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

Extended Events Recommendations

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

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

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

Set Max File Size Appropriately

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

Be Cautious with Debug Events

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

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

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

Avoid Use of No_Event_Loss

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

Other Methods for Query Performance Metrics

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

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

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

Table 6-5. *sys.dm_exec_query_stats Output*

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

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

Summary

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

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

CHAPTER 7

Analyzing Query Performance

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

In this chapter, I cover the following topics:

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

Costly Queries

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

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

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

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

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

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

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

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

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

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

Identifying Costly Queries

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

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

(8KB in size) a query operated on and thereby indicate the amount of memory or I/O stress caused by the query. They also indicate disk stress since memory pages have to be backed up in the case of action queries, populated during first-time data access, and displaced to disk during memory bottlenecks. The higher the number of logical reads for a query, the higher the possible stress on the disk could be. An excessive number of logical pages also increases load on the CPU in managing those pages. This is not an automatic correlation. You can't always count on the query with the highest number of reads being the poorest performer. But it is a general metric and a good starting point. Although minimizing the number of I/Os is not a requirement for a cost-effective plan, you will often find that the least costly plan generally has the fewest I/Os because I/O operations are expensive.

The queries that cause a large number of logical reads usually acquire locks on a correspondingly large set of data. Even reading (as opposed to writing) may require shared locks on all the data, depending on the isolation level. These queries block all other queries requesting this data (or part of the data) for the purposes of modifying it, not for reading it. Since these queries are inherently costly and require a long time to execute, they block other queries for an extended period of time. The blocked queries then cause blocks on further queries, introducing a chain of blocking in the database. (Chapter 13 covers lock modes.)

As a result, it makes sense to identify the costly queries and optimize them first, thereby doing the following:

- Improving the performance of the costly queries themselves
- Reducing the overall stress on system resources
- Reducing database blocking

The costly queries can be categorized into the following two types:

- *Single execution*: An individual execution of the query is costly.
- *Multiple executions*: A query itself may not be costly, but the repeated execution of the query causes pressure on the system resources.

You can identify these two types of costly queries using different approaches, as explained in the following sections.

Costly Queries with a Single Execution

You can identify the costly queries by analyzing a session output file, by using the Query Store, or by querying `sys.dm_exec_query_stats`. For this example, we'll start with identifying queries that perform a large number of logical reads, so you should sort the session output on the `logical_reads` data column. You can change that around to sort on duration or CPU or even combine them in interesting ways. You can access the session information by following these steps:

1. Capture a session that contains a typical workload.
2. Save the session output to a file.
3. Open the file by using File ➤ Open and select a `.xel` file to use the data browser window. Sort the information there.
4. Alternatively, you can query the trace file for analysis sorting by the `logical_reads` field.

```
WITH    xEvents
          AS (SELECT    object_name AS xEventName,
                        CAST (event_data AS XML) AS xEventData
                FROM    sys.fn_xe_file_target_read_file('C:\Sessions\
                        QueryPerformanceMetrics*.xel',
                                                        NULL, NULL, NULL)
             )
    SELECT  xEventName,
            xEventData.value('(/event/data[@name="duration"]/value)[1]',
                              'bigint') Duration,
            xEventData.value('(/event/data[@name="physical_reads"]
            /value)[1]', 'bigint') PhysicalReads,
            xEventData.value('(/event/data[@name="logical_reads"]
            /value)[1]',
                              'bigint') LogicalReads,
            xEventData.value('(/event/data[@name="cpu_time"]/value)[1]',
                              'bigint') CpuTime,
            CASE xEventName
              WHEN 'sql_batch_completed'
              THEN xEventData.value('(/event/data[@name="batch_text"]/
              value)[1]',

                                    'varchar(max)')
              WHEN 'rpc_completed'
              THEN xEventData.value('(/event/data[@name="statement"]/value)[1]',

                                    'varchar(max)')
            END AS SQLText,
            xEventData.value('(/event/data[@name="query_hash"]/value)[1]',
                             'binary(8)') QueryHash
    INTO    Session_Table
    FROM    xEvents;

SELECT  st.xEventName,
        st.Duration,
        st.PhysicalReads,
        st.LogicalReads,
        st.CpuTime,
        st.SQLText,
        st.QueryHash
FROM    Session_Table AS st
ORDER BY st.LogicalReads DESC;
```

