Table 6-3. *SQL Trace Filters*

Events	Filter Criteria Example	Use
`sqlserver.username`	= <some value>	This captures events only for a single user or login.
`sqlserver.database_id`	= <ID of the database to monitor>	This filters out events generated by other databases. You can determine the ID of a database from its name as follows: `SELECT DB_ID('AdventureWorks2012')`.
`duration`	>= 200	For performance analysis, you will often capture a trace for a large workload. In a large trace, there will be many event logs with a duration that is less than what you're interested in. Filter out these event logs because there is hardly any scope for optimizing these SQL activities.
`physical_reads`	>= 2	This is similar to the criterion on the duration filter.
`sqlserver.session_id`	= <Database users to monitor>	This troubleshoots queries sent by a specific server session.

Figure 6-4 shows a snippet of the preceding filter criteria selection in the Session window.

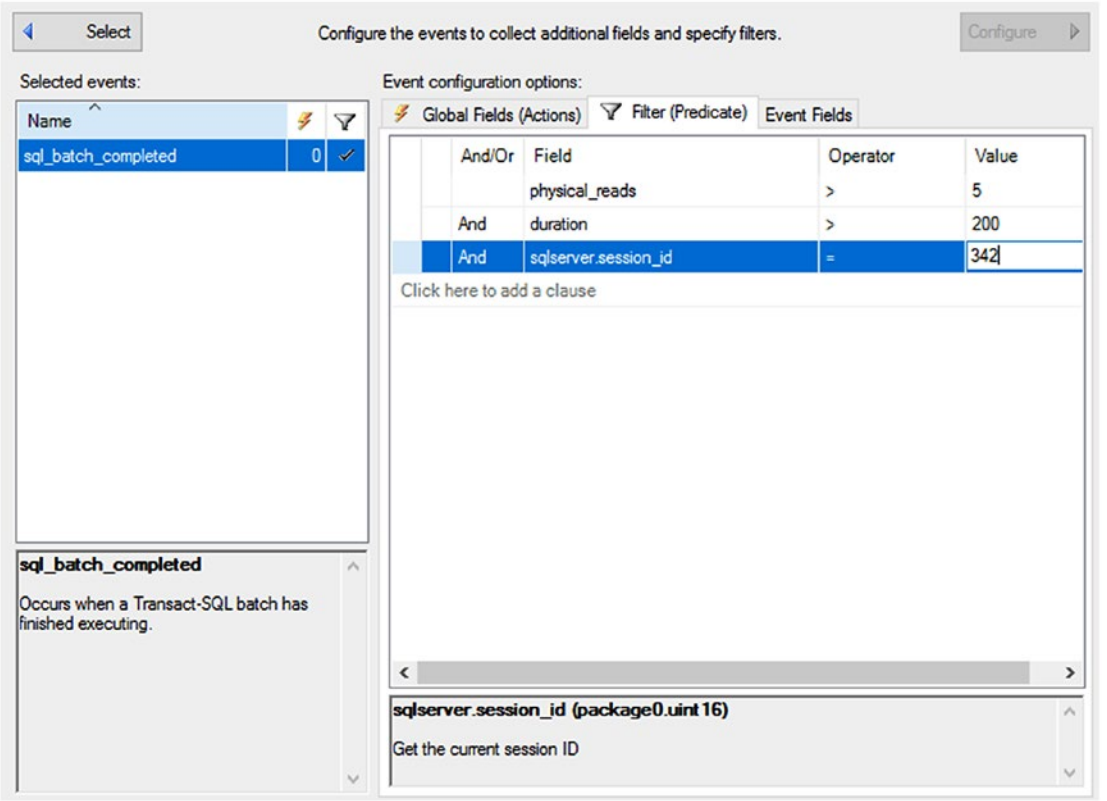

Figure 6-4. *Filters applied in the Session window*

If you look at the Field value in Figure 6-4, you'll note that it says `sqlserver.session_id`. This is because different sets of data are available to you, and they are qualified by the type of data being referenced. In this case, I'm talking specifically about a `sqlserver.session_id`. But I could be referring to something from SQL OS or even the Extended Events package itself.

Event Fields

The standard event fields are included automatically with the event type. Table 6-4 shows some of the common actions that you use for performance analysis.

Table 6-4. *Actions Commands for Query Analysis*

Data Column	Description
`Statement`	The SQL text from the `rpc_completed` event.
`Batch_text`	The SQL text from the `sql_batch_completed` event.
`cpu_time`	The CPU cost of an event in microseconds (mc). For example, CPU = 100 for a `SELECT` statement indicates that the statement took 100mc to execute.
`logical_reads`	The number of logical reads performed for an event. For example, logical_reads = 800 for a `SELECT` statement indicates that the statement required a total of 800 page reads.
`Physical_reads`	The number of physical reads performed for an event. This can differ from the `logical_reads` value because of access to the disk subsystem.
`writes`	The number of logical writes performed for an event.
`duration`	The execution time of an event in ms.

