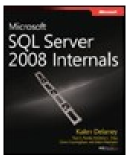


Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.

Chapter 2: Change Tracking, Tracing, and Extended Events

Adam Machanic

Overview

As the Microsoft SQL Server engine processes user requests, a variety of actions can occur: data structures are interrogated; files are read from or written to; memory is allocated, deallocated, or accessed; data is read or modified; an error may be raised; and so on. Classified as a group, these actions can be referred to as the collection of *run-time events* that can occur within SQL Server.

From the point of view of a user—a DBA or database developer working with SQL Server—the fact that certain events are occurring may be interesting in the context of supporting debugging, auditing, and general server maintenance tasks. For example, it may be useful to track when a specific error is raised, every time a certain column is updated, or how much CPU time various stored procedures are consuming.

To support these kinds of user scenarios, the SQL Server engine is instrumented with a variety of infrastructures designed to support event consumption. These range from relatively simple systems such as triggers—which call user code in response to data modifications or other events—to the complex and extremely flexible Extended Events Engine, which is new in SQL Server 2008.

This chapter covers the key areas of each of the common event systems that you might encounter as a SQL Server DBA or database developer: triggers, event notifications, Change Tracking, SQL Trace, and extended events. Each of these has a similar basic goal—to react or report when something happens—but each works somewhat differently.

The Basics: Triggers and Event Notifications

Although the majority of this chapter is concerned with larger and more complex eventing infrastructures, the basics of how SQL Server internally deals with events can be learned more easily through an investigation of triggers and event notifications; therefore, they are a good place to begin.

Triggers come in a couple of basic varieties. Data Manipulation Language (DML) triggers can be defined to fire on operations like inserts and updates, and Data Definition Language (DDL) triggers can be defined to fire on either server-level or database-level actions such as creating a login or dropping a table. DML triggers can fire instead of the triggering event, or after the event has completed but before the transaction is committed. DDL triggers can be configured to fire only after the event has completed, but again, before the transaction has committed. Event notifications are really nothing more than special-case DDL triggers that send a message to a SQL Service Broker queue rather than invoking user code. The most important difference is that they do not require a transaction and as a result support many non-transactional events—for example, a user disconnecting from the SQL Server instance—that standard DDL triggers do not.

Run-Time Trigger Behavior

DML triggers and DDL triggers have slightly different run-time behaviors owing to their different modes of operation and the nature of the required data within the trigger. Because DDL triggers are associated with metadata operations, they require much less data than their DML counterparts.

DML triggers are resolved during DML compilation. After the query has been parsed, each table involved is checked via an internal function for the presence of a trigger. If triggers are found, they are compiled and checked for tables that have triggers, and the process recursively continues. During the actual DML operation, the triggers are fired and the rows in the inserted and deleted virtual tables are populated in *tempdb*, using the version store infrastructure.

DDL triggers and event notifications follow similar paths, which are slightly different from that of DML triggers. In both cases, the triggers themselves are resolved via a check only after the DDL change to which they are bound has been applied. DDL triggers and event notifications are fired after the DDL operation has occurred, as a post-operation step rather than during the operation as with DML triggers. The only major difference between DDL triggers and event notifications is that DDL triggers run user-defined code, whereas event notifications send a message to a Service Broker queue.

Change Tracking

Change Tracking is a feature designed to help eliminate the need for many of the custom synchronization schemes that developers must often create from scratch during an application's lifetime. An example of this kind of system is when an application pulls data from the database into a local cache and occasionally asks the database whether any of the data has been updated, so that the data in the local store can be brought up to date. Most of these systems are implemented using triggers or timestamps, and they are often riddled with performance issues or subtle logic flaws. For example, schemes using timestamps often break down if the timestamp column is populated at insert time rather than at commit time. This can cause a problem if a large insert happens simultaneously with many smaller inserts, and the large insert commits later than smaller inserts that started afterward, thereby ruining the ascending nature of the timestamp. Triggers can remedy this particular problem, but they cause their own problems—namely, they can introduce blocking issues because they increase the amount of time needed for transactions to commit.

Unlike custom systems, Change Tracking is deeply integrated into the SQL Server relational engine and designed from the ground up with performance and scalability in mind. The system is designed to track data changes in one or more tables in a database and is designed to let the user easily determine the order in which changes occurred, as a means by which to support multitable synchronization. Changes are tracked synchronously as part of the transaction in which the change is made, meaning that the list of changed rows is always up to date and consistent with the actual data in the table.

Change Tracking is based on the idea of working forward from a baseline. The data consumer first requests the current state of all the rows in the tracked tables and is given a version number with each row. The baseline version number—effectively, the maximum version number that the system currently knows about—is also queried at that time and is recorded until the next synchronization request. When the request is made, the baseline version number is sent back to the Change Tracking system, and the system determines which rows have been modified since the first request. This way, the consumer needs to concern itself only with deltas; there is generally no reason to reacquire rows that have not changed. In addition to sending a list of rows that have changed, the system identifies the nature of the change since the baseline—a new row, an update to an existing row, or a deleted row. The maximum row version returned when requesting an update becomes the new baseline.

SQL Server 2008 includes two similar technologies that can be used to support synchronization: Change Tracking and Change Data Capture (the details of which are outside the scope of this book because it is not an engine feature per se—it uses an external log reader to do its work). It is worth spending a moment to discuss where and when Change Tracking should be used. Change Tracking is designed to support offline applications, occasionally connected applications, and other applications that don't need real-time notification as data is updated. The Change Tracking system sends back only the current versions of any rows requested after the baseline—incremental row states are not preserved—so the ideal Change Tracking application does not require the full history of a given row. As compared with Change Data Capture, which records the entire modification history of each row, Change Tracking is lighter weight and less applicable to auditing and data warehouse extract, transform, and load (ETL) scenarios.

Change Tracking Configuration

Although Change Tracking is designed to track changes on a table-by-table basis, it is actually configured at two levels: the database in which the tables reside and the tables themselves. A table cannot be enabled for Change Tracking until the feature has been enabled in the containing database.

Database-Level Configuration

SQL Server 2008 extends the *ALTER DATABASE* command to support enabling and disabling Change Tracking, as well as configuring options that define whether and how often the history of changes that have been made to participating tables is purged. To enable Change Tracking for a database with the default options, the following *ALTER DATABASE* syntax is used:

```
ALTER DATABASE AdventureWorks2008
SET CHANGE_TRACKING = ON;
```

Running this statement enables a configuration change to metadata that allows two related changes to occur once table-level configuration is enabled: First, a hidden system table will begin getting populated in the target database, should qualifying transactions occur (see the next section). Second, a cleanup task will begin eliminating old rows found in the internal table and related tables.

Commit Table

The hidden table, known as the Commit Table, maintains one row for every transaction in the database that modifies at least one row in a table that participates in Change Tracking. At transaction commit time, each qualifying transaction is assigned a unique, ascending identifier called a Commit Sequence Number (CSN). The CSN is then inserted—along with the transaction identifier, log sequence information, begin time, and other data—into the Commit Table. This table is central to the Change Tracking process and is used to help determine which changes need to be synchronized when a consumer requests an update, by maintaining a sequence of committed transactions.

Although the Commit Table is an internal table and users can't access it directly, except administrators, via the dedicated administrator connection (DAC), it is still possible to view its columns and indexes by starting with the *sys.all_columns* catalog view. The physical name for the table is *sys.syscommitab*, and the following query returns six rows, as described in Table 2-1:

```
SELECT *
FROM sys.all_columns
WHERE object_id = OBJECT_ID('sys.syscommittab');
```

Table 2-1: Columns in the sys.syscommittab System Table

Column Name	Type	Description
commit_ts	*BIGINT*	The ascending CSN for the transaction
xdes_id	*BIGINT*	The internal identifier for the transaction
commit_lbn	*BIGINT*	The log block number for the transaction
commit_csn	*BIGINT*	The instance-wide sequence number for the transaction
commit_time	*DATETIME*	The time the transaction was committed
dbfragid	*INT*	Reserved for future use

The *sys.syscommitab* table has two indexes (which are visible via the *sys.indexes* catalog view): a unique clustered index on the *commit_ts* and *xdes_id* columns and a unique nonclustered index on the *xdes_id* column that includes the *dbfragid* column. None of the columns are nullable, so the per-row data size is 44 bytes for the clustered index and 20 bytes for the non-clustered index.

Note that this table records information about transactions, but none about which rows were actually modified. That related data is stored in separate system tables, created when Change Tracking is enabled on a user table. Because one transaction can span many different tables and many rows within each table, storing the transaction-specific data in a single central table saves a considerable number of bytes that need to be written during a large transaction.

All the columns in the *sys.syscommitab* table except *dbfragid* are visible via the new *sys.dm_tran_commit_table* DMV. This view is described by *SQL Server Books Online* as being included for "supportability purposes," but it can be interesting to look at for the purpose of learning how Change Tracking behaves internally, as well as to watch the cleanup task, discussed in the next section, in action.

Internal Cleanup Task

Once Change Tracking is enabled and the Commit Table and related hidden tables fill with rows, they can begin taking up a considerable amount of space in the database. Consumers—that is, synchronizing databases and applications—may not need a change record beyond a certain point of time, and so keeping it around may be a waste. To eliminate this overhead, Change Tracking includes functionality to enable an internal task that removes change history on a regular basis.

When enabling Change Tracking using the syntax listed previously, the default setting, Remove History Older Than Two Days, is used. This setting can be specified when enabling Change Tracking using optional parameters to the *ALTER DATABASE* syntax:

```
ALTER DATABASE AdventureWorks2008
SET CHANGE_TRACKING = ON
(AUTO_CLEANUP=ON, CHANGE_RETENTION=1 hours);
```

The AUTO_CLEANUP option can be used to disable the internal cleanup task completely, and the CHANGE_RETENTION option can be used to specify the interval after which history should be removed, in an interval that can be defined by a number of minutes, hours, or days.

If enabled, the internal task runs once every 30 minutes and evaluates which transactions need to be removed by subtracting the retention interval from the current time and then using an interface into the Commit Table to find a list of transaction IDs older than that period. These transactions are then purged from both the Commit Table and other hidden

Change Tracking tables.

The current cleanup and retention settings for each database can be queried from the *sys.change_tracking_databases* catalog view.

> **Note** When setting the cleanup retention interval, it is important to err on the side of being too long, to ensure that data consumers do not end up with a broken change sequence. If this does become a concern, applications can use the *CHANGE_TRACKING_MIN_VALID_VERSION* function to find the current minimum version number stored in the database. If the minimum version number is higher than the application's current baseline, the application has to resynchronize all data and take a new baseline.

Table-Level configuration

Once Change Tracking is enabled at the database level, specific tables must be configured to participate. By default, no tables are enlisted in Change Tracking as a result of the feature being enabled at the database level.

The *ALTER TABLE* command has been modified to facilitate enabling of Change Tracking at the table level. To turn on the feature, use the new ENABLE CHANGE_TRACKING option, as shown in the following example:

```
ALTER TABLE HumanResources.Employee
ENABLE CHANGE_TRACKING;
```

If Change Tracking has been enabled at the database level, running this statement causes two changes to occur. First, a new internal table is created in the database to track changes made to rows in the target table. Second, a hidden column is added to the target table to enable tracking of changes to specific rows by transaction ID. An optional feature called Column Tracking can also be enabled; this is covered in the section entitled "Column Tracking," later in this chapter.

Internal Change Table

The internal table created by enabling Change Tracking at the table level is named *sys.change_tracking_[object id]*, where *[object id]* is the database object ID for the target table. The table can be seen by querying the *sys.all_objects* catalog view and filtering on the *parent_object_id* column based on the object ID of the table you're interested in, or by looking at the *sys.internal_tables* view for tables with an *internal_type* of 209.

The internal table has five static columns, plus at least one additional column depending on how many columns participate in the target table's primary key, as shown in Table 2-2.

Table 2-2: Columns in the Internal Change Tracking Table

Column Name	Type	Description
sys_change_xdes_id	*BIGINT NOT NULL*	Transaction ID of the transaction that modified the row.
sys_change_xdes_id_seq	*BIGINT NOT NULL (IDENTITY)*	Sequence identifier for the operation within the transaction.
sys_change_operation	*NCHAR(1) NULL*	Type of operation that affected the row: insert, update, or delete.
sys_change_columns	*VARBINARY(4100) NULL*	List of which columns were modified (used for updates, only if column tracking is enabled).
sys_change_context	*VARBINARY(128) NULL*	Application-specific context information provided during the DML operation using the WITH CHANGE_ TRACKING_CONTEXT option.
k_[name]_[ord]	[type] *NOT NULL*	Primary key column(s) from the target table. *[name]* is the name of the primary key column, *[ord]* is the ordinal position in the key, and *[type]* is the data type of the column.

Calculating the per-row overhead of the internal table is a bit trickier than for the Commit Table, as several factors can influence overall row size. The fixed cost includes 18 bytes for the transaction ID, CSN, and operation type, plus the size of the primary key from the target table. If the operation is an update and column tracking is enabled (as described in the section entitled "Column Tracking," later in this chapter), up to 4,100 additional bytes per row may be consumed by the *sys_change_columns* column. In addition, context information—such as the name of the application or user doing the modification—can be provided using the new WITH CHANGE_TRACKING_CONTEXT DML option (see the section entitled "Query Processing and DML Operations," later in this chapter), and this adds a maximum of another 128 bytes to each row.

The internal table has a unique clustered index on the transaction ID and transaction sequence identifier and no nonclustered indexes.

Change Tracking Hidden Columns

In addition to the internal table created when Change Tracking is enabled for a table, a hidden 8-byte column is added to the table to record the transaction ID of the transaction that last modified each row. This column is not visible in any relational engine metadata (that is, catalog views and the like), but can be seen referenced in query plans as *$sys_change_xdes_id*. In addition, you may notice the data size of tables increasing accordingly after Change Tracking is updated. This column is removed, along with the internal table, if Change Tracking is disabled for a table.

Note The hidden column's value can be seen by connecting via the DAC and explicitly referencing the column name. It never shows up in the results of a *SELECT* * query.

Change Tracking Run-Time Behavior

The various hidden and internal objects covered to this point each have a specific purpose when Change Tracking interfaces with the query processor at run time. Enabling Change Tracking for a table modifies the behavior of every subsequent DML operation against the table, in addition to enabling use of the *CHANGETABLE* function that allows a data consumer to find out which rows have changed and need to be synchronized.

Query Processing and DML Operations

Once Change Tracking has been enabled for a given table, all existing query plans for the table that involve row modification are marked for recompilation. New plans that involve modifications to the rows in the table include an insert into the internal change table, as shown in Figure 2-1. Because the internal table represents all operations—inserts, updates, and deletes—by inserting new rows, the subtree added to each of the new query plans is virtually identical.

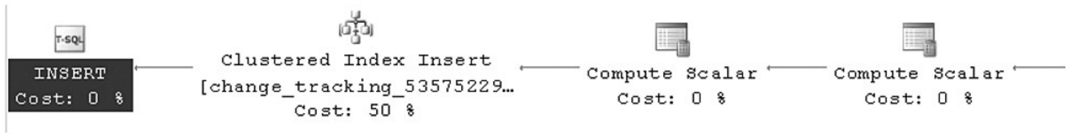


Figure 2-1: Query plan subtree involving an insert into the internal change table

In addition to the insert into the internal table, the query processor begins processing a new DML option thanks to Change Tracking: the *WITH CHANGE_TRACKING_CONTEXT* function. This function allows the storage of up to 128 bytes of binary data, alongside other information about the change, in the internal table's *sys_change_context* column. This column can be used by developers to persist information about which application or user made a given change, using the Change Tracking system as a metadata repository with regard to row changes.

The syntax for this option is similar to a Common Table Expression and is applied at the beginning of the DML query, as in the following example:

```
DECLARE @context VARBINARY(128) =
    CONVERT(VARBINARY(128), SUSER_SNAME());

WITH CHANGE_TRACKING_CONTEXT(@context)
UPDATE AdventureWorks2008.HumanResources.Employee
SET
    JobTitle = 'Production Engineer'
WHERE
    JobTitle = 'Design Engineer';
```

Note This syntax is perfectly valid for tables that do not have Change Tracking enabled. However, in those cases, the query processor simply ignores the call to the *CHANGE_TRACKING_CONTEXT* function.