Let's break down this query a little. First, I'm creating a common table expression (CTE) called `xEvents`. I'm doing that just because it makes the code a little easier to read. It doesn't fundamentally change any behavior. I prefer it when I have to both read from a file and convert the data type. Then my XML queries in the following statement make a little more sense. Note that I'm using a wildcard when reading from the file, `QueryPerformanceMetrics*.xel`. This makes it possible for me to read in all rollover files created by the Extended Events session (for more details, see Chapter 6).

Depending on the amount of data collected and the size of your files, running queries directly against the files you've collected from Extended Events may be excessively slow. In that case, use the same basic function, `sys.fn_xe_file_target_read_file`, to load the data into a table instead of querying it directly. Once that's done, you can apply indexing to the table to speed up the queries. I used the previous script to put the data into a table and then queried that table for my output. This will work fine for testing, but for a more permanent solution you'd want to have a database dedicated to storing this type of data with tables having the appropriate structures rather than using a shortcut like `INTO` as I did here.

In some cases, you may have identified a large stress on the CPU from the System Monitor output. The pressure on the CPU may be because of a large number of CPU-intensive operations, such as stored procedure recompilations, aggregate functions, data sorting, hash joins, and so on. In such cases, you should sort the session output on the `cpu_time` field to identify the queries taking up a large number of processor cycles.

Costly Queries with Multiple Executions

As I mentioned earlier, sometimes a query may not be costly by itself, but the cumulative effect of multiple executions of the same query might put pressure on the system resources. In this situation, sorting on the `logical_reads` field won't help you identify this type of costly query. You instead want to know the total number of reads, the total CPU time, or just the accumulated duration performed by multiple executions of the query.

- Query the session output and group on some of the values you're interested in.
- Query the information within the Query Store.
- Access the `sys.dm_exec_query_stats` DMO to retrieve the information from the production server. This assumes you're dealing with an issue that is either recent or not dependent on a known history because this data is only what is currently in the procedure cache.

If you're looking for an accurate historical view of the data, you can go to the metrics you've collected with Extended Events or to the information with the Query Store, depending on how often you purge that data (more on this in Chapter 11). The Query Store has aggregated data that you can use for this type of investigation. However, it has only aggregated information. If you also want detailed, individual call, you will be back to using Extended Events.

Once the session data is imported into a database table, execute a SELECT statement to find the total number of reads performed by the multiple executions of the same query, as follows:

```
SELECT COUNT(*) AS TotalExecutions,
    st.xEventName,
    st.SQLText,
    SUM(st.Duration) AS DurationTotal,
    SUM(st.CpuTime) AS CpuTotal,
    SUM(st.LogicalReads) AS LogicalReadTotal,
    SUM(st.PhysicalReads) AS PhysicalReadTotal
FROM Session_Table AS st
GROUP BY st.xEventName, st.SQLText
ORDER BY LogicalReadTotal DESC;
```

The TotalExecutions column in the preceding script indicates the number of times a query was executed. The LogicalReadTotal column indicates the total number of logical reads performed by the multiple executions of the query.

The costly queries identified by this approach are a better indication of load than the costly queries with single execution identified by a session. For example, a query that requires 50 reads might be executed 1,000 times. The query itself may be considered cheap enough, but the total number of reads performed by the query turns out to be 50,000 (= 50 × 1,000), which cannot be considered cheap. Optimizing this query to reduce the reads by even 10 for individual execution reduces the total number of reads by 10,000 (= 10 × 1,000), which can be more beneficial than optimizing a single query with 5,000 reads.