Each logical read and write consists of an 8KB page activity in memory, which may require zero or more physical I/O operations. You can see the fields for any given event by clicking the Event Fields tab on display in Figure 6-5.

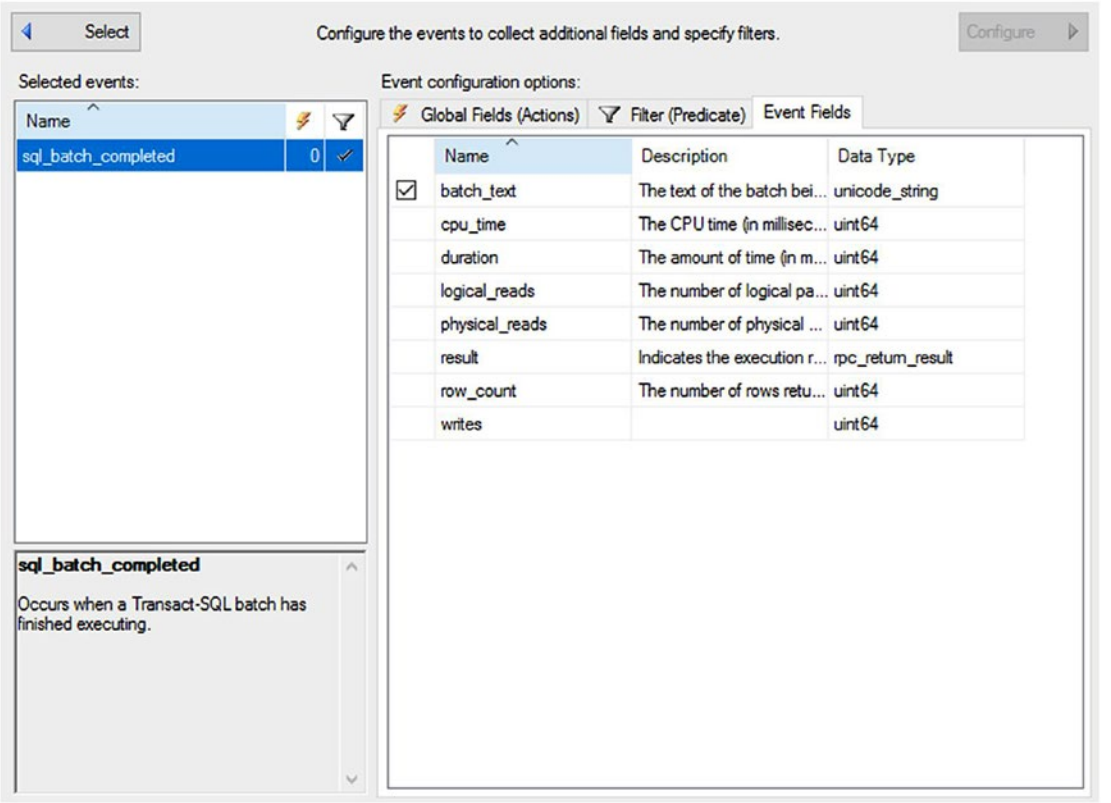

Figure 6-5. *New Session window with the Event Fields tab in Configure on display*

Some of the event fields are optional, but most of them are automatically included with the event. You can decide whether you want to include the optional fields. In Figure 6-5 you could exclude the `batch_text` field by clicking the check box next to it.

Data Storage

The next page in the new Session window, Data Storage in the "Select a page" pane, is for determining how you're going to deal with the data generated by the session. The output mechanism is referred to as the *target*. You have two basic choices: output the information to a file or simply use the buffer to capture the events. There are seven different types of output, but most of them are out of scope for the book. For the purposes of collecting performance information, you're going to use either `event_file` or `ring_buffer`. You should use only small data sets with the buffer because it will consume memory. Because it works with memory within the system, the buffer is built

so that, rather than overwhelm the system memory, it will drop events, so you're more likely to lose information using the buffer. In most circumstances for monitoring query performance, you should capture the output of the session to a file.

You have to select your target, as shown in Figure 6-6.