In addition to the insert into the internal table that occurs synchronously at the end of the transaction, an insert into the Commit Table also occurs at commit time. The inserted row contains the same transaction ID that is used both in the internal table and in the hidden column on the target table. A CSN is also assigned for the transaction at this time; this number can, therefore, be thought of as the version number that applies to all rows modified by the transaction.

Column Tracking

When working with tables that have a large number of columns or tables with one or more extremely wide columns, the synchronization process can be optimized by not reacquiring the data from those columns that were not updated. To support this kind of optimization, Change Tracking includes a feature called Column Tracking, which works by recording, in the internal table and only in the case of an update operation, which columns were updated.

The column list is persisted within the internal table in the *sys_change_columns* column. Each column is stored as an integer, and a column list including as many as 1,024 columns can be stored. If more than 1,024 columns are modified in a transaction, the column list is not stored and the application must reacquire the entire row.

To enable Column Tracking, a switch called TRACK_COLUMNS_UPDATED is applied to the *ALTER TABLE* statement, as in the following example:

```
ALTER TABLE HumanResources.Employee
ENABLE CHANGE_TRACKING
WITH (TRACK_COLUMNS_UPDATED = ON);
```

Once enabled, the changed columns list is returned with the output of the *CHANGETABLE(CHANGES)* function, which is described in the next section. The bitmap can be evaluated for the presence of a particular column by using the *CHANGE_TRACKING_IS_ COLUMN_IN_MASK* function.

> **Caution** Be careful when enabling Column Tracking for active tables. Although this feature may help to optimize the synchronization process by resulting in fewer bytes being sent out at synchronization time, it also increases the number of bytes that must be written with each update against the target table. This may result in a net decrease in overall performance if the columns are not sufficiently large enough to balance the additional byte requirements of the bitmap.

CHANGETABLE Function

The primary API that users can use to leverage the Change Tracking system is the *CHANGETABLE* function. This function has the dual purpose of returning the baseline version for all rows in the target table and returning a set containing only updated versions and related change information. The function accomplishes each of these tasks with the help of the various internal and hidden structures created and populated when Change Tracking is enabled for a given table or set of tables in a database.

CHANGETABLE is a system table-valued function, but unlike other table-valued functions, its result shape changes at run time based on input parameters. In VERSION mode, used for acquiring the baseline values of each row in the table, the function returns only a primary key, version number, and context information for each row. In CHANGES mode, used for getting a list of updated rows, the function also returns the operation that affected the change and the column list.

Because the VERSION mode for *CHANGETABLE* is designed to help callers get a baseline, calling the function in this mode requires a join to the target table, as in the following example:

```
SELECT
    c.SYS_CHANGE_VERSION,
    c.SYS_CHANGE_CONTEXT,
    e.*
FROM AdventureWorks2008.HumanResources.Employee e
CROSS APPLY CHANGETABLE
(
    VERSION AdventureWorks2008.HumanResources.Employee,
    (BusinessEntityId),
    (e.BusinessEntityId)
) c;
```

A quick walk-through of this example is called for here. In VERSION mode, the first parameter to the function is the target table. The second parameter is a comma-delimited list of the primary key columns on the target table. The third parameter is a comma-delimited list, in the same order, of the associated primary key columns from the target table as used in the query. The columns are internally correlated in this order to support the joins necessary to get the baseline versions of each row.

When this query is executed, the query processor scans the target table, visiting each row and getting the values for every column, along with the value of the hidden column (the last transaction ID that modified the row). This transaction ID is used as a key to join to the Commit Table to pick up the associated CSN and to populate the *sys_change_version* column. The transaction ID and primary key are also used to join to the internal tracking table in order to populate the *sys_change_context* column.

Once a baseline has been acquired, it is up to the data consumer to call the *CHANGE_TRACKING_CURRENT_VERSION* function, which returns the maximum Change Tracking version number currently stored in the database. This number becomes the baseline version number that the application can use for future synchronization requests. This number is passed into the *CHANGETABLE* function in CHANGES mode to get subsequent versions of the rows in the table, as in the following example:

```
DECLARE @last_version BIGINT = 8;

SELECT
    c.*
FROM CHANGETABLE
(
    CHANGES AdventureWorks2008.HumanResources.Employee,
    @last_version
) c;
```

This query returns a list of all changed rows since version 8, along with what operation caused each row to be modified. Note that the output reflects only the most recent version of the row as of the time that the query is run. For example, if a row existed as of version 8 and was subsequently updated three times and then deleted, this query shows only one change for the row: a delete. This query includes in its output the primary keys that changed, so it is possible to join to the target table to get the most recent version of each row that changed. Care must be taken to use an OUTER JOIN in that case, as shown in the following example, as a row may no longer exist if it was deleted:

```
DECLARE @last_version BIGINT = 8;

SELECT
    c.SYS_CHANGE_VERSION,
    c.SYS_CHANGE_OPERATION,
    c.SYS_CHANGE_CONTEXT,
    e.*
FROM CHANGETABLE
(
    CHANGES AdventureWorks2008.HumanResources.Employee,
    @last_version
) c
LEFT OUTER JOIN AdventureWorks2008.HumanResources.Employee e ON
    e.BusinessEntityID = c.BusinessEntityID;
```

When *CHANGETABLE* is run in CHANGES mode, the various internal structures are used slightly differently than in the VERSION example. The first step of the process is to query the Commit Table for all transaction IDs associated with CSNs greater than the one passed in to the function. This list of transaction IDs is next used to query the internal tracking table for the primary keys associated with changes rendered by the transactions. The rows that result from this phase must be aggregated based on the primary key and transaction sequence identifier from the internal table to find the most recent row for each primary key. No join to the target table is necessary in this case unless the consumer would like to retrieve all associated row values.

Because rows may be changing all the time—including while the application is requesting a list of changes—it is important to keep consistency in mind when working with Change Tracking. The best way to ensure consistent results is to either make use of SNAPSHOT isolation if the application retrieves a list of changed keys and then subsequently requests the row value, or READ COMMITTED SNAPSHOT isolation if the values are retrieved using a JOIN. SNAPSHOT isolation and READ COMMITTED SNAPSHOT isolation are discussed in Chapter 10.

Tracing and Profiling

Query tuning, optimization, and general troubleshooting are all made possible through visibility into what's going on within SQL Server; it would be impossible to fix problems without being able to identify what caused them. SQL Trace is one of the more powerful tools provided by SQL Server to give you a real-time or near-real-time look at exactly what the database engine is doing, at a very granular level.

Included in the tracing toolset are 180 events that you can monitor, filter, and manipulate to get a look at anything from a broad overview of user logins down to such fine-grained information as the lock activity done by a specific session id (SPID). This data is all made available via SQL Server Profiler, as well as a series of server-side stored procedures and .NET classes, giving you the flexibility to roll a custom solution when a problem calls for one.

SQL Trace Architecture and Terminology

SQL Trace is a SQL Server database engine technology, and it is important to understand that the client-side Profiler tool is really nothing more than a wrapper over the server-side functionality. When tracing, we monitor for specific events that are generated when various actions occur in the database engine. For example, a user logging onto the server or executing a query are each actions that cause events to fire. These events are fired by instrumentation of the database engine code; in other words, special code has been added to these and other execution paths that cause the events to fire when hit.

Each event has an associated collection of “columns,” which are attributes that contain data collected when the event fires. For instance, in the case of a query, we can collect data about when the query started, how long it took, and how much CPU time it used. Finally, each trace can specify filters, which limit the results returned based on a set of criteria. One could, for example, specify that only events that took longer than 50 milliseconds should be returned.

With 180 events and 66 columns to choose from, the number of data points that can be collected is quite large. Not every column can be used with every event, but the complete set of allowed combinations is over 4,000. Thinking about memory utilization to hold all this data and the processor time needed to create it, you might be interested in how SQL Server manages to run efficiently while generating so much information. The answer is that SQL Server doesn’t actually collect any data until someone asks for it—instead, the model is to selectively enable collection only as necessary.

Internal Trace Components

The central component of the SQL Trace architecture is the *trace controller,* which is a shared resource that manages all traces created by any consumer. Throughout the database engine are various *event producers*; for example, they are found in the query processor, lock manager, and cache manager. Each of these producers is responsible for generating events that pertain to certain categories of server activity, but each of the producers is disabled by default and therefore generates no data. When a user requests that a trace be started for a certain event, a global bitmap in the trace controller is updated, letting the event producer know that at least one trace is listening, and causing the event to begin firing. Managed along with this bitmap is a secondary list of which traces are monitoring which events.

Once an event fires, its data is routed into a global event sink, which queues the event data for distribution to each trace that is actively listening. The trace controller routes the data to each listening trace based on its internal list of traces and watched events. In addition to the trace controller’s own lists, each individual trace keeps track of which events it is monitoring, along with which columns are actually being used, as well as what filters are in place. The event data returned by the trace controller to each trace is filtered, and the data columns are trimmed down as necessary, before the data is routed to an *I/O provider.*

Trace I/O Providers

The trace I/O providers are what actually send the data along to its final destination. The available output formats for trace data are either a file on the database server (or a network share) or a rowset to a client. Both providers use internal buffers to ensure that if the data is not consumed quickly enough (that is, written to disk or read from the rowset) that it will be queued. However, there is a big difference in how the providers handle a situation in which the queue grows beyond a manageable size.

The *file provider* is designed with a guarantee that no event data will be lost. To make this work even if an I/O slowdown or stall occurs, the internal buffers begin to fill if disk writes are not occurring quickly enough. Once the buffers fill up, threads sending event data to the trace begin waiting for buffer space to free up. To avoid threads waiting on trace buffers, it is imperative to ensure that tracing is performed using a sufficiently fast disk system. To monitor for these waits, watch the SQLTRACE_LOCK and IO_COMPLETION wait types.

The *rowset provider,* on the other hand, is not designed to make any data loss guarantees. If data is not being consumed quickly enough and its internal buffers fill, it waits up to 20 seconds before it begins jettisoning events to free buffers and get things moving. The SQL Server Profiler client tool sends a special error message if events are getting dropped, but you can also find out if you’re headed in that direction by monitoring the TRACEWRITE wait type in SQL Server, which is incremented as threads are waiting for buffers to free up.

A background trace management thread is also started whenever at least one trace is active on the server. This background thread is responsible for flushing file provider buffers (which is done every four seconds), in addition to closing rowset-based traces that are considered to be expired (this occurs if a trace has been dropping events for more than 10 minutes). By flushing the file provider buffers only occasionally rather than writing the data to disk every time an event is collected, SQL Server can take advantage of large block writes, dramatically reducing the overhead of tracing, especially on extremely active servers.

A common question asked by DBAs new to SQL Server is why no provider exists that can write trace data directly to a table. The reason for this limitation is the amount of overhead that would be required for such activity. Because a table does not support large block writes, SQL Server would have to write the event data row by row. The performance degradation caused by event consumption would require either dropping a lot of events or, if a lossless guarantee were enforced, causing a lot of blocking to occur. Neither scenario is especially palatable, so SQL Server simply does not provide this ability. However, as we will see later in the chapter, it is easy enough to load the data into a table either during or after tracing, so this is not much of a limitation.

Security and Permissions

Tracing can expose a lot of information about not only the state of the server, but also the data sent to and returned from the database engine by users. The ability to monitor individual queries down to the batch or even query plan level is at once both powerful and worrisome; even exposure of stored procedure input arguments can give an attacker a lot of information about the data in your database.

To protect SQL Trace from users that should not be able to view the data it exposes, versions of SQL Server prior to SQL Server 2005 allowed only administrative users (members of the *sysadmin* fixed server role) access to start traces. That restriction proved a bit too inflexible for many development teams, and as a result, it has been loosened.

ALTER TRACE Permission

Starting with SQL Server 2005, a new permission exists, called ALTER TRACE. This is a server-level permission (granted to a login principal), and allows access to start, stop, or modify a trace, in addition to providing the ability to generate user-defined events.

> **Tip** Keep in mind that the ALTER TRACE permission is granted at the server level, and access is at the server level; if a user can start a trace, he or she can retrieve event data no matter what database the event was generated in. The inclusion of this permission in SQL Server is a great step in the right direction for handling situations in which developers might need to run traces on production systems to debug application issues, but it’s important not to grant this permission too lightly. It’s still a potential security threat, even if it’s not nearly as severe as giving someone full *sysadmin* access.

To grant ALTER TRACE permission to a login, use the *GRANT* statement as follows (in this example, the permission is granted to a server principal called “Jane”):

```
GRANT ALTER TRACE TO Jane;
```

Protecting Sensitive Event Data

In addition to being locked down so that only certain users can use SQL Trace, the tracing engine itself has a couple of built-in security features to keep unwanted eyes—including those with access to trace—from viewing private information. SQL Trace automatically omits data if an event contains a call to a password-related stored procedure or statement. For example, a call to *CREATE LOGIN* that includes the WITH PASSWORD option is blanked out by SQL Trace.

> **Note** In versions of SQL Server before SQL Server 2005, SQL Trace automatically blanked out a query event if the string *sp_password* was found anywhere in the text of the query. This feature has been removed in SQL Server 2005 and SQL Server 2008, and you should not depend on it to protect your intellectual capital.

Another security feature of SQL Trace is knowledge of encrypted modules. SQL Trace does not return statement text or query plans generated within an encrypted stored procedure, user-defined function, or view. Again, this helps to safeguard especially sensitive data even from users who should have access to see traces.

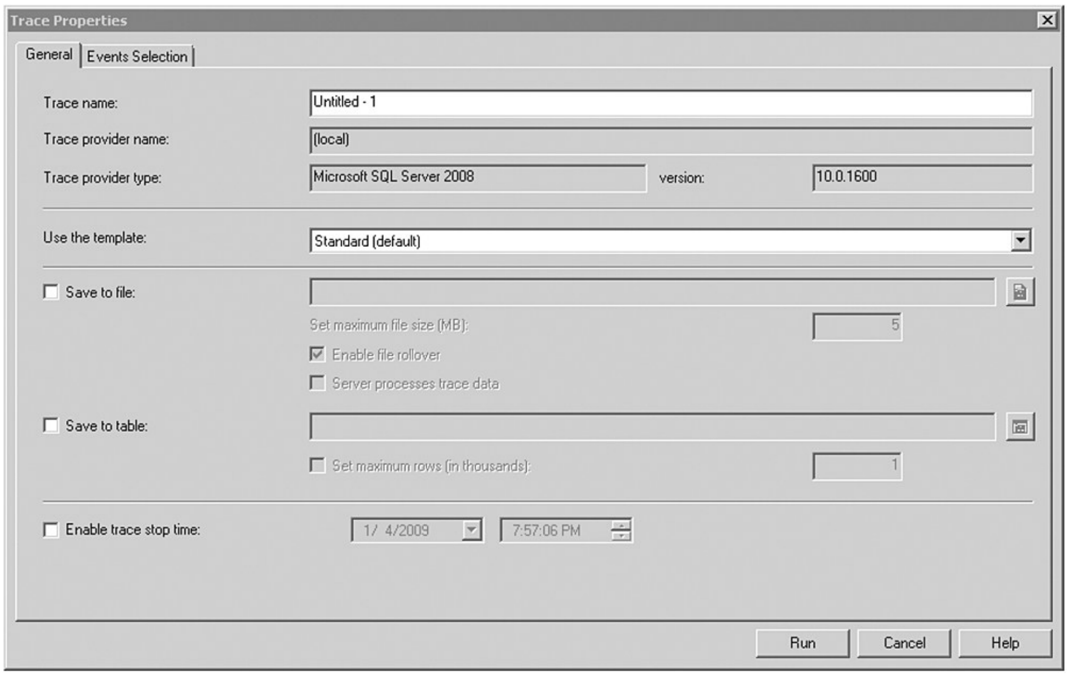
Getting Started: Profiler

SQL Server 2008 ships with Profiler, a powerful user interface tool that can be used to create, manipulate, and manage traces. This tool is the primary starting point for most tracing activity, and thanks to the ease with which it can help you get traces up and running, it is perhaps the most important SQL Server component available for quickly troubleshooting database issues. Profiler also adds a few features to the toolset that are not made possible by SQL Trace itself. This section discusses those features in addition to the base tracing capabilities.

Profiler Basics

The Profiler tool can be found in the Performance Tools subfolder of the SQL Server 2008 Start Menu folder (which you get to by clicking Start and selecting All Programs, SQL Server 2008, Performance Tools, SQL Server Profiler). Once the tool is started, you see a blank screen. Click File, New Trace…and connect to a SQL Server instance. You are shown a Trace Properties dialog box with two tabs, General and Events Selection.