The problem with this approach is that most queries will have a varying set of criteria in the WHERE clause or that procedure calls will have different values passed in. That makes the simple grouping by the query or procedure with parameters just impossible. You can take care of this problem with a number of approaches. Because you have Extended Events, you can actually put it to work for you. For example, the rpc_completed event captures the procedure name as a field. You can simply group on that field. For batches, you can add the query_hash field and then group on that. Another way is to clean the data, removing the parameter values, as outlined on the Microsoft Developers Network at http://bit.ly/1e1I38f. Although it was written originally for SQL Server 2005, the concepts will work fine with other versions of SQL Server up to SQL Server 2017.

Getting the same information out of the sys.dm_exec_query_stats view simply requires a query against the DMV.

```
SELECT s.TotalExecutionCount,
       t.text,
       s.TotalExecutionCount,
       s.TotalElapsedTime,
       s.TotalLogicalReads,
       s.TotalPhysicalReads
FROM
(
    SELECT deqs.plan_handle,
           SUM(deqs.execution_count) AS TotalExecutionCount,
           SUM(deqs.total_elapsed_time) AS TotalElapsedTime,
           SUM(deqs.total_logical_reads) AS TotalLogicalReads,
           SUM(deqs.total_physical_reads) AS TotalPhysicalReads
    FROM sys.dm_exec_query_stats AS deqs
    GROUP BY deqs.plan_handle
) AS s
    CROSS APPLY sys.dm_exec_sql_text(s.plan_handle) AS t
ORDER BY s.TotalLogicalReads DESC;
```

Another way to take advantage of the data available from the execution DMOs is to use query_hash and query_plan_hash as aggregation mechanisms. While a given stored procedure or parameterized query might have different values passed to it, changing query_hash and query_plan_hash for these will be identical (most of the time). This means you can aggregate against the hash values to identify common plans or common query patterns that you wouldn't be able to see otherwise. The following is just a slight modification from the previous query:

```
SELECT s.TotalExecutionCount,
       t.text,
       s.TotalExecutionCount,
       s.TotalElapsedTime,
       s.TotalLogicalReads,
       s.TotalPhysicalReads
```

```
FROM
(
    SELECT deqs.query_plan_hash,
           SUM(deqs.execution_count) AS TotalExecutionCount,
           SUM(deqs.total_elapsed_time) AS TotalElapsedTime,
           SUM(deqs.total_logical_reads) AS TotalLogicalReads,
           SUM(deqs.total_physical_reads) AS TotalPhysicalReads
    FROM sys.dm_exec_query_stats AS deqs
    GROUP BY deqs.query_plan_hash
) AS s
    CROSS APPLY
(
    SELECT plan_handle
    FROM sys.dm_exec_query_stats AS deqs
    WHERE s.query_plan_hash = deqs.query_plan_hash
) AS p
    CROSS APPLY sys.dm_exec_sql_text(p.plan_handle) AS t
ORDER BY TotalLogicalReads DESC;
```

This is so much easier than all the work required to gather session data that it makes you wonder why you would ever use Extended Events at all. The main reason is, as I wrote at the start of this chapter, precision. The `sys.dm_exec_ query_stats` view is a running aggregate for the time that a given plan has been in memory. An Extended Events session, on the other hand, is a historical track for whatever time frame you ran it in. You can even add session results from Extended Events to a database. With a list of data, you can generate totals about the events in a more precise manner rather than simply relying on a given moment in time. However, please understand that a lot of troubleshooting of performance problems is focused on what has happened recently on the server, and since `sys.dm_exec_query_stats` is based in the cache, the DMV usually represents a recent picture of the system, so `sys.dm_exec_query_stats` is extremely important. But, if you're dealing with that much more tactical situation of what the heck is running slow right now, you would use `sys.dm_exec_requests`.

You'll find that the Query Store is the same as the DMOs for ease of use. However, since the information within it is not cache dependent, it can be more useful than the DMO data. Just like the DMOs, though, the Query Store doesn't have the detailed record of an Extended Events session.