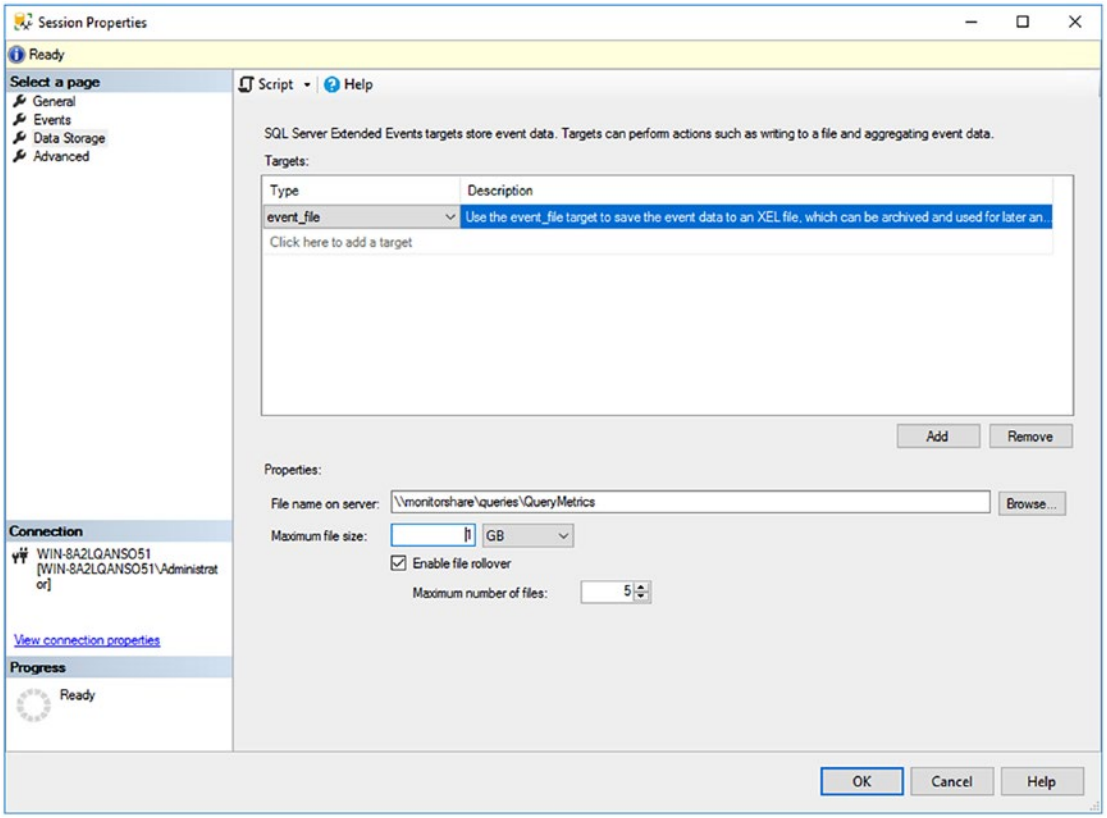

Figure 6-6. *Data Storage window in the New Session window*

You should specify an appropriate storage location on your system. You can also decide whether you're using more than one file, how many, and whether those files roll over. All of those are management decisions that you'll have to deal with as part of working with your environment and your SQL query monitoring. You can run this 24/7, but you have to be prepared to deal with large amounts of data depending on how stringent the filters you've created are.

In addition to the buffer or the file, you have other output options, but they're usually reserved for special types of monitoring and not usually necessary for query performance tuning.

Finishing the Session

Once you've defined the storage, you've set everything needed for the session. There is an Advanced page as well, but you really shouldn't need to modify this from the defaults on most systems. When you click OK, the session will get created. If you instructed on the first tab that the session start after creation, it will start immediately, but whether it starts or not, it will be stored on the server. One of the beauties of Extended Events sessions is that they're stored on the server, so you can turn them on and off as needed with no need to re-create the session. The sessions are stored permanently until you remove them and will even survive a reboot, although, depending on how you've configured the session, you may have to restart them as necessary.

Assuming you either didn't automatically start the session or selected the option to watch the data live, you can do both to the session you just created. Right-click the session, and you'll see a menu of actions including Start Session, Stop Session, and Watch Live Data. If you start the session and you chose to observe the output, you should see a new window appear in Management Studio showing the events you're capturing. These events are coming off the same buffer as the one that is writing out to disk, so you can watch events in real time. Take a look at Figure 6-7 to see this in action.