The General tab, shown in Figure 2-2, allows you to control how the trace is processed by the consumer. The default setting is to use the rowset provider, displaying the events in real time in the SQL Server Profiler window. Also available are options to save the events to a file (on either the server or the client), or to a table. However, we generally recommend that you avoid these options on a busy server.



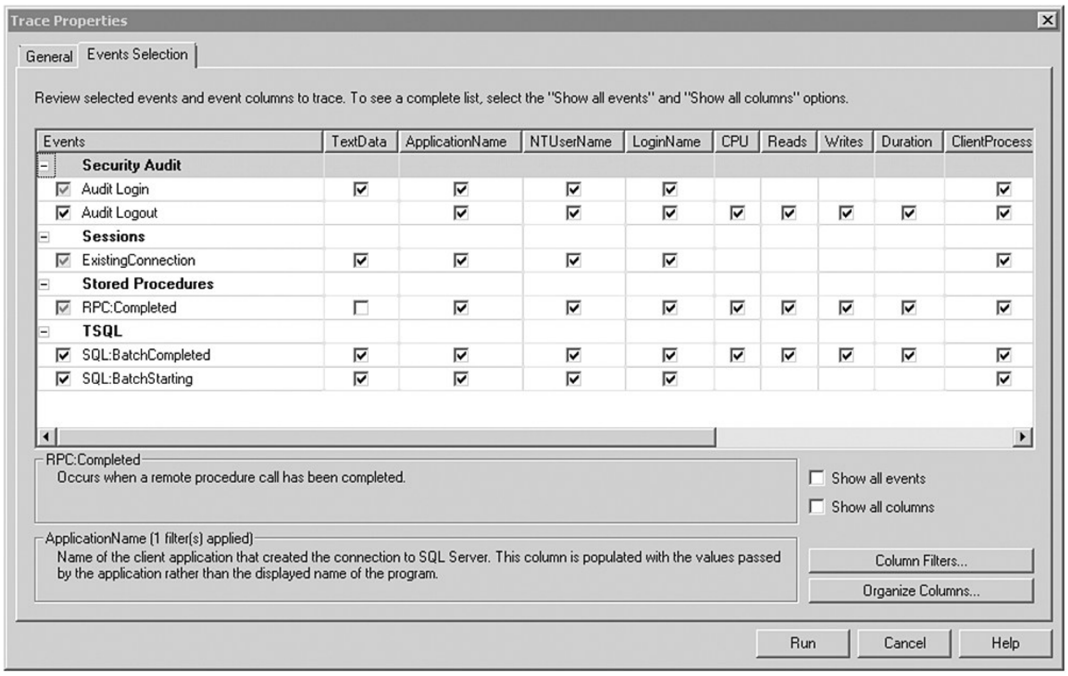
Trace name: Untitled - 1
Trace provider name: (local)
Trace provider type: Microsoft SQL Server 2008 version: 10.0.1600
Use the template: Standard (default)
☐ Save to file:
Set maximum file size (MB): 5
☑ Enable file rollover
☐ Server processes trace data
☐ Save to table:
☐ Set maximum rows (in thousands): 1
☐ Enable trace stop time: 1/ 4/2009 7:57:06 PM

Figure 2-2: Choosing the I/O provider for the trace

When you ask Profiler to save the events to a server-side file (by selecting the Server Processes Trace Data option), it actually starts two equivalent traces, one using the rowset provider and the other using the file provider. Having two traces means twice as much overhead, and that is generally not a good idea. See the section entitled "Server-Side Tracing and Collection," later in this chapter for information, on how to set up a trace using the file provider, which allows you to save to a server-side file efficiently. Saving to a client-side file does not use the file provider at all. Rather, the data is routed to the Profiler tool via the rowset provider and then saved from there to a file. This is more efficient than using Profiler to write to a server-side file, but you do incur network bandwidth because of using the rowset provider, and you also do not get the benefit of the lossless guarantee that the file provider offers.

Note Seeing the Save To Table option, you might wonder why we stated earlier in this chapter that tracing directly to a table is not possible in SQL Trace. The fact is that SQL Trace exposes no table output provider. Instead, when you use this option, the Profiler tool uses the rowset provider and routes the data back into a table. If the table you save to is on the same server you're tracing, you can create quite a large amount of server overhead and bandwidth utilization, so if you must use this option we recommend saving the data to a table on a different server. Profiler also provides an option to save the data to a table *after* you're done tracing, and this is a much more scalable choice in most scenarios.

The Events Selection tab, shown in Figure 2-3, is where you'll spend most of your time configuring traces in Profiler. This tab allows you to select events that you'd like to trace, along with associated data columns. The default options, shown in Figure 2-3, collect data about any connections that exist when the trace starts (the *ExistingConnection* event) when a login or logout occurs (the *Audit Login* and *Audit Logout* events), when remote procedure calls complete (the *RPC:Completed* event), and when T-SQL batches start or complete (the *SQL:BatchCompleted* and *SQL:BatchStarting* events). By default, the complete list of both events and available data columns is hidden. Selecting the Show All Events and Show All Columns check boxes brings the available selections into the UI.



Trace Properties

General | Events Selection

Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration	ClientProcess
Security Audit									
✓ Audit Login	✓	✓	✓	✓					✓
✓ Audit Logout		✓	✓	✓	✓	✓	✓	✓	✓
Sessions									
✓ ExistingConnection	✓	✓	✓	✓					✓
Stored Procedures									
✓ RPC:Completed	☐	✓	✓	✓	✓	✓	✓	✓	✓
TSQL									
✓ SQL:BatchCompleted	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓ SQL:BatchStarting	✓	✓	✓	✓					✓

RPC:Completed
Occurs when a remote procedure call has been completed.

☐ Show all events
☐ Show all columns

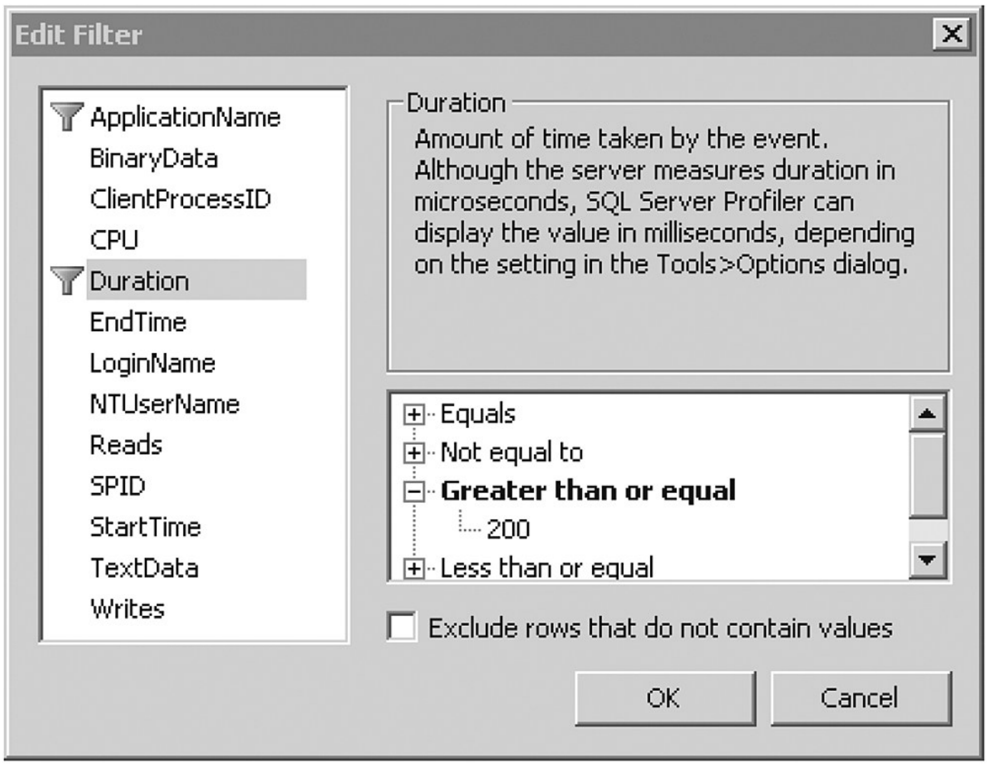
ApplicationName (1 filter(s) applied)
Name of the client application that created the connection to SQL Server. This column is populated with the values passed by the application rather than the displayed name of the program.

Figure 2-3: Choosing event/column combinations for the trace

These default selections are a great starting point and can be used as the basis for a lot of commonly required traces. The simplest questions that DBAs generally answer using SQL Trace are based around query cost and/or duration. What are the longest queries, or the queries that are using the most resources? The default selections can help you answer those types of questions, but on an active server, a huge amount of data would have to be collected, which not only means more work for you to be able to answer your question, but also more work for the server to collect and distribute that much data.

To narrow your scope and help ensure that tracing does not cause performance issues, SQL Trace offers the ability to filter the events based on various criteria. Filtration is exposed in SQL Profiler via the Column Filters… button in the Events Selection tab. Click this button to bring up an Edit Filter dialog box similar to the one shown in Figure 2-4. In this example, we want to see only events with a duration greater than or equal to 200 milliseconds. This is just an arbitrary number; an optimal choice should be discovered iteratively as you build up your knowledge of the tracing requirements for your particular application. Keep raising this number until you mostly receive only the desired events (in this case, those with long durations) from your trace. By working this way, you can isolate the slowest queries in your system easily and quickly.

Tip The list of data columns made available by SQL Profiler for you to use as a filter is the same list of columns available in the outer Events Selection user interface. Make sure to select the Show All Columns check box to ensure that you see a complete list.



Amount of time taken by the event. Although the server measures duration in microseconds, SQL Server Profiler can display the value in milliseconds, depending on the setting in the Tools>Options dialog.

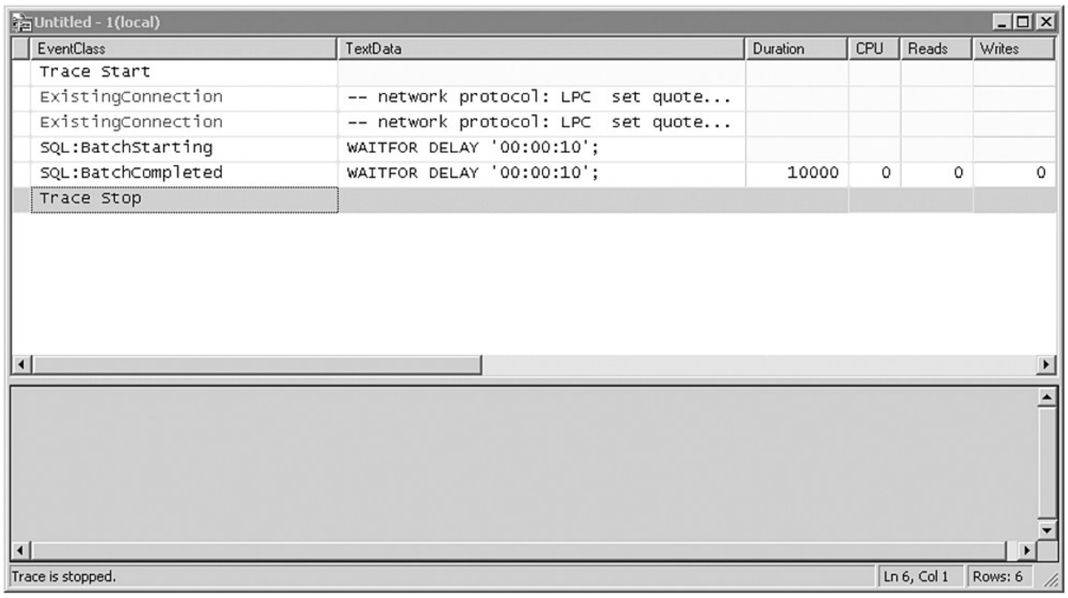
Greater than or equal: 200

☐ Exclude rows that do not contain values

Figure 2-4: Defining a filter for events greater than 200 milliseconds

Once events are selected and filters are defined, the trace can be started. In the Trace Properties dialog box, click Run. Because Profiler uses the rowset provider, data begins streaming back immediately. If you find that data is coming in too quickly for you to be able to read it, consider disabling auto scrolling using the Auto Scroll Window button on the SQL Profiler toolbar.

An important note on filters is that, by default, events that do not produce data for a specific column are not filtered if a trace defines a filter for that column. For example, the *SQL:BatchStarting* event does not produce duration data—the batch is considered to start more or less instantly the moment it is submitted to the server. Figure 2-5 shows a trace that we ran with a filter on the *Duration* column for values greater than 200 milliseconds. Notice that both the *ExistingConnection* and *SQL:BatchStarting* events are still returned even though they lack the *Duration* output column. To modify this behavior, select the Exclude Rows That Do Not Contain Values check box in the Edit Filter dialog box for the column for which you want to change the setting.



EventClass	TextData	Duration	CPU	Reads	Writes
Trace Start					
ExistingConnection	-- network protocol: LPC set quote...				
ExistingConnection	-- network protocol: LPC set quote...				
SQL:BatchStarting	WAITFOR DELAY '00:00:10';				
SQL:BatchCompleted	WAITFOR DELAY '00:00:10';	10000	0	0	0
Trace Stop					

Trace is stopped. Ln 6, Col 1 Rows: 6

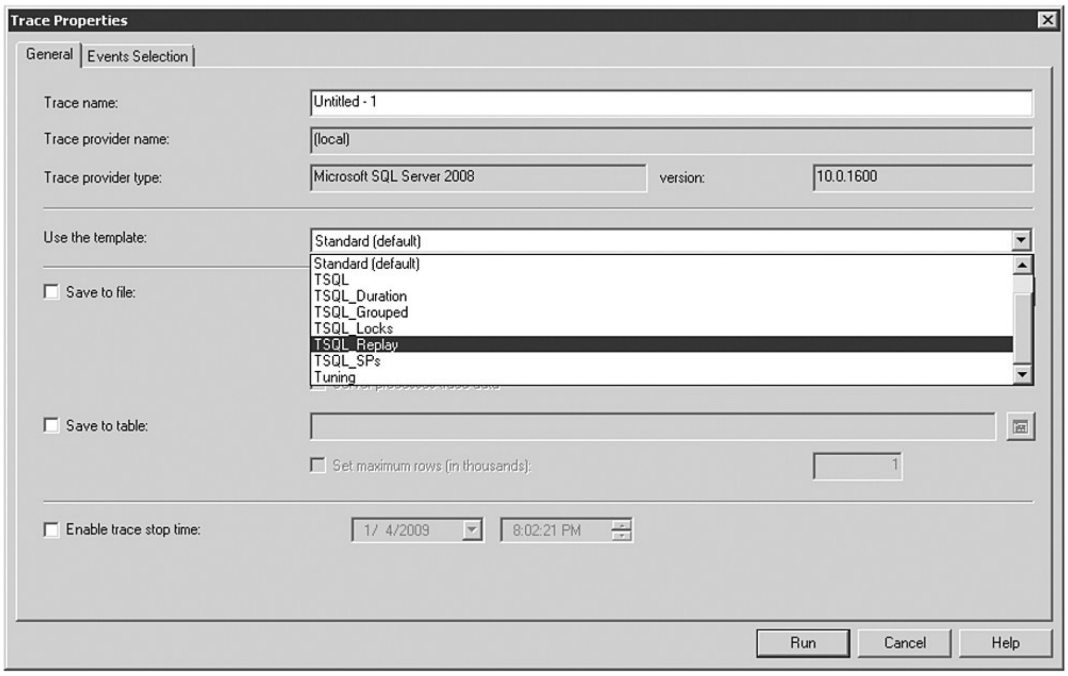
Figure 2-5: By default, trace filters treat empty values as valid for the sake of the filter.

Saving and Replaying Traces

The functionality covered up through this point in the chapter has all been made possible by Profiler merely acting as a wrapper over what SQL Trace provides. In the section entitled "Server-Side Tracing and Collection," later in this chapter, we show you the mechanisms by which Profiler does its work. But first we'll get into the features offered by Profiler that make it more than a simple UI wrapper over the SQL Trace features.

When we discussed the General tab of the Trace Properties window earlier, we glossed over how the default events are actually set: They are included in the standard events template that ships with the product. A *template* is a collection of event and column selections, filters, and other settings that you can save to create reusable trace definitions. This feature can be extremely useful if you do a lot of tracing; reconfiguring the options each time you need them is generally not a good use of your time.

