.Fields.Item("C2").Value
    Write-Output "C1 = $C1 and C2 = $C2"
    $AdoRecordset.MoveNext()
    } until     ($AdoRecordset.EOF -eq $True)
$AdoRecordset.Close()
$AdoConn.Close()
```

This is not how you normally access databases from PowerShell, but it does show how a client-side cursor operates. Note that the table has two rows with the size of each row equal to 1,000 bytes (= 4 bytes for `INT` + 996 bytes for `CHAR(996)`) without considering the internal overhead. Therefore, the size of the complete result set returned by the `SELECT` statement is approximately 2,000 bytes (= 2 × 1,000 bytes).

On execution of the cursor open statement (`$AdoRecordset.Open()`), a default result set is created on the client machine running the code. The default result set holds as many rows as the client-network buffer can cache.

Since the size of the result set is small enough to be cached by the client-network buffer, all the cursor rows are cached on the client machine during the cursor open statement itself, without retaining any lock on the `dbo.Test1` table. You can verify the lock status for the connection using the `sys.dm_tran_locks` dynamic management view. During the complete cursor operation, the only request from the client to SQL Server is the `SELECT` statement associated to the cursor, as shown in the Extended Events output in Figure 23-1.

	name	batch_text	duration	logical_reads	row_count
	sql_batch_completed	SELECT * FROM dbo.Test1	83	2	2

Figure 23-1. *Profiler trace output showing database requests made by the default result set*

To find out the effect of a large result set on the default result set processing, let's add some more rows to the test table.

```
SELECT TOP 100000
       IDENTITY(INT, 1, 1) AS n
INTO #Tally
FROM master.dbo.syscolumns AS scl,
     master.dbo.syscolumns AS sc2;

INSERT INTO dbo.Test1 (C1,
                       C2)
SELECT n,
       n
FROM #Tally AS t;
GO
```

The additional rows generated by this example increase the size of the result set considerably. Depending on the size of the client-network buffer, only part of the result set can be cached. On execution of the `Ado.Recordset.Open` statement, the default result set on the client machine will get part of the result set, with SQL Server waiting on the other end of the network to send the remaining rows.

On my machine during this period, the locks shown in Figure 23-2 are held on the underlying `Test1` table as obtained from the output of `sys.dm_tran_locks`.

	request_session_id	resource_database_id	resource_associated_entity_id	resource_type	resource_description	request_mode	request_status
7	55	6	72057594081116160	PAGE	1:34491	IS	GRANT
8	55	6	320720195	OBJECT		IS	GRANT

Figure 23-2. *sys.dm_tran_locks output showing the locks held by the default result set while processing the large result set*

The (IS) lock on the table will block other users trying to acquire an (X) lock. To minimize the blocking issue, follow these recommendations:

- Process all rows of the default result set immediately.
- Keep the result set small. As demonstrated in the example, if the size of the result set is small, then the default result set will be able to read all the rows during the cursor open operation itself.