Identifying Slow-Running Queries

Because a user's experience is highly influenced by the response time of their requests, you should regularly monitor the execution time of incoming SQL queries and find out the response time of slow-running queries, creating a query performance baseline. If the response time (or duration) of slow-running queries becomes unacceptable, then you should analyze the cause of performance degradation. Not every slow-performing query is caused by resource issues, though. Other concerns such as blocking can also lead to slow query performance. Blocking is covered in detail in Chapter 12.

To identify slow-running queries, just change the queries against your session data to change what you're ordering by, like this:

```
WITH xEvents
AS (SELECT object_name AS xEventName,
           CAST(event_data AS XML) AS xEventData
    FROM sys.fn_xe_file_target_read_file('Q:\Sessions\
    QueryPerformanceMetrics*.xel', NULL, NULL, NULL)
   )
SELECT xEventName,
       xEventData.value('(/event/data[@name="duration"]/value)[1]',
       'bigint') Duration,
       xEventData.value('(/event/data[@name="physical_reads"]/value)[1]',
       'bigint') PhysicalReads,
       xEventData.value('(/event/data[@name="logical_reads"]/value)[1]',
       'bigint') LogicalReads,
       xEventData.value('(/event/data[@name="cpu_time"]/value)[1]',
       'bigint') CpuTime,
       xEventData.value('(/event/data[@name="batch_text"]/value)[1]',
       'varchar(max)') BatchText,
       xEventData.value('(/event/data[@name="statement"]/value)[1]',
       'varchar(max)') StatementText,
       xEventData.value('(/event/data[@name="query_plan_hash"]/value)[1]',
       'binary(8)') QueryPlanHash
FROM xEvents
ORDER BY Duration DESC;
```

For a slow-running system, you should note the duration of slow-running queries before and after the optimization process. After you apply optimization techniques, you should then work out the overall effect on the system. It is possible that your optimization steps may have adversely affected other queries, making them slower.

Execution Plans

Once you have identified a costly query, you need to find out *why* it is so costly. You can identify the costly procedure from Extended Events, the Query Store, or `sys.dm_exec_procedure_stats`; rerun it in Management Studio; and look at the execution plan used by the query optimizer. An execution plan shows the processing strategy (including multiple intermediate steps) used by the query optimizer to execute a query.

To create an execution plan, the query optimizer evaluates various permutations of indexes, statistics, constraints, and join strategies. Because of the possibility of a large number of potential plans, this optimization process may take a long time to generate the most cost-effective execution plan. To prevent the overoptimization of an execution plan, the optimization process is broken into multiple phases. Each phase is a set of transformation rules that evaluate various database objects and settings directly related to the optimization process, ultimately attempting to find a good enough plan, not a perfect plan. It's that difference between good enough and perfect that can lead to poor performance because of inadequately optimized execution plans. The query optimizer will attempt only a limited number of optimizations before it simply goes with the least costly plan it has currently (this is known as a *timeout*).

After going through a phase, the query optimizer examines the estimated cost of the resulting plan. If the query optimizer determines that the plan is cheap enough, it will use the plan without going through the remaining optimization phases. However, if the plan is not cheap enough, the optimizer will go through the next optimization phase. I will cover execution plan generation in more depth in Chapter 15.

SQL Server displays a query execution plan in various forms and from two different types. The most commonly used forms in SQL Server 2017 are the graphical execution plan and the XML execution plan. Actually, the graphical execution plan is simply an XML execution plan parsed for the screen. The two types of execution plan are the estimated plan and the actual plan. The *estimated* plan represents the results coming from the query optimizer, and the *actual* plan is that same plan plus some runtime metrics. The beauty of the estimated plan is that it doesn't require the query to be executed. The plans generated by these types can differ, but only if a statement-level recompile occurs during execution. Most of the time the two types of plans will be the same. The primary difference is the inclusion of some execution statistics in the actual plan that are not present in the estimated plan.