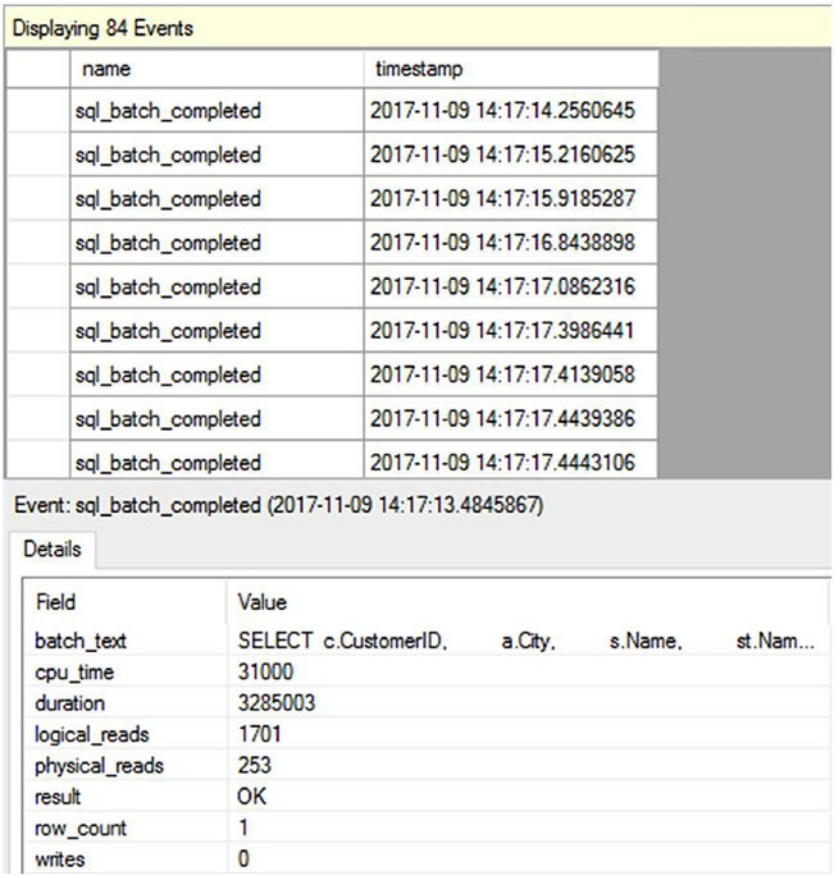

Figure 6-7. *Live output of the Extended Events session created by the wizard*

You can see the events at the top of the window showing the type of event and the date and time of the event. Clicking the event at the top will open the fields that were captured with the event on the bottom of the screen. As you can see, all the information I've been talking about is available to you. Also, if you're unhappy with having a divided output, you can right-click a column and select Show Column in Table from the context menu. This will move it up into the top part of the screen, displaying all the information in a single location, as shown in Figure 6-8.

	name	timestamp	batch_text
	sql_batch_completed	2017-11-09 14:16:34.0421370	SELECT @@SPID;
▶	sql_batch_completed	2017-11-09 14:17:13.4845867	SELECT c.CustomerID, a.City, s.Name, s...
	sql_batch_completed	2017-11-09 14:16:34.7759510	SELECT @@SPID;
	sql_batch_completed	2017-11-09 14:16:34.8497793	EXEC dbo.AddressByCity @City = N'London' -- nvarchar(30)
	sql_batch_completed	2017-11-09 14:16:39.3452838	SELECT @@SPID;
	sql_batch_completed	2017-11-09 14:16:39.3542044	EXEC dbo.AddressByCity @City = N'Mentor' -- nvarchar(30)
	sql_batch_completed	2017-11-09 14:16:55.0558793	SELECT SYSTEM_USER
	sql_batch_completed	2017-11-09 14:16:55.0561748	SET ROWCOUNT 0 SET TEXTSIZE 2147483647 SET N...
	sql_batch_completed	2017-11-09 14:16:55.0568252	select @@spid; select SERVERPROPERTY('ProductLev...

Figure 6-8. *The statement column has been added to the table.*

You can also open the files you've collected through this interface and use it to browse the data. You can search within a column on the collected data, sort by them, and group by fields. One of the great ways to see an aggregate of all calls to a particular query is to use `query_hash`, a global field that you can add to your data collection. The GUI offers a lot of ways to manipulate the information you've collected.

Watching this information through the GUI and browsing through files is fine, but you're going to want to automate the creation of these sessions. That's what the next section covers.

The Built-in system_health Session

Built in to SQL Server and automatically running by default, there is an Extended Event session called `system_health`. It's primarily meant as a mechanism for observing the overall health of the system and collecting errors and diagnostics about internals. However, it also automatically captures some information that is useful when we're talking about query performance tuning.

By default, out of the box, it collects the full information on deadlocks as they occur. Deadlocks are absolutely a performance issue and are covered in Chapter 22. The `system_health` Extended Events session means we don't have to do any other work to begin diagnosing deadlock situations.