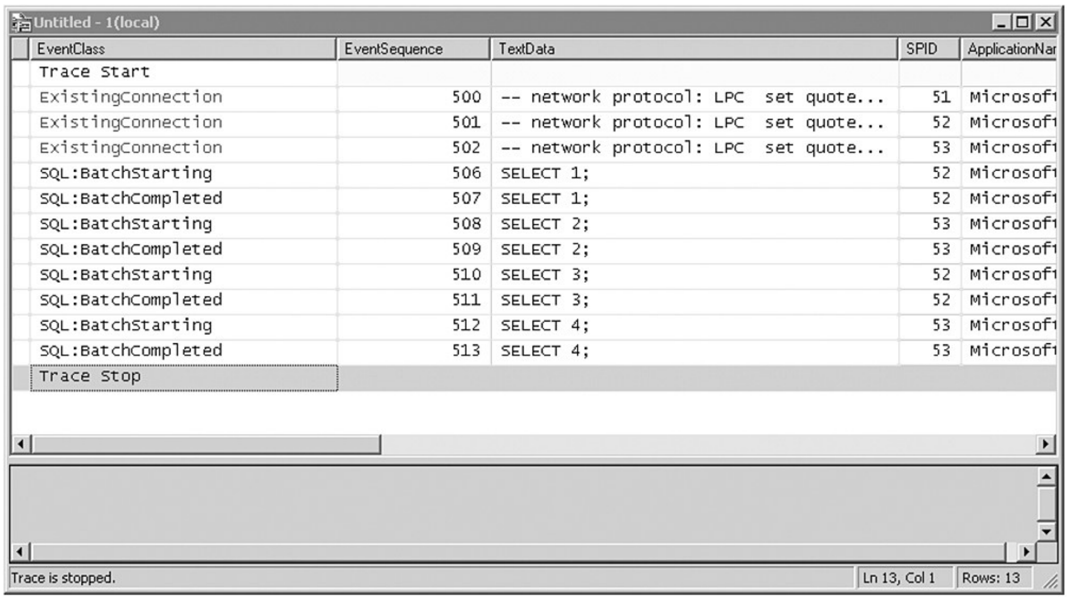
In addition to the ability to save your own templates, Profiler ships with nine predefined templates. Aside from the standard template that we already explored, one of the most important of these is the TSQL_Replay template, which is selected in Figure 2-6. This template selects a variety of columns for 15 different events, each of which are required for Profiler to be able to play back (or replay) a collected trace at a later time. By starting a trace using this template and then saving the trace data once collection is complete, you can do things such as use a trace as a test harness for reproducing a specific problem that might occur when certain stored procedures are called in the correct order.



Trace name: Untitled - 1
Trace provider name: (local)
Trace provider type: Microsoft SQL Server 2008 version: 10.0.1600
Use the template: Standard (default)
Standard (default)
TSQL
TSQL_Duration
TSQL_Grouped
TSQL_Locks
TSQL_Replay
TSQL_SPs
Tuning
1
1/ 4/2009
8:02:21 PM

Figure 2-6: Selecting the TSQL_Replay template

To illustrate this functionality, we started a new trace using the TSQL_Replay template and sent two batches from each of two connections, as shown in Figure 2-7. The first SPID (53, in this case) selected 1, and then the second SPID (54) selected 2. Back to SPID 53, which selected 3, and then finally back to SPID 54, which selected 4. The most interesting thing to note in the figure is the second column, *EventSequence*. This column can be thought of almost like the *IDENTITY* property for a table. Its value is incremented globally, as events are recorded by the trace controller, to create a single representation of the order in which events occurred in the server. This avoids problems that might occur when ordering by StartTime/EndTime (also in the trace, but not shown in Figure 2-7), as there will be no ties—the *EventSequence* is unique for every trace. The number is a 64-bit integer, and it is reset whenever the server is restarted, so it is unlikely that you can ever trace enough to run it beyond its range.



EventClass	EventSequence	TextData	SPID	ApplicationNar
Trace Start				
ExistingConnection	500	-- network protocol: LPC set quote...	51	Microsof
ExistingConnection	501	-- network protocol: LPC set quote...	52	Microsof
ExistingConnection	502	-- network protocol: LPC set quote...	53	Microsof
SQL:BatchStarting	506	SELECT 1;	52	Microsof
SQL:BatchCompleted	507	SELECT 1;	52	Microsof
SQL:BatchStarting	508	SELECT 2;	53	Microsof
SQL:BatchCompleted	509	SELECT 2;	53	Microsof
SQL:BatchStarting	510	SELECT 3;	52	Microsof
SQL:BatchCompleted	511	SELECT 3;	52	Microsof
SQL:BatchStarting	512	SELECT 4;	53	Microsof
SQL:BatchCompleted	513	SELECT 4;	53	Microsof
Trace Stop				

```
Trace is stopped.                    Ln 13, Col 1    Rows: 13
```

Figure 2-7: Two SPIDs sending interleaved batches

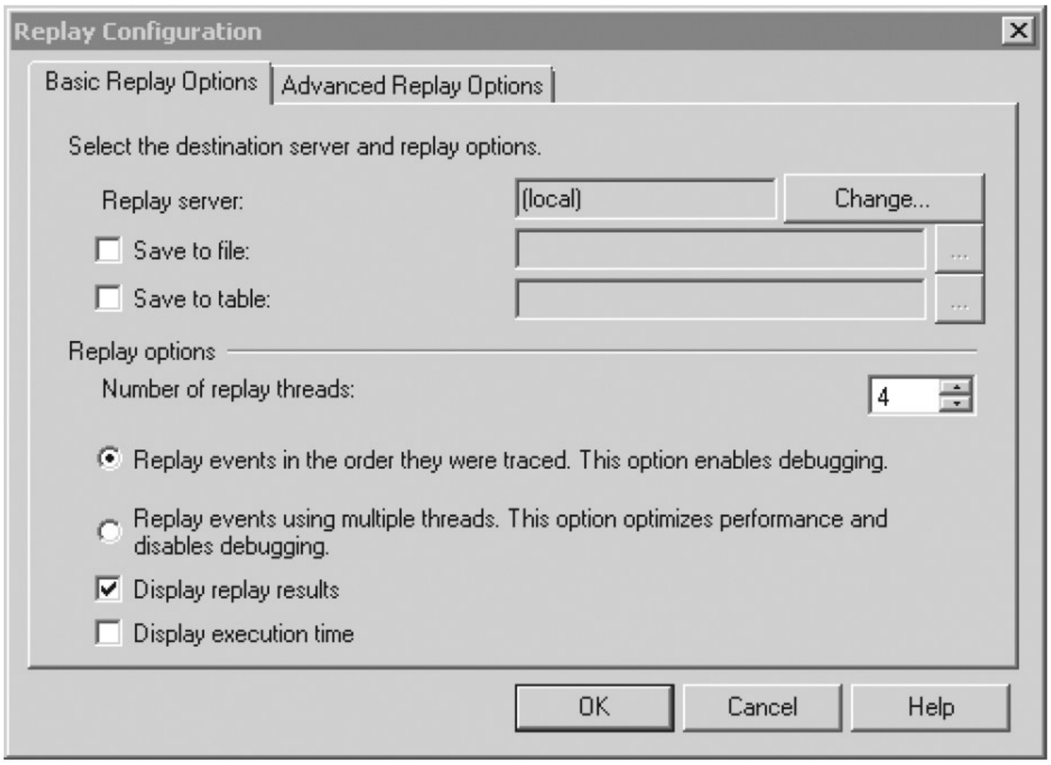
Once the trace data has been collected, it must be saved and then reopened before a replay can begin. Profiler offers the following options for saving trace data, which are available from the File menu:

- The Trace File option is used to save the data to a file formatted using a proprietary binary format. This is generally the fastest way to save the data, and it is also the smallest in terms of bytes on disk.
- The Trace Table option is used to save the data to a new or previously created table in a database of your choosing. This option is useful if you need to manipulate or report on the data using T-SQL.
- The Trace XML File option saves the data to a text file formatted as XML.
- The Trace XML File For Replay option also saves the data to an XML text file, but only those events and columns needed for replay functionality are saved.

Any of these formats can be used as a basis from which to replay a trace, so long as you've collected all the required events and columns needed to do a replay (guaranteed when you use the TSQL_Replay template). We generally recommend using the binary file format as a starting point and saving to a table if manipulation using T-SQL is necessary. For instance, you might want to create a complex query that finds the top queries that use certain tables; something like that would be beyond the abilities of Profiler. With regard to the XML file formats, so far I have not found much use for them. But as more third-party tools hit the market that can use trace data, we may see more use cases.

Once the data has been saved to a file or table, the original trace window can be closed and the file or table reopened via the File menu in the Profiler tool. Once a trace is reopened in this way, a Replay menu appears on the Profiler toolbar, allowing you to start replaying the trace, stop the replay, or set a breakpoint—which is useful when you want to test only a small portion of a larger trace.

After clicking Start in Profiler, you are asked to connect to a server—either the server from which you did the collection, or another server if you want to replay the same trace somewhere else. After connecting, the Replay Configuration dialog box shown in Figure 2-8 is presented. The Basic Replay Options tab allows you to save results of the trace in addition to modifying how the trace is played back.



Select the destination server and replay options.

Replay server: (local)

[ ] Save to file:

[ ] Save to table:

Replay options

Number of replay threads: 4

(●) Replay events in the order they were traced. This option enables debugging.

( ) Replay events using multiple threads. This option optimizes performance and disables debugging.

[✓] Display replay results

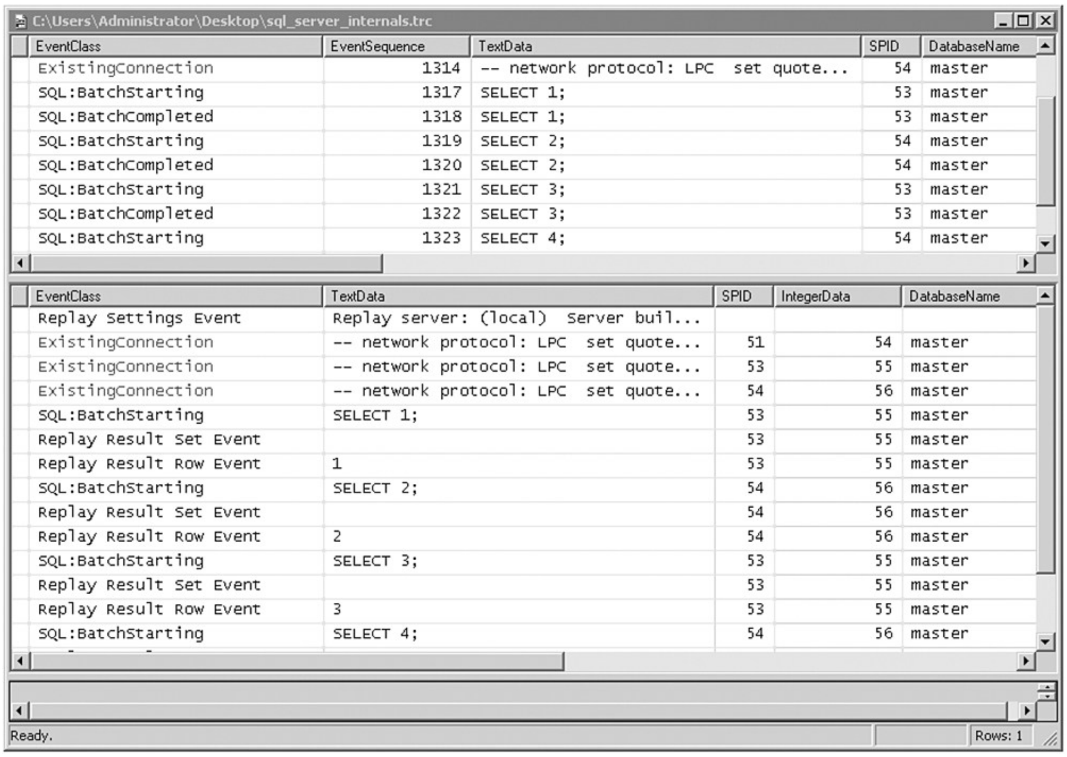
[ ] Display execution time

Figure 2-8: The Replay Configuration dialog box

During the course of the replay, the same events used to produce the trace being replayed are traced from the server on which you replay. The Save To File and Save To Table options are used for a client-side save. No server-side option exists for saving playback results.

The Replay Options pane of the Replay Configurations dialog box is a bit confusing as worded. No matter which option you select, the trace is replayed on multiple threads, corresponding to at most the number you selected in the Number Of Replay Threads drop-down list. However, selecting the Replay Events In The Order They Were Traced option ensures that all events are played back in exactly the order in which they occurred, as based upon the *EventSequence* column. Multiple threads are still used to simulate multiple SPIDs. Selecting the Replay Events Using Multiple Threads option, on the other hand, allows Profiler to rearrange the order in which each SPID starts to execute events, in order to enhance playback performance. Within a given SPID, however, the order of events remains consistent with the *EventSequence*.

To illustrate this difference, we replayed the trace shown in Figure 2-7 twice, each using a different replay option. Figure 2-9 shows the result of the Replay In Order option, whereas Figure 2-10 shows the result of the Multiple Threads option. In Figure 2-9, the results show that the batches were started and completed in exactly the same order in which they were originally traced, whereas in Figure 2-10 the two participating SPIDs have had all their events grouped together rather than interleaved.



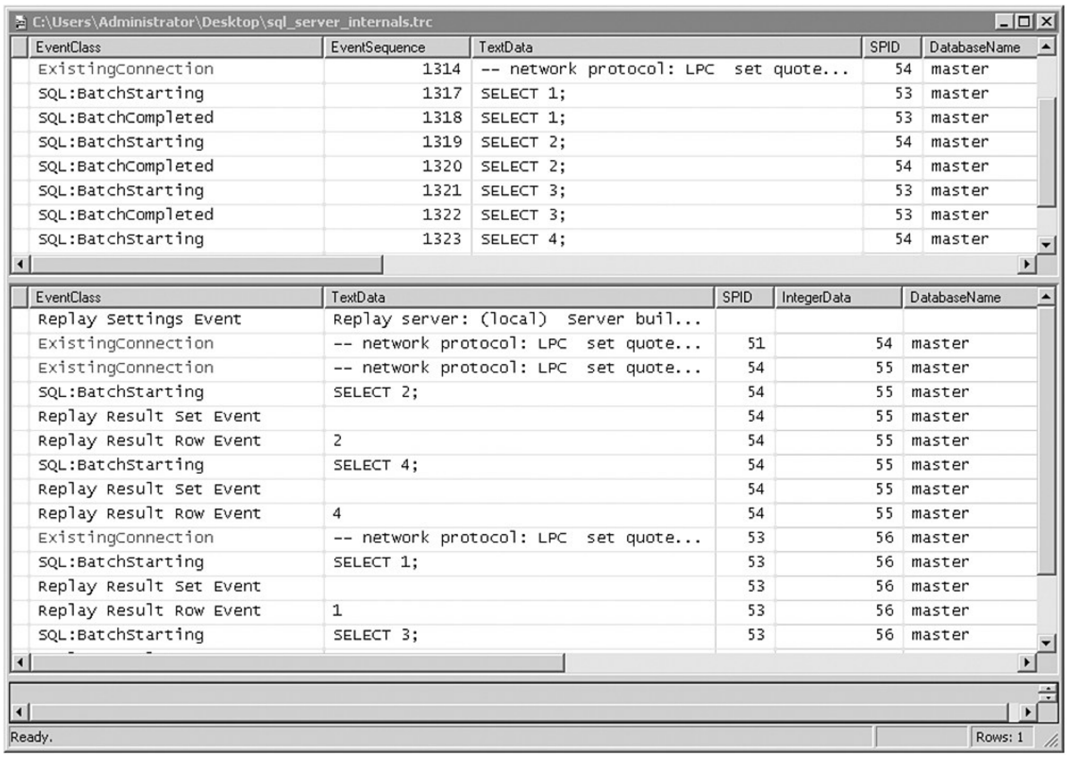
C:\Users\Administrator\Desktop\sql_server_internals.trc