The graphical execution plan uses icons to represent the processing strategy of a query. To obtain a graphical estimated execution plan, select Query ➤ Display Estimated Execution Plan. An XML execution plan contains the same data available through the graphical plan but in a more programmatically accessible format. Further, with the XQuery capabilities of SQL Server, XML execution plans can be queried as if they were tables. An XML execution plan is produced by the statement `SET SHOWPLAN_XML` for an estimated plan and by the statement `SET STATISTICS XML` for the actual execution plan. You can also right-click a graphical execution plan and select Showplan XML. You can also pull plans directly out of the plan cache using a DMO, `sys.dm_exec_query_plan`. The plans stored in cache have no runtime information, so they are technically estimated plans. The same goes for the plans stored in the Query Store.

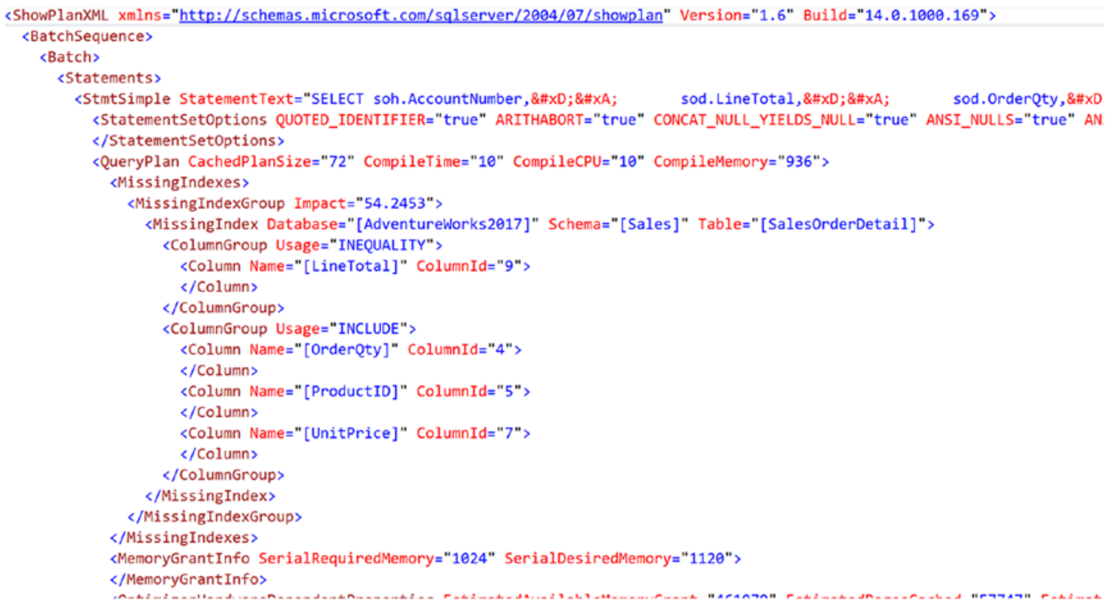
Note You should make sure your database is set to Compatibility Mode 140 so that it accurately reflects updates to SQL Server 2017.

You can obtain the estimated XML execution plan for the costliest query identified previously using the `SET SHOWPLAN_XML` command as follows:

```
USE AdventureWorks2017;
GO
SET SHOWPLAN_XML ON;
GO
SELECT soh.AccountNumber,
       sod.LineTotal,
       sod.OrderQty,
       sod.UnitPrice,
       p.Name
FROM Sales.SalesOrderHeader soh
    JOIN Sales.SalesOrderDetail sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product p
        ON sod.ProductID = p.ProductID
WHERE sod.LineTotal > 20000;

GO
SET SHOWPLAN_XML OFF;
GO
```

Running this query results in a link to an execution plan, not an execution plan or any data. Clicking the link will open an execution plan. Although the plan will be displayed as a graphical plan, right-clicking the plan and selecting Show Execution Plan XML will display the XML data. Figure 7-1 shows a portion of the XML execution plan output.