The system_health session captures the sql_text and session_id for any processes that have waited on latches for longer than 15 seconds. That information is useful for immediately identifying queries that may need tuning. You also get the sql_text and session_id for any queries that waited longer than 30 seconds for a lock. Again, this is a way to identify immediately, with no other work than searching the system_health information, which queries may need tuning.

Because this is just another session, you have full control over it and can even remove it from your system, although I certainly don't recommend that. It collects its information in a 5MB file and keeps a rolling set of four files. You won't be able to go back to the beginning of your server install with this information, but it should have all the recent behavior of your server. The files are located by default with your other log files. You can find the location like this:

```
SELECT path
FROM sys.dm_os_server_diagnostics_log_configurations;
```

With that location you can query the session or open it in the Live Data explorer window, as shown in Figure 6-9.

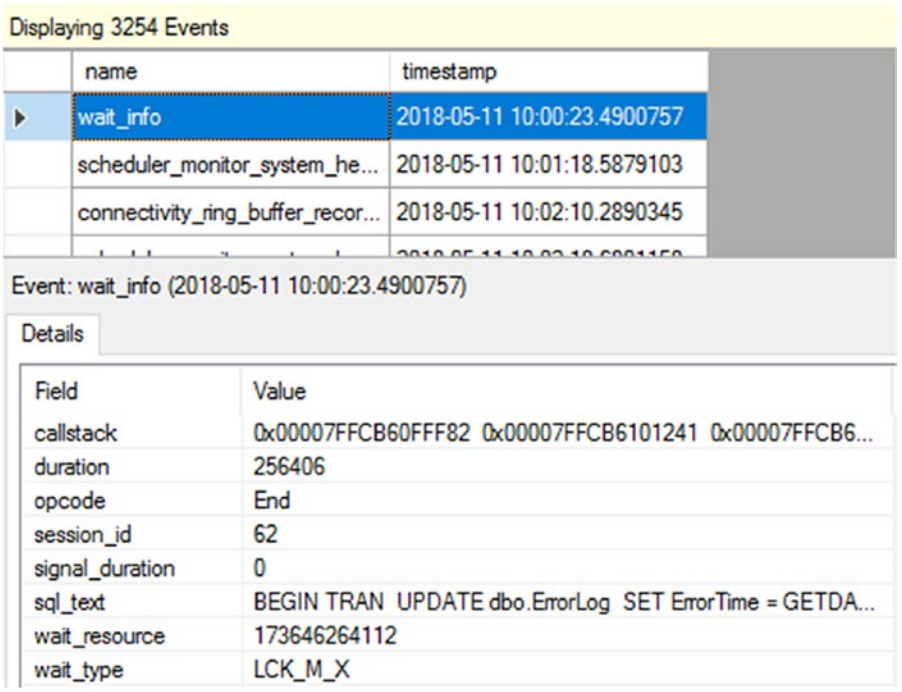

Figure 6-9. *Wait_info event in the system_health Extended Event session*

The event on display in Figure 6-9 is the `wait_info` event, which shows that I had a process waiting to obtain a lock for more than 30 seconds. The `sql_text` field will show the query in question. As you can see, from a performance tuning standpoint, this is invaluable information. Best of all, it's available on your systems right now. You don't have to do anything to set it up.

Extended Events Automation

The ability to use the GUI to build a session and define the events you want to capture does make things simple, but, unfortunately, it's not a model that will scale. If you need to manage multiple servers where you're going to create sessions for capturing key query performance metrics, you're not going to want to connect to each one and go through the GUI to select the events, the output, and so on. This is especially true if you take into account the chance of a mistake. Instead, it's much better to learn how to work with sessions directly from T-SQL. This will enable you to build a session that can be run on a number of servers in your system. Even better, you're going to find that building sessions directly is easier in some ways than using the GUI, and you're going to be much more knowledgeable about how these processes work.

Creating a Session Script Using the GUI

You can create a scripted trace in one of two ways, manually or with the GUI. Until you get comfortable with all the requirements of the scripts, the easy way is to use the Extended Events GUI. These are the steps you'll need to perform:

1. Define a session.
2. Right-click the session, and select Script Sessions As, CREATE To, and File to output straight to a file. Or, use the Script button at the top of the New Session window to create a T-SQL command in the Query window.

These steps will generate the script you need to create a session and output it to a file. To manually create this new trace, use Management Studio as follows:

1. Open the script file or navigate to the Query window.
2. Modify the path and file location for the server you're creating this session on.
3. Execute the script.

Once the session is created, you can use the following command to start it:

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

You may want to automate the execution of the last step through the SQL Agent, or you can even run the script from the command line using the `sqlcmd.exe` utility. Whatever method you use, the final step will start the session. To stop the session, just run the same script with the STATE set to stop. I'll show how to do that in the next section.