EventClass	EventSequence	TextData	SPID	DatabaseName
ExistingConnection	1314	-- network protocol: LPC set quote...	54	master
SQL:BatchStarting	1317	SELECT 1;	53	master
SQL:BatchCompleted	1318	SELECT 1;	53	master
SQL:BatchStarting	1319	SELECT 2;	54	master
SQL:BatchCompleted	1320	SELECT 2;	54	master
SQL:BatchStarting	1321	SELECT 3;	53	master
SQL:BatchCompleted	1322	SELECT 3;	53	master
SQL:BatchStarting	1323	SELECT 4;	54	master

EventClass	TextData	SPID	IntegerData	DatabaseName
Replay Settings Event	Replay server: (local) Server buil...			
ExistingConnection	-- network protocol: LPC set quote...	51	54	master
ExistingConnection	-- network protocol: LPC set quote...	53	55	master
ExistingConnection	-- network protocol: LPC set quote...	54	56	master
SQL:BatchStarting	SELECT 1;	53	55	master
Replay Result Set Event		53	55	master
Replay Result Row Event	1	53	55	master
SQL:BatchStarting	SELECT 2;	54	56	master
Replay Result Set Event		54	56	master
Replay Result Row Event	2	54	56	master
SQL:BatchStarting	SELECT 3;	53	55	master
Replay Result Set Event		53	55	master
Replay Result Row Event	3	53	55	master
SQL:BatchStarting	SELECT 4;	54	56	master

Ready. Rows: 1

Figure 2-9: Replay using the Replay In Order option



C:\Users\Administrator\Desktop\sql_server_internals.trc

EventClass	EventSequence	TextData	SPID	DatabaseName
ExistingConnection	1314	-- network protocol: LPC set quote...	54	master
SQL:BatchStarting	1317	SELECT 1;	53	master
SQL:BatchCompleted	1318	SELECT 1;	53	master
SQL:BatchStarting	1319	SELECT 2;	54	master
SQL:BatchCompleted	1320	SELECT 2;	54	master
SQL:BatchStarting	1321	SELECT 3;	53	master
SQL:BatchCompleted	1322	SELECT 3;	53	master
SQL:BatchStarting	1323	SELECT 4;	54	master

EventClass	TextData	SPID	IntegerData	DatabaseName
Replay Settings Event	Replay server: (local) Server buil...			
ExistingConnection	-- network protocol: LPC set quote...	51	54	master
ExistingConnection	-- network protocol: LPC set quote...	54	55	master
SQL:BatchStarting	SELECT 2;	54	55	master
Replay Result Set Event		54	55	master
Replay Result Row Event	2	54	55	master
SQL:BatchStarting	SELECT 4;	54	55	master
Replay Result Set Event		54	55	master
Replay Result Row Event	4	54	55	master
ExistingConnection	-- network protocol: LPC set quote...	53	56	master
SQL:BatchStarting	SELECT 1;	53	56	master
Replay Result Set Event		53	56	master
Replay Result Row Event	1	53	56	master
SQL:BatchStarting	SELECT 3;	53	56	master

Ready. Rows: 1

Figure 2-10: Replay using the Multiple Threads option

The Multiple Threads option can be useful if you need to replay a lot of trace data where each SPID has no dependency upon other SPIDs. For example, this might be done to simulate, on a test server, a workload captured from a production system. On the other hand, the Replay In Order option is useful if you need to ensure that you can duplicate the specific conditions that occurred during the trace. For example, this might apply when debugging a deadlock or blocking condition that results from specific interactions of multiple threads accessing the same data.

Profiler is a full-featured tool that provides extensive support for both tracing and doing simple work with trace data, but if you need to do advanced queries against your collected data or run traces against extremely active production systems, Profiler falls short of the requirements. Again, Profiler is essentially nothing more than a wrapper over functionality provided within the database engine, and instead of using it for all stages of the trace lifestyle, we can exploit the tool directly to increase flexibility in some cases. In the following section, you learn how Profiler works with the database engine to start, stop, and manage traces, and how you can harness the same tools for your needs.

Server-Side Tracing and Collection

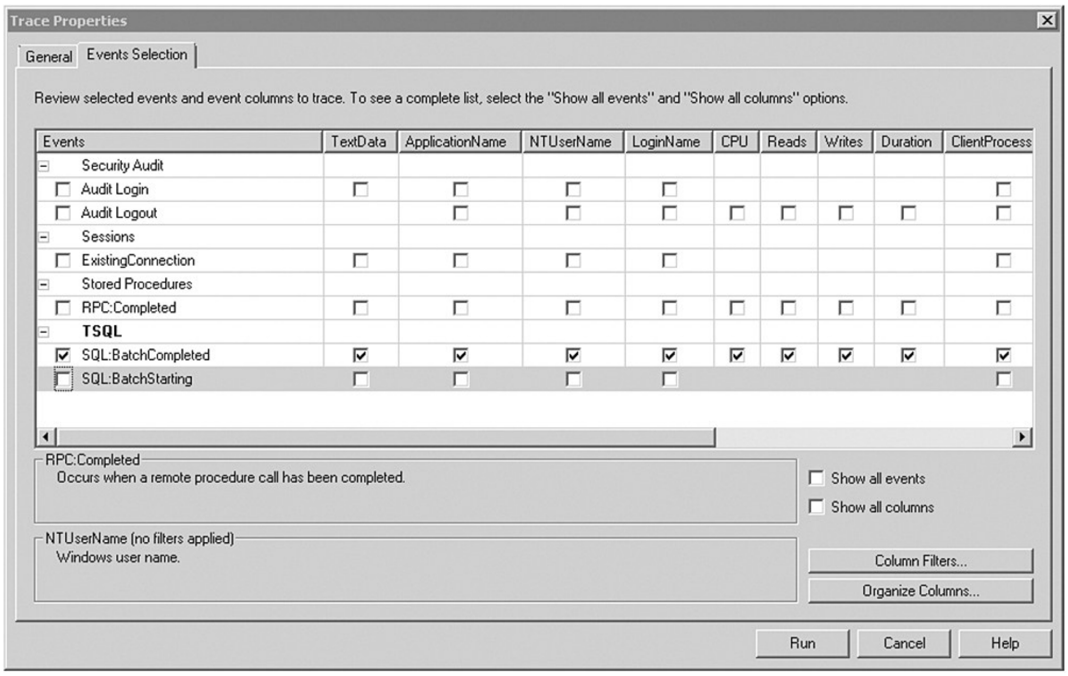
Behind its nice user interface, Profiler is nothing more than a fairly lightweight wrapper over a handful of system stored procedures that expose the true functionality of SQL Trace. In this section, we explore which stored procedures are used and how to harness SQL Server Profiler as a scripting tool rather than a tracing interface.

The following system stored procedures are used to define and manage traces:

- *sp_trace_create* is used to define a trace and specify an output file location as well as other options that I'll cover in the coming pages. This stored procedure returns a handle to the created trace, in the form of an integer trace ID.
- *sp_trace_setevent* is used to add event/column combinations to traces based on the trace ID, as well as to remove them, if necessary, from traces in which they have already been defined.
- *sp_trace_setfilter* is used to define event filters based on trace columns.
- *sp_trace_setstatus* is called to turn on a trace, to stop a trace, and to delete a trace definition once you're done with it. Traces can be started and stopped multiple times over their lifespan.

Scripting Server-Side Traces

Rather than delve directly into the syntax specifications for each of the stored procedures— all which are documented in detail in *SQL Server Books Online*—it is a bit more interesting to observe them in action. To begin, open up SQL Server Profiler, start a new trace with the default template, and clear all the events except for *SQL:BatchCompleted*, as shown in Figure 2-11.



Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration	ClientProcess
Security Audit									
☐ Audit Login	☐	☐	☐	☐					☐
☐ Audit Logout		☐	☐	☐	☐	☐	☐	☐	☐
Sessions									
☐ ExistingConnection	☐	☐	☐	☐					☐
Stored Procedures									
☐ RPC:Completed	☐	☐	☐	☐	☐	☐	☐	☐	☐
TSQL									
☑ SQL:BatchCompleted	☑	☑	☑	☑	☑	☑	☑	☑	☑
☐ SQL:BatchStarting	☐	☐	☐	☐					☐

RPC:Completed
Occurs when a remote procedure call has been completed.

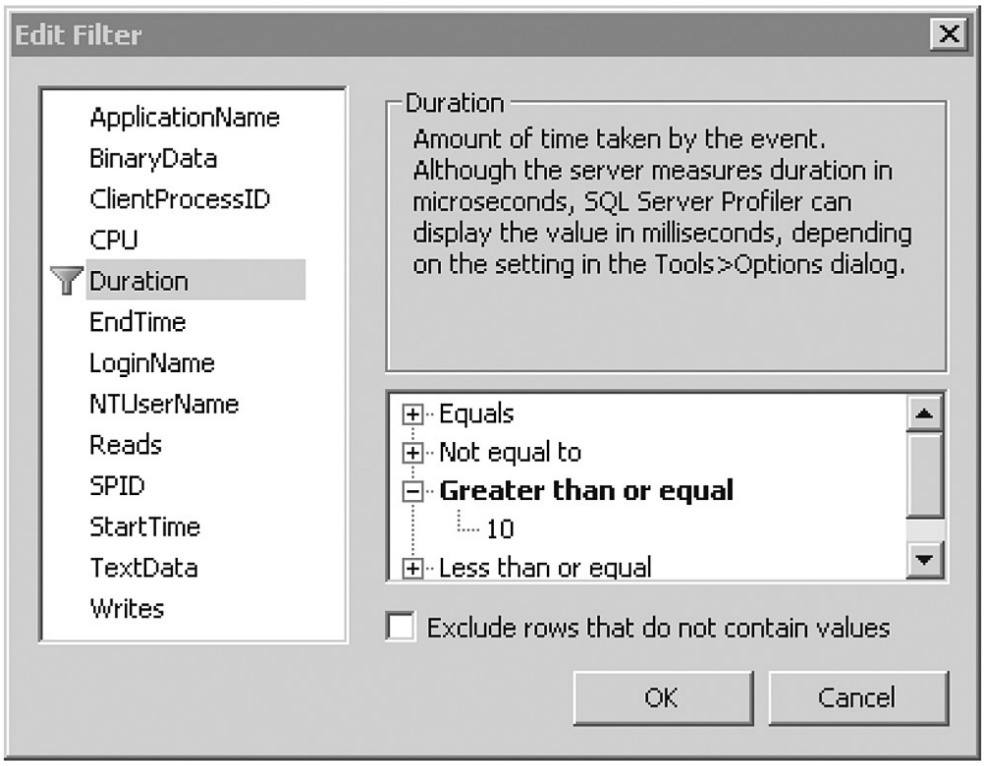
NTUserName (no filters applied)
Windows user name.

☐ Show all events
☐ Show all columns

Figure 2-11: Trace events with only SQL:BatchCompleted selected

Next, remove the default filter on the *ApplicationName* column (set to not pick up SQL Server Profiler events), and add a filter on *Duration* for greater than or equal to 10 milliseconds, as shown in Figure 2-12. Once you're finished, click Run to start the trace, then immediately click Stop. Because of the workflow required by the SQL Profiler user interface, you must actually start a trace before you can script it. On the File menu, select Export, Script Trace Definition, and For SQL Server 2005 - 2008. This will produce a script similar to the following (edited for brevity and readability):

```
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

exec @rc = sp_trace_create
    @TraceID output,
    0,
    N'InsertFileNameHere',
    @maxfilesize,
    NULL
if (@rc != 0) goto finish

-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 12, 15, @on
exec sp_trace_setevent @TraceID, 12, 16, @on
exec sp_trace_setevent @TraceID, 12, 1, @on
exec sp_trace_setevent @TraceID, 12, 9, @on
exec sp_trace_setevent @TraceID, 12, 17, @on
exec sp_trace_setevent @TraceID, 12, 6, @on
exec sp_trace_setevent @TraceID, 12, 10, @on
exec sp_trace_setevent @TraceID, 12, 14, @on
exec sp_trace_setevent @TraceID, 12, 18, @on
exec sp_trace_setevent @TraceID, 12, 11, @on
exec sp_trace_setevent @TraceID, 12, 12, @on
exec sp_trace_setevent @TraceID, 12, 13, @on
-- Set the Filters
declare @bigintfilter bigint

set @bigintfilter = 10000
exec sp_trace_setfilter @TraceID, 13, 0, 4, @bigintfilter

-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1

-- display trace id for future references
select TraceID=@TraceID

finish:
go
```


Edit Filter

ApplicationName
BinaryData
ClientProcessID
CPU
Duration
EndTime
LoginName
NTUserName
Reads
SPID
StartTime
TextData
Writes

Duration
Amount of time taken by the event. Although the server measures duration in microseconds, SQL Server Profiler can display the value in milliseconds, depending on the setting in the Tools>Options dialog.

Equals
Not equal to
Greater than or equal
10
Less than or equal

Exclude rows that do not contain values

OK
Cancel

Figure 2-12: Filter on Duration set to greater than or equal to 10 milliseconds

Note An option also exists to script the trace definition for SQL Server 2000. The SQL Trace stored procedures did not change much between SQL Server 2000 and SQL Server 2005—and it did not change at all between SQL Server 2005 and SQL Server 2008—but several new events and columns were added to the product. Scripting for SQL Server 2000 simply drops from the script any events that are not backward-compatible.

This script is an extremely simple yet complete definition of a trace that uses the file provider. A couple of placeholder values need to be modified, but for the most part, it is totally functional. Given the complexity of working directly with the SQL Trace stored procedures, we generally define a trace using SQL Profiler's user interface, and then script it and work from there. This way, you get the best of both worlds: ease of use combined with the efficiency of server-side traces using the file provider.

This script does a few different things, so we will walk through each stage:

1. The script defines a few variables to be used in the process. The *@rc* variable is used to get a return code from *sp_trace_create*. The *@TraceID* variable holds the handle to the newly created trace. Finally, the *@maxfilesize* variable defines the maximum size (in megabytes) per trace file. When running server-side traces, the file provider can be configured to create *rollover files* automatically as the primary trace file fills up. This can be useful if you're working on a drive with limited space, as you can move previously filled files to another device. In addition, smaller files can make it easier to manipulate subsets of the collected data. Finally, rollover files also have their utility in high-load scenarios. However, most of the time these are not necessary, and a value of 5 is a bit small for the majority of scenarios.

2. The script calls the *sp_trace_create* stored procedure, which initializes—but does not start—the trace. The parameters specified here are the output parameter for the trace ID of the newly created trace; 0 for the options parameter—meaning that rollover files should not be used; a placeholder for a server-side file path, which should be changed before using this script; the maximum file size as defined by the *@maxfilesize* variable; and NULL for the stop date—this trace only stops when it is told to. Note that there is also a final parameter in *sp_trace_create*, which allows the user to set the maximum number of rollover files. This parameter, called *@filecount* in the *sp_trace_create* documentation, was added in SQL Server 2005 and is not added automatically to the trace definition scripts created with the Script Trace Definition option. The @ *filecount* parameter doesn't apply here because the options parameter was set to 0 and no rollover files are created, but it can be useful in many other cases. Note that because rollover files are disabled, if the maximum file size is reached, the trace automatically stops and closes.

   Note The file extension .trc is appended to the file path specified for the output trace file automatically. If you use the .trc extension in your file name (for example, C:\mytrace.trc), the file on disk is C:\mytrace.trc.trc.

3. *sp_trace_setevent* is used to define the event/column combinations used for the trace. In this case, to keep things simple, only event 12—*SQL:BatchCompleted*—is used. One call to *sp_trace_setevent* is required for each event/column combination used in the trace. As an aside, note that the *@on* parameter must be a *bit*. Because numeric literals in SQL Server 2005 and earlier are cast as integers implicitly by default, the local *@on* variable is needed to force the value to be treated appropriately by the stored procedure in those versions.

4. Once events are set, filters are defined. In this case, column 13 (*Duration*) is filtered using the *and* logical operator (the third parameter, with a value of 0) and the *greater than or equal to* comparison operator (the fourth parameter, with a value of 4). The actual value is passed in as the final parameter. Note that it is shown in the script in microseconds; SQL Trace uses microseconds for its durations, although the default standard of time in SQL Profiler is milliseconds. To change the SQL Profiler default, click Tools, Options, and then select the Show Values In Duration Column In Microseconds check box (note that microsecond durations are available in SQL Server 2005 and SQL Server 2008 only).