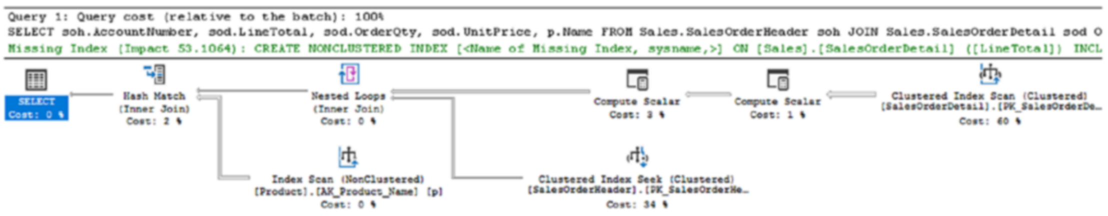


```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.6" Build="14.0.1000.169">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="SELECT soh.AccountNumber,&#xD;&#xA;       sod.LineTotal,&#xD;&#xA;       sod.OrderQty,&#xD
          <StatementSetOptions QUOTED_IDENTIFIER="true" ARITHABORT="true" CONCAT_NULL_YIELDS_NULL="true" ANSI_NULLS="true" AN
          </StatementSetOptions>
          <QueryPlan CachedPlanSize="72" CompileTime="10" CompileCPU="10" CompileMemory="936">
            <MissingIndexes>
              <MissingIndexGroup Impact="54.2453">
                <MissingIndex Database="[AdventureWorks2017]" Schema="[Sales]" Table="[SalesOrderDetail]">
                  <ColumnGroup Usage="INEQUALITY">
                    <Column Name="[LineTotal]" ColumnId="9">
                    </Column>
                  </ColumnGroup>
                  <ColumnGroup Usage="INCLUDE">
                    <Column Name="[OrderQty]" ColumnId="4">
                    </Column>
                    <Column Name="[ProductID]" ColumnId="5">
                    </Column>
                    <Column Name="[UnitPrice]" ColumnId="7">
                    </Column>
                  </ColumnGroup>
                </MissingIndex>
              </MissingIndexGroup>
            </MissingIndexes>
            <MemoryGrantInfo SerialRequiredMemory="1024" SerialDesiredMemory="1120">
            </MemoryGrantInfo>
```

Figure 7-1. *XML execution plan output*

Analyzing a Query Execution Plan

Let's start with the costly query identified in the previous section. Copy it (minus the `SET SHOWPLAN_XML` statements) into Management Studio into a query window. We can immediately capture an execution plan by selecting the Display Estimated Execution Plan button or hitting Ctrl+L. You'll see the execution plan in Figure 7-2.



```
Query 1: Query cost (relative to the batch): 100%
SELECT soh.AccountNumber, sod.LineTotal, sod.OrderQty, sod.UnitPrice, p.Name FROM Sales.SalesOrderHeader soh JOIN Sales.SalesOrderDetail sod O
Missing Index (Impact 53.1064): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[SalesOrderDetail] ([LineTotal]) INCL
```

SELECT
Cost: 0 %
Hash Match
(Inner Join)
Cost: 2 %
Nested Loops
(Inner Join)
Cost: 0 %
Compute Scalar
Cost: 3 %
Compute Scalar
Cost: 1 %
Clustered Index Scan (Clustered)
[SalesOrderDetail].[PK_SalesOrderDe…
Cost: 60 %
Index Scan (NonClustered)
[Product].[AK_Product_Name] [p]
Cost: 0 %
Clustered Index Seek (Clustered)
[SalesOrderHeader].[PK_SalesOrderHe…
Cost: 34 %