Defining a Session Using T-SQL

If you followed the steps from the previous section to create a script, you would see something like this in your Query Editor window:

```
CREATE EVENT SESSION [QueryMetrics]
ON SERVER
    ADD EVENT sqlserver.sql_batch_completed
    (SET collect_batch_text = (1)
     WHERE ([sqlserver].[database_name] = N'AdventureWorks2017')
    )
    ADD TARGET package0.event_file
    (SET filename = N'q:\PerfData\QueryMetrics')
WITH
(
    MAX_MEMORY = 4096KB,
    EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,
    MAX_DISPATCH_LATENCY = 30 SECONDS,
    MAX_EVENT_SIZE = 0KB,
    MEMORY_PARTITION_MODE = NONE,
    TRACK_CAUSALITY = OFF,
    STARTUP_STATE = OFF
);
GO
```

To create an Extended Events session, a single command defines the session, `CREATE EVENT SESSION`. You then just use `ADD EVENT` within that command to define the session. The filters are simply a `WHERE` clause added to each event definition. Finally, you add a target defining where the data captured should be stored. The `WITH` clause is actually just the default values from the Advanced page in the GUI. You can leave off the `WITH` clause and those values, and they'll still be set for the session.

Once the session has been defined, you can activate it using `ALTER EVENT`, as shown earlier.

Once a session is started on the server, you don't have to keep Management Studio or the Query Editor open anymore. You can identify the active sessions by using the dynamic management view `sys.dm_xe_sessions`, as shown in the following query:

```
SELECT  dxs.name,
        dxs.create_time
FROM    sys.dm_xe_sessions AS dxs;
```

Figure 6-10 shows the output of the view.

	name	create_time
1	hkenginexesession	2017-11-09 05:16:09.693
2	system_health	2017-11-09 05:16:10.193
3	sp_server_diagnostics session	2017-11-09 05:16:10.347
4	QueryMetrics	2017-11-09 16:07:09.507
5	telemetry_xevents	2017-11-10 11:47:52.280

Figure 6-10. *Output of sys.dm_xe_sessions*

The number of rows returned indicates the number of sessions active on SQL Server. I have four other sessions, all system defaults, running in addition to the one I created in this chapter. You can stop a specific session by executing the stored procedure `ALTER EVENT SESSION`.

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

To verify that the session is stopped successfully, reexecute the query against the catalog view `sys.dm_xe_sessions`, and ensure that the output of the view doesn't contain the named session.

Using a script to create your sessions allows you to automate across a large number of servers. Using the scripts to start and stop the sessions means you can control them through scheduled events such as through SQL Agent. In Chapter 20, you will learn how to control the schedule of a session while capturing the activities of a SQL workload over an extended period of time.

Note The time captured through a session defined as illustrated in this section is stored in microseconds, not milliseconds. This difference between units can cause confusion if not taken into account. You must filter based on microseconds.

Using Causality Tracking

Defining sessions through either the GUI or T-SQL is fairly simple. Consuming the information is also pretty easy. However, you'll quickly find that you don't simply want to observe single batch statements or single procedure calls. You're going to want to see all the statements within a procedure as well as the procedure call. You're going to want to see statement-level recompiles, waits, and all sorts of other events and have them all directly tied together back to an individual stored procedure or statement. That's where *causality tracking* comes in.

You can enable causality tracking as noted earlier through the GUI, or you can include it in an SQL command. The following script captures the start and stop of remote procedure calls and all the statements within those calls. I've also enabled causality tracking.

```
CREATE EVENT SESSION ProcedureMetrics
ON SERVER
    ADD EVENT sqlserver.rpc_completed
    (WHERE (sqlserver.database_name = N'AdventureWorks2017')),
    ADD EVENT sqlserver.rpc_starting
    (WHERE (sqlserver.database_name = N'AdventureWorks2017')),
    ADD EVENT sqlserver.sp_statement_completed
```

```
    (SET collect_object_name = (1))
    ADD TARGET package0.event_file
    (SET filename = N'C:\PerfData\ProcedureMetrics.xel')
WITH
(
    TRACK_CAUSALITY = ON
);
```

Extended Events Recommendations

Extended Events is such a game-changer in the way that information is collected that many of the problematic areas that used to come up when using trace events have been largely eliminated. You have a much reduced need to worry as much about severely limiting the number of events collected or the number of fields returned. But, as was noted earlier, you can still negatively impact the system by overloading the events being collected. There are still a few specific areas you need to watch out for.

- Set the max file size appropriately.
- Be cautious with debug events.
- Avoid use of `No_Event_Loss`.