   Note SQL Trace offers both *and* and *or* logical operators that can be combined if multiple filters are used. However, there is no way to indicate parentheses or other grouping constructs, meaning that the order of operations is limited to left-to-right evaluation. This means that an expression such as *A and B or C and D* is logically evaluated by SQL Trace as (((*A and B) or C) and D*). However, SQL Trace internally breaks the filters into groups based on columns being filtered. So the expression *Column1=10 or Column1=20 and Column3=15 or Column3=25* is actually evaluated as *(Column1=10 or Column1=20) and* (*Column3=15 or Column3=25*). Not only is this somewhat confusing, but it can make certain conditions difficult or impossible to express. Keep in mind that in some cases, you may have to break up your filter criteria and create multiple traces to capture everything the way you intend to.

5. The trace has now been created, event and column combinations set, and filters defined. The final thing to do is actually start tracing. This is done via the call to *sp_trace_setstatus*, with a value of 1 for the second parameter.

Querying Server-Side Trace Metadata

After modifying the file name placeholder appropriately and running the test script on my server, I received a value of 2 for the trace ID. Using a trace ID, you can retrieve a variety of metadata about the trace from the *sys.traces* catalog view, such as is done by the following query:

```
SELECT
    status,
    path,
    max_size,
    buffer_count,
    buffer_size,
    event_count,
    dropped_event_count
FROM sys.traces
WHERE id = 2;
```

This query returns the trace status, which is 1 (started) or 0 (stopped); the server-side path to the trace file (or NULL if the trace is using the rowset provider); the maximum file size (or again, NULL in the case of the rowset provider); information about how many buffers of what size are in use for processing the I/O; the number of events captured; and the number of dropped events (in this case, NULL if your trace is using the file provider).

Note For readers migrating from SQL Server 2000, note that the *sys.traces* view replaces the older *fn_trace_getinfo* function. This older function returns only a small subset of the data returned by the *sys.traces* view, so it's definitely better to use the view going forward.

In addition to the *sys.traces* catalog view, SQL Server ships with a few other views and functions to help derive information about traces running on the server. They are described in the upcoming sections.

fn_trace_geteventinfo This function returns the numeric combinations of events and columns selected for the trace, in a tabular format. The following T-SQL code returns this data for trace ID 2:

```
SELECT *
FROM fn_trace_geteventinfo(2);
```

The output from running this query on the trace created in the preceding script follows:

eventid	columnid
12	1
12	6
12	9
12	10
12	11
12	12
12	13
12	14
12	15
12	16
12	17
12	18

sys.trace_events and sys.trace_columns The numeric representations of trace events and columns are not especially interesting on their own. To be able to query this data properly, a textual representation is necessary. The *sys.trace_events* and *sys.trace_columns* contain not only text describing the events and columns, respectively, but also other information such as data types for the columns and whether they are filterable. Combining these views with the previous query against the *fn_trace_geteventinfo* function, we can get a version of the same output that is much easier to read:

```
SELECT
    e.name AS Event_Name,
    c.name AS Column_Name
FROM fn_trace_geteventinfo(2) ei
JOIN sys.trace_events e ON ei.eventid = e.trace_event_id
JOIN sys.trace_columns c ON ei.columnid = c.trace_column_id;
```

The output from this query follows:

Event_Name	Column_Name
SQL:BatchCompleted	*TextData*
SQL:BatchCompleted	*NTUserName*
SQL:BatchCompleted	*ClientProcessID*
SQL:BatchCompleted	*ApplicationName*
SQL:BatchCompleted	*LoginName*
SQL:BatchCompleted	*SPID*
SQL:BatchCompleted	*Duration*
SQL:BatchCompleted	*StartTime*
SQL:BatchCompleted	*EndTime*
SQL:BatchCompleted	*Reads*
SQL:BatchCompleted	*Writes*
SQL:BatchCompleted	*CPU*

fn_trace_getfilterinfo To get information about which filter values were set for a trace, the *fn_trace_getfilterinfo* function can be used. This function returns the column ID being filtered (which can be joined to the *sys.trace_columns* view for more information), the logical operator, comparison operator, and the value of the filter. The following code shows an example of its use:

```
SELECT
    columnid,
    logical_operator,
    comparison_operator,
    value
FROM fn_trace_getfilterinfo(2);
```

Retrieving Data from Server-Side Traces

Once a trace is started, the obvious next move is to actually read the collected data. This is done using the *fn_trace_gettable* function. This function takes two parameters: The name of the first file from which to read the data, and the maximum number of rollover files to read from (should any exist). The following T-SQL reads the trace file located at C:\sql_server_internals.trc*:*

```
SELECT *
FROM fn_trace_gettable('c:\sql_server_internals.trc', 1);
```

A trace file can be read at any time, even while a trace is actively writing data to it. Note that this is probably not a great idea in most scenarios because it increases disk contention, thereby decreasing the speed with which events can be written to the table and increasing the possibility of blocking. However, in situations in which you're collecting data infrequently—such as when you've filtered for a very specific stored procedure pattern that isn't called often—this is an easy way to find out what you've collected so far.

Because *fn_trace_gettable* is a table-valued function, its uses within T-SQL are virtually limitless. It can be used to formulate queries, or it can be inserted into a table so that indexes can be created. In the latter case, it's probably a good idea to use *SELECT INTO* to take advantage of minimal logging:

```
SELECT *
INTO sql_server_internals
FROM fn_trace_gettable('c:\sql_server_internals.trc', 1);
```

Once the data has been loaded into a table, it can be manipulated any number of ways to troubleshoot or answer questions.

Stopping and Closing Traces

When a trace is first created, it has the status of 0, stopped (or not yet started, in that case). A trace can be brought back to that state at any time using *sp_trace_setstatus*. To set trace ID 2 to a status of stopped, the following T-SQL code is used:

```
EXEC sp_trace_setstatus 2, 0;
```

Aside from the obvious benefit that the trace no longer collects data, there is another perk to doing this: Once the trace is in a stopped state, you can modify the event/column selections and filters using the appropriate stored procedures without re-creating the trace. This can be extremely useful if you need to make only a minor adjustment.

If you are actually finished tracing and do not wish to continue at a later time, you can remove the trace definition from the system altogether by setting its status to 2:

```
EXEC sp_trace_setstatus 2, 2;
```

Tip Trace definitions are removed automatically in the case of a SQL Server service restart, so if you need to run the same trace again later, either save it as a Profiler template or save the script used to start it.

Investigating the Rowset Provider

Most of this section has dealt with how to work with the file provider using server-side traces, but some readers are undoubtedly asking themselves how SQL Server Profiler interfaces with the rowset provider. The rowset provider and its interfaces are completely undocumented. However, because Profiler is doing nothing more than calling stored procedures under the covers, it is not too difficult to find out what's going on. As a matter of fact, you can use a somewhat recursive process: use Profiler to trace activity generated by itself.

A given trace session cannot capture all its own events (the trace won't be running yet when some of them occur), so to see how Profiler works, we need to set up two traces: an initial trace configured to watch for Profiler activity, and a second trace to produce the activity for the first trace to capture. To begin with, open SQL Profiler and create a new trace using the default template. In the Edit Filter dialog box, remove the default Not Like filter on *ApplicationName* and replace it with a Like filter on *ApplicationName* for the string *SQL Server Profiler%*. This filter captures all activity that is produced by any SQL Server Profiler session.

Start that trace, then load up another trace using the default template and start it. The first trace window now fills with calls to the various *sp_trace* stored procedures, fired via *RPC:Completed* events. The first hint that something different happens when using the rowset provider is the call made to *sp_trace_create*:

```
declare @p1 int;
exec sp_trace_create @p1 output,1,NULL,NULL,NULL;
select @p1;
```

The second parameter, used for options, is set to 1, a value not documented in *SQL Server Books Online*. This is the value that turns on the rowset provider. And the remainder of the parameters, which deal with file output, are populated with NULLs.

> **Tip** The *sp_trace_create options* parameter is actually a bit mask—multiple options can be set simultaneously. To do that, simply add up the values for each of the options you want. With only three documented values and one undocumented value, there aren't a whole lot of possible combinations, but it's still something to keep in mind.

Much of the rest of the captured activity looks familiar at this point; you see normal-looking calls to *sp_trace_setevent*, *sp_trace_setfilter*, and *sp_trace_setstatus*. However, to see the complete picture, you must stop the second trace (the one actually generating the trace activity being captured). As soon as the second trace stops, the first trace captures the following *RPC:Completed* event:

```
exec sp_executesql N'exec sp_trace_getdata @P1, 0',N'@P1 int',3;
```

In this case, 3 is the trace ID for the second trace on our system. Given this set of input parameters, the *sp_trace_getdata* stored procedure streams event data back to the caller in a tabular format and does not return until the trace is stopped.

Unfortunately, the tabular format produced by *sp_trace_getdata* is far from recognizable and is not in the standard trace table format. By modifying the previous file-based trace, we can produce a rowset-based trace using the following T-SQL code:

```
declare @rc int
declare @TraceID int

exec @rc = sp_trace_create
    @TraceID output,
    1,
    NULL,
    NULL,
    NULL
if (@rc != 0) goto finish

-- Set the events
declare @on bit
set @on = 1
exec sp_trace_setevent @TraceID, 12, 15, @on
exec sp_trace_setevent @TraceID, 12, 16, @on
exec sp_trace_setevent @TraceID, 12, 1, @on
exec sp_trace_setevent @TraceID, 12, 9, @on
exec sp_trace_setevent @TraceID, 12, 17, @on
exec sp_trace_setevent @TraceID, 12, 6, @on
exec sp_trace_setevent @TraceID, 12, 10, @on
exec sp_trace_setevent @TraceID, 12, 14, @on
exec sp_trace_setevent @TraceID, 12, 18, @on
exec sp_trace_setevent @TraceID, 12, 11, @on
exec sp_trace_setevent @TraceID, 12, 12, @on
exec sp_trace_setevent @TraceID, 12, 13, @on

-- Set the Filters
declare @bigintfilter bigint

set @bigintfilter = 10000
exec sp_trace_setfilter @TraceID, 13, 0, 4, @bigintfilter

-- Set the trace status to start
exec sp_trace_setstatus @TraceID, 1

-- display trace id for future references
select TraceID=@TraceID
```

```
exec sp_executesql
    N'exec sp_trace_getdata @P1, 0',
    N'@P1 int',
    @TraceID

finish:
go
```

Running this code, then issuing a *WAITFOR DELAY '00:00:10'* in another window, produces the following output (truncated and edited for brevity):

ColumnId	Length	Data
65526	6	0xFEFF63000000
14	16	0xD707050002001D001 …
65533	31	0x01010000000300000 …
65532	26	0x0C000100060009000 …
65531	14	0x0D000004080010270 …
65526	6	0xFAFF00000000
65526	6	0x0C000E010000
1	48	0x57004100490054004 …
6	8	0x4100640061006D00
9	4	0xC8130000
10	92	0x4D006900630072006 …

Each of the values in the *columnid* column corresponds to a trace data column ID. The *length* and *data* columns are relatively self-explanatory—*data* is a binary-encoded value that corresponds to the collected column, and *length* is the number of bytes used by the *data* column. Each row of the output coincides with one column of one event. SQL Server Profiler pulls these events from the rowset provider via a call to *sp_trace_getdata* and performs a pivot to produce the human-readable output that we're used to seeing. This is yet another reason that the rowset provider can be less efficient than the file provider—sending so many rows can produce a huge amount of network traffic.

If you do require rowset provider–like behavior for your monitoring needs, luckily you do not need to figure out how to manipulate this data. SQL Server 2008 ships with a series of managed classes in the *Microsoft.SqlServer.Management.Trace* namespace, designed to help with setting up and consuming rowset traces. The use of these classes is beyond the scope of this chapter, but they are well documented in the SQL Server TechCenter and readers should have no trouble figuring out how to exploit what they offer.

Extended Events

As useful as SQL Trace can be for DBAs and developers who need to debug complex scenarios within SQL Server, the fact is that it has some key limitations. First, its column-based architecture makes it difficult to add new events that don't fit nicely into the existing set of output columns. Second, large traces can have a greater impact on system performance than many DBAs prefer. Finally, SQL Trace is a tracing infrastructure only; it cannot be extended into other areas that a general-purpose eventing system can be used for.

The solution to all these problems is Extended Events (XE, XEvents, or X/Events for short, depending on which article or book you happen to be reading—we'll use the XE shorthand for the remainder of this chapter). Unlike SQL Trace, XE is designed as a general eventing system that can be used to fulfill tracing requirements but that also can be used for a variety of other purposes—both internal to the engine and external. Events in XE are not bound to a general set of output columns as are SQL Trace events. Instead, each XE event publishes its data using its own unique schema, making the system as flexible as possible. XE also answers some of the performance problems associated with SQL Trace. The system was engineered from the ground up with performance in mind, and so in most cases, events have minimal impact on overall system performance.

Due to its general nature, XE is much bigger and more complex than SQL Trace, and learning the system requires that DBAs understand a number of new concepts. In addition, because the system is new for SQL Server 2008, there is not yet

UI support in the form of a Profiler or similar tool. Given the steep learning curve, many DBAs may be less than excited about diving in. However, as you will see in the remainder of this chapter, XE is a powerful tool and certainly worth learning today. The next several versions of SQL Server will see XE extended and utilized in a variety of ways, so understanding its foundations today is a good investment for the future.

Components of the XE Infrastructure

The majority of the XE system lives in an overarching layer of SQL Server that is architecturally similar to the role of the SQL operating system (SQLOS). As a general-purpose eventing and tracing system, it must be able to interact with all levels of the SQL Server host process, from the query processing APIs all the way down into the storage engine. To accomplish its goals, XE exposes several types of components that work together to form the complete system.

Packages

Packages are the basic unit within which all other XE objects ship. Each package is a collection of types, predicates, targets, actions, maps, and events—the actual user-configurable components of XE that you work with as you interact with the system. SQL Server 2008 ships with four packages, which can be queried from the *sys.dm_xe_packages* DMV, as in the following example:

```
SELECT *
FROM sys.dm_xe_packages;
```

Packages can interact with one another to avoid having to ship the same code in multiple contexts. For example, if one package exposes a certain action that can be bound to an event, any number of other events in other packages can use it. As a means by which to use this flexibility, Microsoft ships a package called *package0* with SQL Server 2008. This package can be considered the base; it contains objects designed to be used by all the other packages currently shipping with SQL Server, as well as those that might ship in the future.

In addition to *package0*, SQL Server ships with three other packages. The *sqlos* package contains objects designed to help the user interact with the SQLOS system. The *sqlserver* package, on the other hand, contains objects specific to the rest of the SQL Server system. The *SecAudit* package is a bit different; it contains objects designed for the use of SQL Audit, which is an auditing technology built on top of Extended Events. Querying the *sys.dm_xe_packages* DMV, you can see that this package is marked as *private* in the *capabilities_desc* column. This means that non-system consumers can't directly use the objects that it contains.

To see a list of all the objects exposed by the system, query the *sys.dm_xe_objects* DMV:

```
SELECT *
FROM sys.dm_xe_objects;
```

This DMV exposes a couple of key columns important for someone interested in exploring the objects. The *package_guid* column is populated with the same GUIDs that can be found in the *guid* column of the *sys.dm_xe_packages* DMV. The *object_type* column can be used to filter on specific types of objects. And just like *sys.dm_xe_packages*, *sys.dm_xe_objects* exposes a *capabilities_desc* column that is sometimes set to *private* for certain objects that are not available for use by external consumers. There is also a column called *description,* which purports to contain human-readable text describing each object, but this is a work in progress as of SQL Server 2008 RTM, and many of the descriptions are incomplete.

In the following sections, we explore, in detail, each of the object types found in *sys.dm_xe_objects*.

Events

Much like SQL Trace, XE exposes a number of events that fire at various expected times as SQL Server goes about its duties. Also, just like with SQL Trace, various code paths throughout the product have been instrumented with calls that cause the events to fire when appropriate. New users of XE will find almost all the same events that SQL Trace exposes, plus many more. SQL Trace ships with 180 events in SQL Server 2008; XE ships with 254. This number increases for XE because many of the XE events are at a much deeper level than the SQL Trace events. For example, XE includes an event that fires each time a page split occurs. This allows a user to track splits at the query level, something that was impossible to do in previous versions of SQL Server.