Figure 7-2. *Query execution plan*

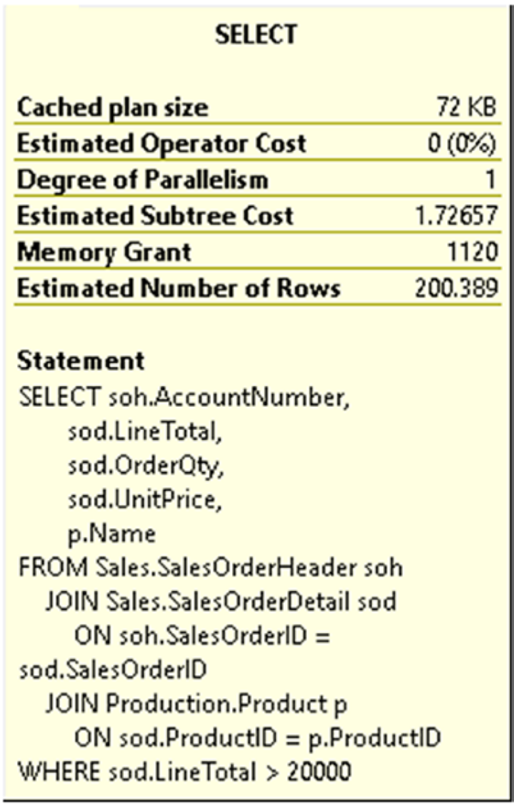
Execution plans show two different flows of information. Reading from the left side, you can see the logical flow, starting with the `SELECT` operator and proceeding through each of the execution steps. Starting from the right side and reading the other way is the physical flow of information, pulling data from the `Clustered Index Scan` operator first and then proceeding to each subsequent step. Most of the time, reading in the direction of the physical flow of data is more applicable to understanding what's happening with the execution plan, but not always. Sometimes the only way to understand what is happening in an execution plan is to read it in the logical processing order, left to right. Each step represents an operation performed to get the final output of the query.

An important aspect of execution plans are the values displayed in them. There are a number that we'll be using throughout the book, but the one that is most immediately apparent is `Cost`, which shows the estimated cost percentage. You can see it in Figure 7-2. The `SELECT` operator on the left has a `Cost` value of 0%, and the `Clustered Index Scan` operation on the right has a `Cost` value of 60%. These costs must be thought of as simply cost units. They are not a literal measure of performance of any kind. They are values assigned by or calculated by the query optimizer. Nominally they represent a mathematical construct of I/O and CPU use. However, they do not represent literal I/O and CPU use. These values are always estimated values, and the units are simply cost units. That's a vital aspect of understanding that we need to establish up front.

Some of the aspects of a query execution represented by an execution plan are as follows:

- If a query consists of a batch of multiple queries, the execution plan for each query will be displayed in the order of execution. Each execution plan in the batch will have a relative estimated cost, with the total cost of the whole batch being 100 percent.

- Every icon in an execution plan represents an operator. They will each have a relative estimated cost, with the total cost of all the nodes in an execution plan being 100 percent. (Although inaccuracies in statistics, or even bugs in SQL Server, can lead to situations where you see costs more than 100 percent, these are mostly seen in older versions of SQL Server.)
- Usually the first physical operator in an execution represents a data retrieval mechanism from a database object (a table or an index). For example, in the execution plan in Figure 7-2, the three starting points represent retrievals from the `SalesOrderHeader`, `SalesOrderDetail`, and `Product` tables.
- Data retrieval will usually be either a table operation or an index operation. For example, in the execution plan in Figure 7-2, all three data retrieval steps are index operations.
- Data retrieval on an index will be either an index scan or an index seek. For example, you can see a clustered index scan, a clustered index seek, and an index scan in Figure 7-2.
- The naming convention for a data retrieval operation on an index is `[Table Name].[Index Name]`.
- The logical flow of the plan is from left to right, just like reading a book in English. The data flows from right to left between operators and is indicated by a connecting arrow between the operators.
- The thickness of a connecting arrow between operators represents a graphical representation of the number of rows transferred.
- The joining mechanism between two operators in the same column will be a nested loop join, a hash match join, a merge join, or an adaptive join (added to SQL Server 2017 and Azure SQL Database). For example, in the execution plan shown in Figure 7-2, there is one hash and one loop join. (Join mechanisms are covered in more detail later.)
- Running the mouse over a node in an execution plan shows a pop-up window with some details. The tooltips are not very useful most of the time. Figure 7-3 shows an example.