I'll go over these in a little more detail in the following sections.

Set Max File Size Appropriately

The default value for files is 1GB. That's actually very small when you consider the amount of information that can be gathered with Extended Events. It's a good idea to set this number much higher, somewhere in the 50GB to100GB range to ensure you have adequate space to capture information and you're not waiting on the file subsystem to create files for you while your buffer fills. This can lead to event loss. But, it does depend on your system. If you have a good grasp of the level of output you can expect, set the file size more appropriate to your individual environment.

Be Cautious with Debug Events

Not only does Extended Events provide you with a mechanism for observing the behavior of SQL Server and its internals in a way that far exceeds what was possible under trace events, but Microsoft uses the same functionality as part of troubleshooting SQL Server. A number of events are related to debugging SQL Server. These are not available by default through the wizard, but you do have access to them through the T-SQL command, and there's a way to enable them through the channel selection in the Session editor window.

Without direct guidance from Microsoft, do not use them. They are subject to change and are meant for Microsoft internal use only. If you do feel the need to experiment, you need to pay close attention to any of the events that include a break action. This means that should the event fire, it will stop SQL Server at the exact line of code that caused the event to fire. This means your server will be completely offline and in an unknown state. This could lead to a major outage if you were to do it in a production system. It could lead to loss of data and corruption of your database.

However, not all of them lead to break actions, and some are even recommended for use. One example is the `query_thread_profile` event. Running this enables you the ability to capture live execution plan events in a light-weight fashion. We'll cover this in more detail in Chapter 15 when we talk about execution plans.

Avoid Use of No_Event_Loss

Extended Events is set up such that some events will be lost. It's extremely likely, by design. But, you can use a setting, `No_Event_Loss`, when configuring your session. If you do this on systems that are already under load, you may see a significant additional load placed on the system since you're effectively telling it to retain information in the buffer regardless of consequences. For small and focused sessions that are targeting a particular behavior, this approach can be acceptable.

Other Methods for Query Performance Metrics

Setting up an Extended Events session allows you to collect a lot of data for later use, but the collection can be a little bit expensive. In addition, you have to wait on the results, and then you have a lot of data to deal with. Another mechanism that comes with a

smaller overall cost is the Query Store. We'll cover that in detail in Chapter 11. If you need to immediately capture performance metrics about your system, especially as they pertain to query performance, then the dynamic management views `sys.dm_exec_query_stats` for queries and `sys.dm_exec_procedure_stats` for stored procedures are what you need. If you still need a historical tracking of when queries were run and their individual costs, an Extended Events session is still the best tool. But if you just need to know, at this moment, the longest-running queries or the most physical reads, then you can get that information from these two dynamic management objects. But, the data in these objects is dependent on the query plan remaining in the cache. If the plan ages out of cache, this data just goes away. The `sys.dm_exec_query_stats` DMO will return results for all queries, including stored procedures, but the `sys.dm_exec_procedure_stats` will return information only for stored procedures.

Since both these DMOs are just views, you can simply query against them and get information about the statistics of queries in the plan cache on the server. Table 6-5 shows some of the data returned from the `sys.dm_exec_query_stats` DMO.

Table 6-5 is just a sampling. For complete details, see Books Online.

Table 6-5. *sys.dm_exec_query_stats Output*

Column	Description
`Plan_handle`	Pointer that refers to the execution plan
`Creation_time`	Time that the plan was created
`Last_execution time`	Last time the plan was used by a query
`Execution_count`	Number of times the plan has been used
`Total_worker_time`	Total CPU time used by the plan since it was created
`Total_logical_reads`	Total number of reads used since the plan was created
`Total_logical_writes`	Total number of writes used since the plan was created
`Query_hash`	A binary hash that can be used to identify queries with similar logic
`Query_plan_hash`	A binary hash that can be used to identify plans with similar logic
`Max_dop`	The max degree of parallelism that was used by the query
`Max_columnstore_segment_skips`	The number of segments that have been skipped over during a query

To filter the information returned from `sys.dm_exec_query_stats`, you'll need to join it with other dynamic management functions such as `sys.dm_exec_sql_text`, which shows the query text associated with the plan, or `sys.dm_query_plan`, which has the execution plan for the query. Once joined to these other DMOs, you can filter on the database or procedure that you want to see. These other DMOs are covered in detail in other chapters of the book. I'll show examples of using `sys.dm_exec_query_stats` and the others, in combination, throughout the rest of the book. Just remember that these queries are cache dependent. As a given execution plan ages out of the cache, this information will be lost.