The most important differentiator of XE events, compared with those exposed by SQL Trace, is that each event exposes its own output schema. These schemas are exposed in the *sys.dm_xe_object_columns* DMV, which can be queried for a list of output columns as in the following example:

```
SELECT *
FROM sys.dm_xe_object_columns
WHERE
    object_name = 'page_split';
```

In addition to a list of column names and column ordinal positions, this query also returns a list of data types associated with each column. These data types, just like every other object defined within the XE system, are contained within packages and each has its own entry in the *sys.dm_xe_objects* DMV. Columns can be marked *readonly* (per the *column_type* column), in which case they have a value defined in the *column_value* column, or they can be marked as *data,* which means that their values will be populated at run time. The *readonly* columns are metadata, used to store various information including a unique identifier for the type of event that fired and a version number so that different versions of the schema for each event can be independently tracked and used.

One of the handful of *readonly* attributes that is associated with each event is the CHANNEL for the event. This is a reflection of one of the XE design goals, to align with the Event Tracing for Windows (ETW) system. Events in SQL Server 2008 are categorized as Admin, Analytic, Debug, or Operational. The following is a description of each of these event channels:

- *Admin events* are those that are expected to be of most use to systems administrators, and this channel includes events such as error reports and deprecation announcements.
- *Analytic events* are those that fire on a regular basis—potentially thousands of times per second on a busy system—and are designed to be aggregated to support analysis about system performance and health. These include events around topics such as lock acquisition and SQL statements starting and completing.
- *Debug events* are those expected to be used by DBAs and support engineers to help diagnose and solve engine-related problems. This channel includes events that fire when threads and processes start and stop, various times throughout a scheduler's lifecycle, and for other similar themes.
- *Operational events* are those expected to be of most use to operational DBAs for managing the SQL Server service and databases. This channel's events relate to databases being attached, detached, started, and stopped, as well as issues such as the detection of database page corruption.

Providing such a flexible event payload system ensures that any consumer can use any exposed event, so long as the consumer knows how to read the schema. Events are designed such that the output of each instance of the event always includes the same attributes, exposed in the exact order defined by the schema, to minimize the amount of work required for consumers to processes bound events. Event consumers can also use this ordering guarantee to more easily ignore data that they are not interested in. For example, if a consumer knows that the first 16 bytes of a given event contains an identifier that is not pertinent to the consumer's requirements, these bytes can simply be disregarded rather than needlessly processed.

Although the schema of each event is predetermined before run time, the actual size of each instance of the event is not. Event payloads can include both fixed and variable-length data elements, in addition to non-schematized elements populated by actions (see the section entitled "Actions" later in this chapter, for more information). To reduce the probability of events overusing memory and other resources, the system sets a hard 32-MB upper limit on the data size of variable-length elements.

One thing you might notice about the list of columns returned for each event is that it is small compared with the number of columns available for each event in SQL Trace. For example, the XE *sql_statement_completed* event exposes only seven columns: *source_database_id, object_id, object_type, cpu, duration, reads,* and *writes*. SQL Trace users might be wondering where all the other common attributes are—session ID, login name, perhaps the actual SQL text that caused the event to fire. These are all available by binding to "actions" (described in the section entitled "Actions," later in this chapter) and are not populated by default by the event's schema. This design further adds to the flexibility of the XE architecture and keeps events themselves as small as possible, thereby improving overall system performance.

As with SQL Trace events, XE events are disabled by default and have virtually no overhead until they are enabled in an event session (the XE equivalent of a trace, covered later in this chapter). Also like SQL Trace events, XE events can be filtered and can be routed to various post-event providers for collection. The terminology here is also a bit different; filters in XE are called *predicates,* and the post-event providers are referred to as *targets,* covered in the sections entitled "Predicates" and "Targets," respectively, later in this chapter.

Types and Maps

In the previous section, we saw that each event exposes its own schema, including column names and type information. Also mentioned was that each of the types included in these schemas is also defined within an XE package.

Two kinds of data types can be defined: scalar types and maps. A scalar type is a single value; something like an integer, a single Unicode character, or a binary large object. A map, on the other hand, is very similar to an enumeration in most object-oriented systems. The idea for a map is that many events have greater value if they can convey to the consumer some human-readable text about what occurred, rather than just a set of machine-readable values. Much of this text can be predefined—for example, the list of wait types supported by

SQL Server—and can be stored in a table indexed by an integer. At the time an event fires, rather than collecting the actual text, the event can simply store the integer, thereby saving large amounts of memory and processing resources.

Types and maps, like events, are visible in the *sys.dm_xe_objects* DMV. To see a list of both types and maps supported by the system, use the following query:

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type IN ('TYPE', 'MAP');
```

Although types are more or less self-describing, maps must expose their associated values so that consumers can display the human-readable text when appropriate. This information is available in a DMV called *sys.dm_xe_map_values*. The following query returns all the wait types exposed by the SQL Server engine, along with the map keys (the integer representation of the type) used within XE events that describe waits:

```
SELECT *
FROM sys.dm_xe_map_values
WHERE
    name = 'wait_types';
```

As of SQL Server 2008 RTM, virtually all the types are exposed via the *package0* package, whereas each of the four packages contain many of their own map values. This makes sense, given that a scalar type such as an integer does not need to be redefined again and again, whereas maps are more aligned to specific purposes.

It is also worth noting, from an architectural point of view, that some thought has been put into optimizing the type system by including pass-by-value and pass-by-reference semantics depending on the size of the object. Any object of 8 bytes or smaller is passed by value as the data flows through the system, whereas larger objects are passed by reference using a special XE-specific 8-byte pointer type.

Predicates

As with SQL Trace events, XE events can be filtered so that only interesting events are recorded. You may wish to record, for example, only events that occur in a specific database, or which fired for a specific session ID. In keeping with the design goal of providing the most flexible experience possible, XE predicates are assigned on a per-event basis, rather than to the entire session. This is quite a departure from SQL Trace, where filters are defined at the granularity of the entire trace, and so every event used within the trace must abide by the overarching filter set. In XE, if it makes sense to only filter some events and to leave other events totally unfiltered—or filtered using a different set of criteria—that is a perfectly acceptable option.

From a metadata standpoint, predicates are represented in *sys.dm_xe_objects* as two different object types: *pred_compare* and *pred_source.* The *pred_compare* objects are comparison functions, each designed to compare instances of a specific data type, whereas the *pred_source* objects are extended attributes that can be used within predicates.

First, we'll take a look at the *pred_compare* objects. The following query against the *sys.dm_xe_objects* DMV returns all >= comparison functions that are available, by filtering on the *pred_compare* object type:

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'pred_compare'
    AND name LIKE 'greater_than_equal%';
```

Running this query, you can see comparison functions defined for a number of base data types—integers, floating-point numbers, and various string types. Each of these functions can be used explicitly by an XE user, but the DDL for creating event sessions has been overloaded with common operators, so that this is unnecessary in the vast majority of cases. For example, if you use the >= operator to define a predicate based on two integers, the XE engine automatically maps the call to the *greater_than_equal_int64* predicate that you can see in the DMV. There is currently only one predicate that is not overloaded with an operator, a modulus operator that tests whether one input equally divides by the other. See the section entitled "Extended Events DDL and Querying," later in this chapter, for more information on how to use the comparison functions.

The other predicate object type—*pred_source*—requires a bit of background explanation. In the XE system, event predicates can filter on one of two types of attribute: a column exposed by the event itself—such as *source_database_id* for the *sql_statement_completed* event—or any of the external attributes (predicate sources) defined as *pred_source* in the *sys.dm_xe_objects* DMV. The available sources are returned by the following query:

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'pred_source';
```

Each of these attributes—29 as of SQL Server 2008 RTM—can be bound to any event in the XE system and can be used anytime you need to filter on an attribute that is not carried by the event's own schematized payload. This lets you ask for events that fired for a specific session ID, for a certain user name, or—if you want to debug at a deeper level—on a specific thread or worker address. The important thing to remember is that these predicate sources are not carried by any of the events by default, and using them forces the XE engine to acquire the data in an extra step during event processing. For most of the predicates, the acquisition cost is quite small, but if you are using several of them, this cost can add up.

We explore when and how predicates fire in the section entitled "Lifecycle of an Event," later in this chapter.

Actions

One quality of an eventing system is that as events fire, it may be prudent to exercise some external code. For example, consider DML triggers, which are events that fire in response to a DML action and exercise code in the form of the body of the trigger. Aside from doing some sort of work, external code can also retrieve additional information that might be important to the event; for example, a trigger can select data from another table in the system.

In XE, a type of object called an *action* takes on these dual purposes. Actions, if bound to an event, are synchronously invoked after the predicate evaluates to true and can both exercise code and write data back into the event's payload, thereby adding additional attributes. As mentioned in the section entitled "Events," earlier in this chapter, XE events are designed to be as lean as possible, including only a few attributes each by default. When dealing with predicates, the lack of a complete set of attributes can be solved using predicate sources, but these are only enabled for filtration. Using a predicate source does not cause its value to be stored along with the rest of the event data. The most common use of actions is to collect additional attributes not present by default on a given event.

It should by this point come as no surprise that to see a list of the available actions, a user should query *sys.dm_xe_objects*, as in the following example:

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'action';
```

As of SQL Server 2008 RTM, XE ships with 37 actions, which include attributes that map to virtually every predicate source, should you wish to filter on a given source as well as include the actual value in your event's output. The list also includes a variety of other attributes, as well as a handful of actions that exercise only code and do not return any data to the event's payload.

Actions fire synchronously on an event immediately after the predicates are evaluated, but before control is returned to the code that caused the event to fire (for more information, see the section entitled "Lifecycle of an Event," later in this chapter). This is done to ensure that actions will be able to collect information as it happens and before the server state changes, which might be a potential problem were they fired asynchronously.

As a result of their synchronous design, actions bear some performance cost. The majority of them—such as those that mirror the available predicates—are relatively inexpensive to retrieve, but others can be costly. For example, an especially interesting action useful for debugging purposes is the *tsql_stack* action, which returns the entire nested stack of stored procedure and/or function calls that resulted in the event firing. Although very useful, this information is not available in the engine without briefly stopping execution of the current thread and walking the stack, so this action bears a heavier performance cost than, for example, retrieving the current session ID.

To see a list of those actions that do not return any data but rather only execute external code, filter on the *type_name* column of *sys.dm_xe_objects* for a ”null” return value, as in the following query:

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'action'
    and type_name = 'null';
```

Note that “null” in this example is actually a string and is not the same as a SQL NULL; *null* is the name of a type defined in *package0* and shows up in the list of objects of type *type.* There are three actions that do not return additional data: two of them perform mini-dumps and the other causes a debugger breakpoint to fire. All these are best used only when instructed to by product support—especially the debug break event, which stops the active thread upon which the breakpoint is hit, potentially blocking the entire SQL Server process depending on where the breakpoint is hit.

Much like predicates, actions are bound on an event-by-event basis rather than at the event session level, so a consumer can choose to invoke actions only when specific events fire within a larger session. Certain actions may not apply to every event in the system, and these will fail to bind with an error at session creation time, if a user attempts to bind them with an incompatible event.

From a performance point of view, aside from the synchronous nature of these actions, it is important to remember that actions that write data back to the event increase the size of each instance of the event. This means that not only do events take longer to fire and return control to the caller—because actions are synchronously executed—but once fired, the event also consumes more memory and requires more processing time to write to the target. The key, as is often the case with performance-related issues, is to maintain a balance between the data needs of the consumer and the performance needs of the server as a whole. Keeping in mind that actions are not free helps you to create XE sessions that have less of an impact on the host server.

Targets

So far, we have seen events that fire when an instrumented code path is encountered, predicates that filter events so that only interesting data is collected, and actions that can add additional data to an event’s payload. Once all this has taken place, the final package of event data needs to go somewhere to be collected. This destination for event data is one or more targets, which are the means by which XE events are consumed.

Targets are the final object type that has metadata exposed within *sys.dm_xe_objects*, and the list of available targets can be seen by running the following query:

