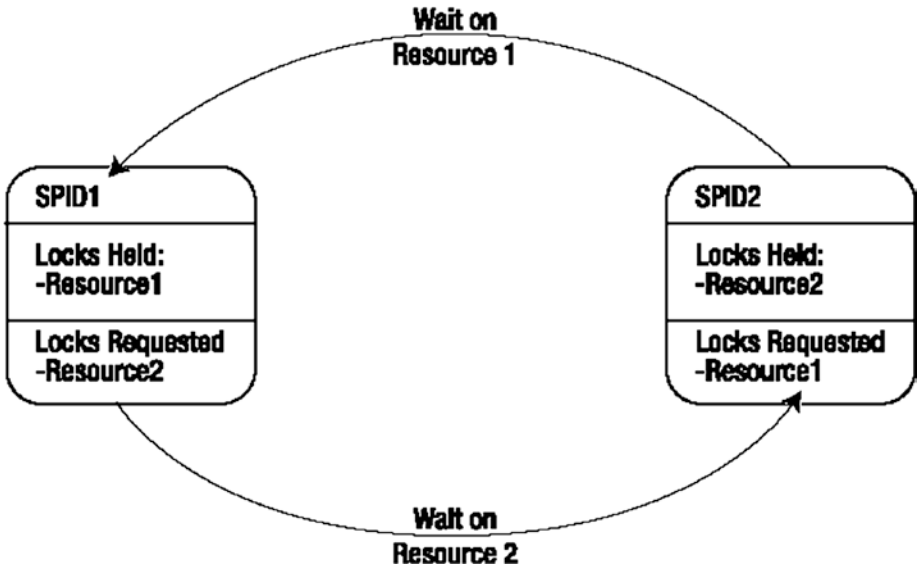

Figure 22-1. *A deadlock scenario*

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.

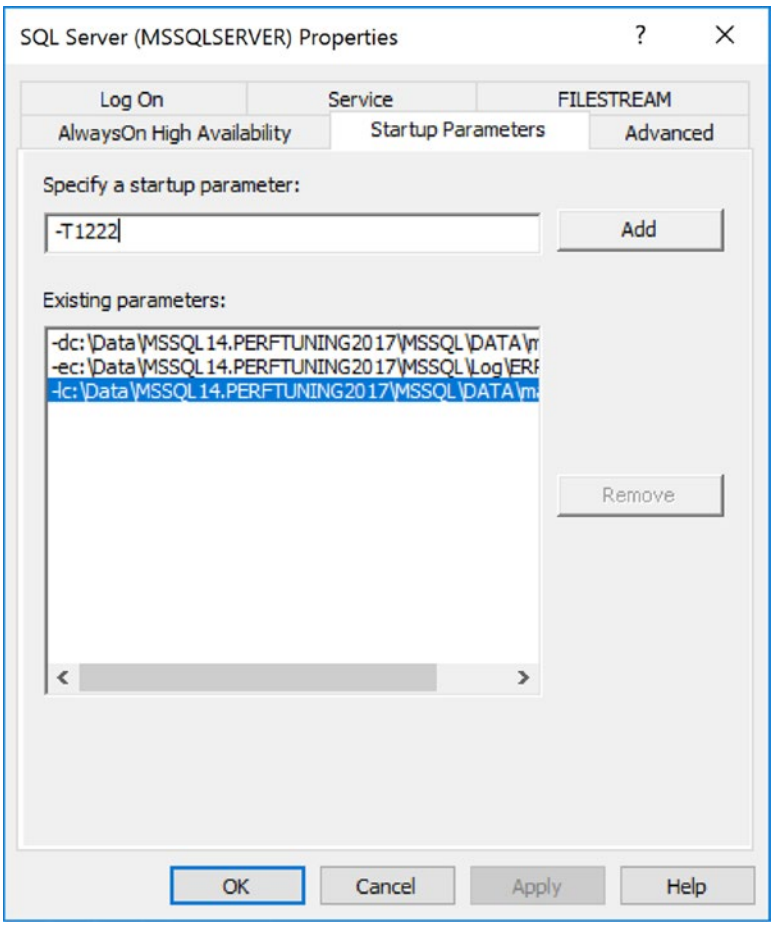

Figure 22-3. *A SQL Server instance's Properties dialog box showing the Startup Parameters tab*

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.