Summary

In this chapter, you saw that you can use Extended Events to identify the queries causing a high amount of stress on the system resources in a SQL workload. Collecting the session data can, and should be, automated using system stored procedures. For immediate access to statistics about running queries, use the DMV `sys.dm_exec_query_stats`.

Now that you have a mechanism for gathering metrics on queries that have been running against your system, in the next chapter you'll explore how to gather information about a query as it runs so that you don't have to resort to these measurement tools each time you run a query.

CHAPTER 7

Analyzing Query Performance

The previous chapter showed how to gather query performance metrics. This chapter will show how to consume those metrics to identify long-running or frequently called queries. Then I'll go over the tools built into Management Studio so you can understand how a given query is performing. I'll also spend a lot of time talking about using execution plans, which are your best view into the decisions made by the query optimizer.

In this chapter, I cover the following topics:

- How to analyze the processing strategy of a costly SQL query using Management Studio
- How to analyze methods used by the query optimizer for a SQL query
- How to measure the cost of a SQL query using T-SQL commands

Costly Queries

Now that you have seen two different ways of collecting query performance metrics, let's look at what the data represents: the costly queries themselves. When the performance of SQL Server goes bad, a few things are most likely happening.

- First, certain queries create high stress on system resources. These queries affect the performance of the overall system because the server becomes incapable of serving other SQL queries fast enough.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_7

- Additionally, the costly queries block all other queries requesting the same database resources, further degrading the performance of those queries. Optimizing the costly queries improves not only their own performance but also the performance of other queries by reducing database blocking and pressure on SQL Server resources.
- It's possible that changes in data or the values passed to queries results in changes in the behavior of the query, degrading its performance.
- Finally, a query that by itself is not terribly costly could be called thousands of times a minute, which, by the simple accumulation of less than optimal code, can lead to major resource bottlenecks.

To begin to determine which queries you need to spend time working with, you're going to use the resources that I've talked about so far. For example, assuming the queries are in cache, you will be able to use the DMOs to pull together meaningful data to determine the most costly queries. Alternatively, because you've captured the queries using Extended Events, you can access that data as a means to identify the costliest queries. One other option is also possible, introduced with SQL Server 2016; you can use the Query Store to capture and examine query performance metrics. We'll examine that mechanism in detail in Chapter 11.

Here we're going to start with Extended Events. The single easiest and most immediate way to capture query metrics is through the DMOs against the queries currently in cache. Unfortunately, this is aggregated data and completely dependent on what is currently in cache (we'll talk about the cache more in Chapter 16), so you don't have a historical record, and you don't get individual measurements and individual parameter values on stored procedures. The second easiest and equally immediate method for looking at query metrics is through the Query Store. It's a more complete record than the DMOs supply, but the data there is aggregated as well. We'll explore all three, but for precision, we'll start with Extended Events.

One small note on the Extended Events data: if it's going to be collected to a file, you'll then need to load the data into a table or just query it directly. You can read directly from the Extended Events file by querying it using this system function:

```
SELECT module_guid,
       package_guid,
       object_name,
```

```
        event_data,
        file_name,
        file_offset,
        timestamp_utc
FROM sys.fn_xe_file_target_read_file('C:\Sessions\QueryPerformanceMetrics*.
xel',
                                     NULL,
                                     NULL,
                                     NULL);
```

The parameters required are first the path, which I supplied. You can use * as I did to deal with the fact that there are multiple rollover files. The second parameter is a holdover from SQL Server 2008R2 and can be ignored. The third parameter will let you pick an initial file name; otherwise, if you do what I did, it'll read all the files from the path. Finally, the last parameter lets you specify an offset so that you can, if you like, skip past certain events. It's only a number, so you can't really filter beyond events; just count to the one you want to start with.

The query returns each event as a single row. The data about the event is stored in an XML column, `event_data`. You'll need to use XQuery to read the data directly, but once you do, you can search, sort, and aggregate the data captured. I'll walk you through a full example of this mechanism in the next section.

Identifying Costly Queries

The goal of SQL Server is to return result sets to the user in the shortest time. To do this, SQL Server has a built-in, cost-based optimizer called the *query optimizer,* which generates a cost-effective strategy called a *query execution plan.* The query optimizer weighs many factors, including (but not limited to) the usage of CPU, memory, and disk I/O required to execute a query, all derived from the various sources such as statistics about the data maintained by indexes or generated on the fly, constraints on the data, and some knowledge of the system the queries are running such as the number of CPUs and the amount of memory. From all that the optimizer creates a cost-effective execution plan.

In the data returned from a session, the `cpu_time` and `logical_reads` or `physical_reads` fields also show where a query costs you. The `cpu_time` field represents the CPU time used to execute the query. The two `reads` fields represent the number of pages