```
SELECT *
FROM sys.dm_xe_objects
WHERE
    object_type = 'target';
```

SQL Server 2008 RTM ships with 13 targets—7 public and 6 private, for use only by SQL Audit. Of the 7 public targets, 3 are marked *synchronous* in the *capabilities_desc* column. These targets collect event data synchronously—much like actions—before control is returned to the code that caused the event to fire. The other five events, in comparison, are asynchronous, meaning that the data is buffered before being collected by the target at some point after the event fires. Buffering results in better performance for the code that caused the event to fire, but it also introduces latency into the process because the target may not collect the event for some time.

XE targets come in a variety of types that are both similar to and somewhat different from the I/O providers exposed by SQL Trace. Similar to the SQL Trace file provider is the XE *asynchronous_file_target*, which buffers data before writing it out to a proprietary binary file format. Another file-based option is the *etw_classic_sync_target*, which synchronously writes data to a file format suitable for consumption by any number of ETW-enabled readers. There is no XE equivalent for the SQL Trace streaming rowset provider.

The remaining five targets are quite different than what is offered by SQL Trace, and all store consumed data in memory rather than persisting it to a file. The most straightforward of these is the *ring_buffer* target, which stores data in a ring buffer with a user-configurable size. A ring buffer loops back to the start of the buffer when it fills and begins overwriting data collected earlier. This means that the buffer can consume an endless quantity of data without using all available system memory, but only the newest data is available at any given time.

Another target type is the *synchronous_event_counter* target, which synchronously counts the number of times events have fired. Along these same lines are two bucketizer targets—one synchronous and the other asynchronous—which create buckets based on a user-defined column, and count the number of times that events occur within each bucket. For example, a user could “bucketize” based on session ID, and the targets would count the number of events that fired for each SPID.

The final target type is called the *pair_matching* target, and it is designed to help find instances where a pair of events is expected to occur, but one or the other is not firing due to a bug or some other problem. The *pair_matching* target works by asynchronously collecting events defined by the user as begin events, and matching them to events defined by the user as end events. When a pair of successfully matched events is found, both events are dropped, leaving only those events that did not have a match. For an example of where this would be useful, consider lock acquisition in the storage engine. Each lock is acquired and—we hope—released within a relatively short period to avoid blocking. If blocking problems are occurring, it is possible that they are due to locks being acquired and held for longer than necessary. By using the *pair_matching* target in conjunction with the lock acquired and lock released events, it is easy to identify those locks that have been taken but not yet released.

Targets can often be used in conjunction with one another, and it is therefore possible to bind multiple targets to a single session, rather than having to create many sessions to collect the same data. For example, a user can create multiple bucketizing targets to simultaneously keep metadata counts based on different bucket criteria, while recording all the unaggregated data to a file for later evaluation.

As with the SQL Trace providers, some action must occur when more data enters the system than can be processed in a reasonable amount of time. When working with the synchronous targets, things are simple; the calling code waits until the target returns control, and the target waits until its event data has been fully consumed. With asynchronous targets, on the other hand, there are a number of configuration options that dictate how to handle the situation.

When event data buffers begin to fill up, the engine can take one of three possible actions depending on how the session was configured by the user. These actions are the following:

- **Block, waiting for buffer space to become available (no event loss)** This is the same behavior characterized by the SQL Trace file provider, and can cause performance degradation.
- **Drop the waiting event (allow single event loss)** In this case, the system drops only a single event at a time while waiting for more buffer space to free up. This is the default mode.
- **Drop a full buffer (allow multiple event loss)** Each buffer can contain many events, and the number of events lost depends upon the size of the events in addition to the size of the buffers (which we will describe shortly).

The various options are listed here in decreasing order of their impact on overall system performance should buffers begin filling up, and in increasing order of the number of events that may be lost while waiting for buffers to become free. It is important to choose an option that reflects the amount of acceptable data loss while keeping in mind that blocking will occur should too restrictive an option be used. Liberal use of predicates, careful attention to the number of actions bound to each event, and attention to other configuration options all help users avoid having to worry about buffers filling up and whether the choice of these options is a major issue.

Along with the ability to specify what should happen when buffers fill up, a user can specify how much memory is allocated, how the memory is allocated across CPU or NUMA node boundaries, and how often buffers are cleared.

By default, one central set of buffers, consuming a maximum of 4 MB of memory, is created for each XE session (as described in the next section). The central set of buffers always contains three buffers, each consuming one-third of the maximum amount of memory specified. A user can override these defaults, creating one set of buffers per CPU or one set per NUMA node, and increasing or decreasing the amount of memory that each set of buffers consumes. In addition, a user can specify that events larger than the maximum allocated buffer memory should be allowed to fire. In that case, those events are stored in special large memory buffers.

Another default option is that buffers are cleared every 30 seconds or when they fill up. This option can be overridden by a user and a maximum latency set. This causes the buffers to be checked and cleared both at a specific time interval (specified as a number of seconds), in addition to when they fill up.

It is important to note that each of these settings applies not on a per-target basis, but rather to any number of targets that are bound to a session. We explore how this works in the next section.

Event Sessions

We have now gone through each of the elements that make up the core XE infrastructure. Bringing each of these together into a cohesive unit at run time are sessions. These are the XE equivalent of a trace in SQL Trace parlance. A session describes the events that the user is interested in collecting, predicates against which the events should be filtered, actions that should fire in conjunction with the events, and finally targets that should be used for data collection at the end of the cycle.

Any number of sessions can be created by users with adequate server-level permission, and sessions are for the most part independent of one another, just as with SQL Trace. The main thread that links any number of sessions is a central bitmap that indicates whether a given event is enabled or disabled. An event can be enabled simultaneously in any number of sessions, but the global bitmap is used to avoid having to check each of those sessions at run time. Beyond this level, sessions are completely separate from one another, and each uses its own memory and has its own set of defined objects.

Session-Scoped Catalog Metadata

Along with defining a set of events, predicates, actions, and targets, various XE configuration options are scoped at the session level. As with the objects that define the basis for XE, a number of views have been added to the metadata repository of SQL Server to support metadata queries about sessions.

The *sys.server_event_sessions* catalog view is the central metadata store for information about XE sessions. The view exposes one row per session defined on the SQL Server instance. Like traces in SQL Trace, XE sessions can be started and stopped at will. But unlike traces, XE sessions are persistent with regard to service restarts, and so querying the view before and after a restart show the same results unless a session has been explicitly dropped. A session can be configured to start itself automatically when the SQL Server instance starts; this setting can be seen via the *startup_state* column of the view.

Along with the central *sys.server_event_sessions* views are a number of other views describing details of how the session was configured. The *sys.server_event_session_events* view exposes one row per event bound to each session, and includes a predicate column that contains the definition of the predicate used to filter the event, if one has been set. There are similar views for actions and targets, namely: *sys.server_event_session_actions* and *sys.server_event_ session_ targets.* A final view, *sys.server_event_session_fields,* contains information about settings that can be customized for a given event or target. For example, the ring buffer target's memory consumption can be set to a specific amount by a user; if the target is used, the memory setting appears in this view.

Session-Scoped Configuration Options

As mentioned in the section entitled "Targets," earlier in this chapter, a number of settings are set globally for a session and, in turn, influence the run-time behavior of the objects that make up the session.

The first set of session-scoped options includes those that we have already discussed: options that determine how asynchronous target buffers are configured, both from a memory and latency standpoint. These settings influence a process called the *dispatcher,* which is responsible for periodically collecting data from the buffers and sending it to each of the asynchronous targets bound to the session. The frequency with which the dispatcher is activated depends on how the memory and latency settings are configured. If a latency value of *infinite* is specified, the dispatcher does not collect data except when the buffers are full. Otherwise, the dispatcher collects data at the interval determined by the setting—as often as once a second.

The *sys.dm_xe_sessions* DMV can be used to monitor whether there are any problems dispatching asynchronous buffers. This DMV exposes one row per XE session that has been started and exposes a number of columns that can give a user insight into how buffers are being handled. The most important columns are the following:

- *regular_buffer_size* and *total_regular_buffers.* These columns expose the number of buffers created—based on the maximum memory and memory partitioning settings—as well as the size of each buffer. Knowing these numbers and estimating the approximate size for each event tells you how many events you might lose in case of a full buffer situation, should you make use of the allow multiple event loss option.
- *dropped_event_count* and *dropped_buffer_count.* These columns expose the number of events and/or buffers that have been dropped due to there not being enough free buffer space to accommodate incoming event data.
- *blocked_event_fire_time.* This column exposes the amount of time that blocking occurred, if the no event loss option

was used.

Another session-scoped option that can be enabled is called *causality tracking.* This option enables users to use a SQL Server engine feature to help correlate events either when there are parent-child relationships between tasks on the same thread or when one thread causes activity to occur on another thread. In the engine code, these relationships are tracked by each task defining a GUID, known as an *activity ID.* When a child task is called, the ID is passed along and continues down the stack as subsequent tasks are called. If activity needs to pass to another thread, the ID is passed in a structure called a *transfer block,* and the same logic continues.

These identifiers are exposed via two XE actions: *package0.attach_activity_id* and *package0. attach_activity_id_xfer*. However, these actions cannot be attached to an event by a user creating a session. Instead, a user must enable the causality tracking option at the session level, which automatically binds the actions to every event defined for the session. Once the actions are enabled, both the activity ID and activity transfer ID are added to each event's payload.

Lifecycle of an Event

The firing of an event means, at its core, that a potentially "interesting" point in the SQL Server code has been encountered. This point in the code calls a central function that handles the event logic, and several things happen, as described in this section and as illustrated in Figure 2-13.

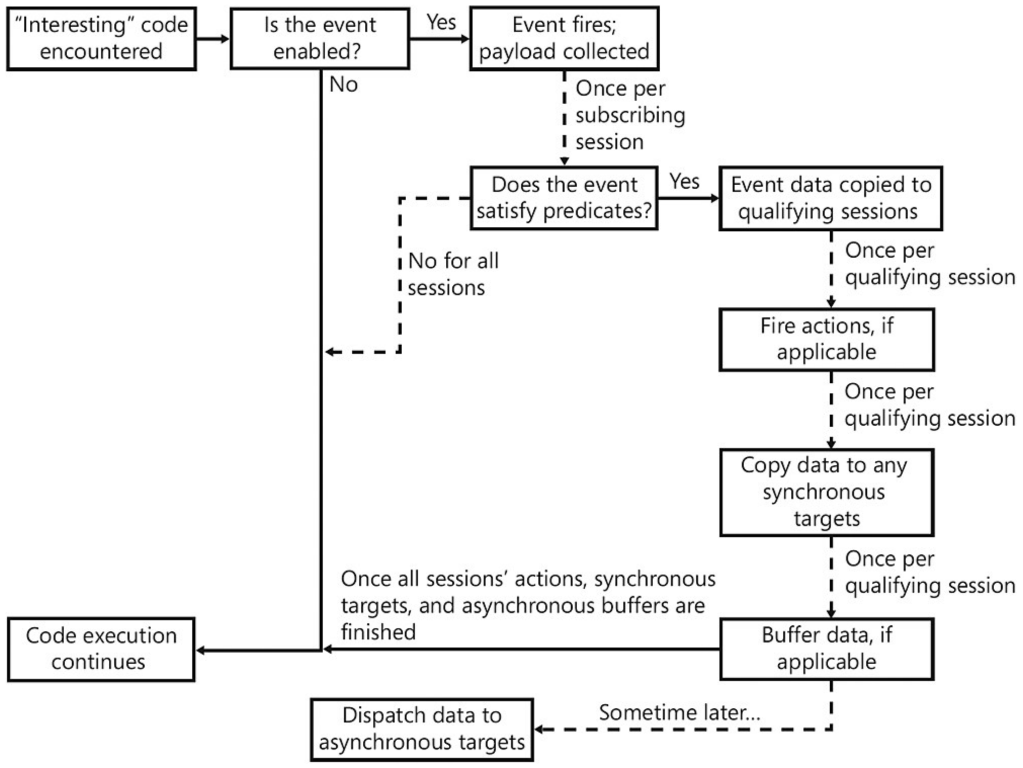

"Interesting" code encountered
Is the event enabled?
Yes
Event fires; payload collected
No
Once per subscribing session
Does the event satisfy predicates?
Yes
Event data copied to qualifying sessions
No for all sessions
Once per qualifying session
Fire actions, if applicable
Once per qualifying session
Copy data to any synchronous targets
Once per qualifying session
Once all sessions' actions, synchronous targets, and asynchronous buffers are finished
Code execution continues
Buffer data, if applicable
Dispatch data to asynchronous targets
Sometime later...

Figure 2-13: The lifecycle of an extended event

Once an event has been defined within at least one session, a global bitmap is set to indicate that the event should fire when code that references it is encountered. Whether or not an event is enabled, the code must always perform this check; for events that are not enabled, the check involves a single code branch and adds virtually no overhead to the SQL Server process. If the event is not enabled, this is the end of the process and the code continues its normal execution path. Only if an event is enabled in one or more sessions must the event-specific code continue processing.

At this point, if enabled, the event fires and all the data elements associated with its schema are collected and packaged. The XE engine next finds each session that has the event enabled and synchronously (one session at a time) takes the following steps:

1. Check whether the event satisfies predicates defined for the event within the session. If not, the engine moves on to the next session without taking any further action.

2. If the predicates are satisfied, the engine copies the event data into the session's context. Any actions defined for the event within the session are then fired, followed by copying the event data to any synchronous targets.

3. Finally, the event data is buffered, if necessary, for any asynchronous targets used by the session.

Once each of these steps has been performed for each session, code execution resumes. It is important to stress that this all happens synchronously, while code execution blocks. Although each of these steps, and the entire system, has been designed for performance, users can still create problems by defining too many sessions, with too many actions or synchronous targets, for extremely active events such as those in the analytic channel. Care should be taken to avoid overusing the synchronous features, lest run-time blocking becomes an issue.

At some point after being buffered—depending on the event latency and memory settings for the session(s)—the event data is passed once more, to any asynchronous targets. At this point, the event data is removed from the buffer to make room for new incoming data.

To help track down problems with targets taking too long to consume the data and therefore causing waiting issues, the *sys.dm_xe_session_targets* DMV can be used. This DMV exposes one row per target defined by each active XE session, and includes a column called *execution_duration_ms*. This column indicates the amount of time that the target took to process the most recent event or buffer (depending on the target). If you see this number begin to climb, waiting issues are almost certainly occurring in SQL Server code paths.

Extended Events DDL and Querying

To complete the overview of XE, we will take a quick tour of the session creation DDL and see how all the objects apply to what you can control when creating actual sessions. We will also look at an example of how to query some of the data collected by an XE session.

Creating an Event Session

The primary DDL hook for XE is the *CREATE EVENT SESSION* statement. This statement allows users to create sessions and map all the various XE objects. An *ALTER EVENT SESSION* statement also exists, allowing a user to modify a session that has already been created. To modify an existing session, it must not be active.

The following T-SQL statement creates a session and shows how to configure all the XE features and options we have reviewed in the chapter:

```
CREATE EVENT SESSION [statement_completed]
ON SERVER
ADD EVENT
    sqlserver.sp_statement_completed,
ADD EVENT
    sqlserver.sql_statement_completed
    (
        ACTION
        (
            sqlserver.sql_text
        )
        WHERE
        (
            sqlserver.session_id = 53
        )
    )
ADD TARGET
    package0.ring_buffer
    (
        SET
            max_memory=4096
    )
WITH
(
    MAX_MEMORY = 4096KB,
    EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,
    MAX_DISPATCH_LATENCY = 1 SECONDS,
    MEMORY_PARTITION_MODE = NONE,
    TRACK_CAUSALITY = OFF,
    STARTUP_STATE = OFF
);
```

The session is called *statement_completed,* and two events are bound: *sp_statement_completed* and *sql_statement_completed,* both exposed by the *sqlserver* package. The *sp_statement_ completed* event has no actions or predicates defined, so it publishes to the session's target with its default set of attributes every time the event fires instance-wide. The *sql_statement_completed* event, on the other hand, has a predicate configured (the WHERE option) so that it publishes only for session ID 53. Note that the predicate uses the equality operator (=) rather than calling the *pred_compare* function for comparing two integers. The standard comparison operators are all defined; currently the only reason to call a function directly is for using the *divides_by_uint64* function, which determines whether one integer exactly divides by another (useful when working with the counter predicate source). Note also that the WHERE clause supports AND, OR, and parentheses—you can create complex predicates that combine many different conditions if needed.

When the *sql_statement_completed* event fires for session ID 53, the event session invokes the *sql_text* action. This action collects the text of the SQL statement that caused the event to fire and adds it to the event's data. After the event data has been collected, it is pushed to the *ring_buffer* target, which is configured to use a maximum of 4,096 KB of memory.

We have also configured some session-level options. The session's asynchronous buffers cannot consume more than 4,096 KB of memory, and should they fill up, we allow events to be dropped. That is probably not likely to happen, though, because we have configured the dispatcher to clear the buffers every second. Memory is not partitioned across CPUs—so we end up with three buffers—and we are not using causality tracking. Finally, after the session is created, it exists only as metadata; it does not start until we issue the following statement:

```
ALTER EVENT SESSION [statement_completed]
ON SERVER
STATE=START;
```

Querying Event Data

Once the session is started, the ring buffer target is updated with new events (assuming there are any) every second. Each of the in-memory targets—the ring buffer, bucketizers, and event count targets—exposes its data in XML format in the *target_data* column of the *sys.dm_xe_session_targets* DMV. Given the fact that the data is in XML format, many DBAs who have not yet delved into XQuery may want to try it; we highly recommend learning how to query the data, given the richness of the information that can be retrieved using XE.

Consuming the XML in a tabular format requires knowledge of which nodes are present. In the case of the ring buffer target, a root node called *RingBufferTarget* includes one event node for each event that fires. The event node contains one data node for each attribute contained within the event data, and one "action" node for actions bound to the event. These data and action nodes contain three nodes each: one node called *type,* which indicates the data type; one called *value*, which includes the value in most cases; and one called *text* which is there for longer text values.

Explaining how to query every possible event and target is beyond the scope of this book, but a quick sample query based on the *statement_completed* session follows; you can use this query as a base from which to work up queries against other events and actions when working with the ring buffer target:

```
SELECT
    theNodes.event_data.value('(data/value)[1]', 'bigint') AS source_database_id,
    theNodes.event_data.value('(data/value)[2]', 'bigint') AS object_id,
    theNodes.event_data.value('(data/value)[3]', 'bigint') AS object_type,
    theNodes.event_data.value('(data/value)[4]', 'bigint') AS cpu,
    theNodes.event_data.value('(data/value)[5]', 'bigint') AS duration,
    theNodes.event_data.value('(data/value)[6]', 'bigint') AS reads,
    theNodes.event_data.value('(data/value)[7]', 'bigint') AS writes,
    theNodes.event_data.value('(action/value)[1]', 'nvarchar(max)') AS sql_text
FROM
(
    SELECT
        CONVERT(XML, st.target_data) AS ring_buffer
    FROM sys.dm_xe_sessions s
    JOIN sys.dm_xe_session_targets st ON
        s.address = st.event_session_address
    WHERE
        s.name = 'statement_completed'
) AS theData
CROSS APPLY theData.ring_buffer.nodes('//RingBufferTarget/event') theNodes (event_data);
```

This query converts the ring buffer data to an XML instance and then uses the *nodes* XML function to create one row per event node found. It then uses the ordinal positions of the various data elements within the event nodes to map the data to output columns. Of course, more advanced sessions require more advanced XQuery to determine the type of each event and do some case logic if the events involved in the session have different schemas—which, thankfully, the two in this example do not. Once you've gotten to this point, the data is just that—standard tabular data, which can be aggregated, joined, inserted into a table, or whatever else you want to do with it.

You can also read from the asynchronous file target via T-SQL, using the *sys.fn_xe_file_target_ read_file* table-valued function. This function returns one row per event, but you still have to get comfortable with XML; the event's data, exposed in a column called *event_data,* is in an XML format similar to data in the ring buffer target. Eventually we can expect a user interface to bear some of the XML burden for us, but just as with SQL Trace, even the most powerful user interfaces aren't enough when complex analysis is required. Therefore, XML is here to stay for those DBAs who wish to be XE power users.

Stopping and Removing the Event Session

Once you have finished reading data from the event session, it can be stopped using the following code:

```
ALTER EVENT SESSION [statement_completed]
ON SERVER
STATE=STOP;
```

Stopping the event session does not remove the metadata; to eliminate the session from the server completely, you must drop it using the following statement:

```
ALTER EVENT SESSION [statement_completed]
ON SERVER;
```

Summary

SQL Server has many eventing systems that range from the simple—like triggers and event notifications—to the intricate—like XE. Each of these systems is designed to help both users and SQL Server itself work better by enabling arbitrary code execution or data collection when specific actions occur in the database engine. In this chapter, we explored the various hidden and internal objects used by Change Tracking to help support synchronization applications, the inner workings of the ubiquitous SQL Trace infrastructure, and the complex architecture of XE, the future of eventing within SQL Server. Events within SQL Server are extremely powerful, and we hope that this chapter has provided you with enough internal knowledge of these systems to understand how to better use the many eventing features in your day-to-day activities.