SELECT

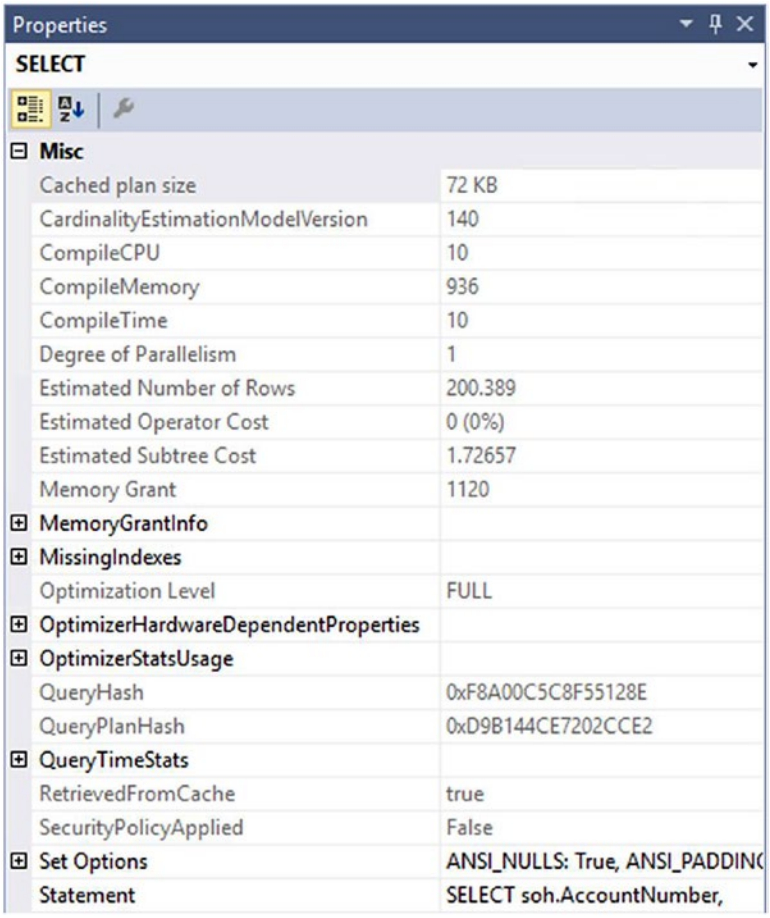
Cached plan size	72 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	1.72657
Memory Grant	1120
Estimated Number of Rows	200.389

Statement

```
SELECT soh.AccountNumber,
       sod.LineTotal,
       sod.OrderQty,
       sod.UnitPrice,
       p.Name
FROM Sales.SalesOrderHeader soh
   JOIN Sales.SalesOrderDetail sod
       ON soh.SalesOrderID =
sod.SalesOrderID
   JOIN Production.Product p
       ON sod.ProductID = p.ProductID
WHERE sod.LineTotal > 20000
```

Figure 7-3. *Tooltip sheet from an execution plan operator*

- A complete set of details about an operator is available in the Properties window, as shown in Figure 7-4, which you can open by right-clicking the operator and selecting Properties from the context menu.
- An operator detail shows both physical and logical operation types at the top. Physical operations represent those actually used by the storage engine, while the logical operations are the constructs used by the optimizer to build the estimated execution plan. If logical and physical operations are the same, then only the physical operation is shown. It also displays other useful information, such as row count, I/O cost, CPU cost, and so on.
- Reading through the properties on many of the operators can be necessary to understand how a query is being executed within SQL Server to better know how to tune that query.



⊟ **Misc**	
Cached plan size	72 KB
CardinalityEstimationModelVersion	140
CompileCPU	10
CompileMemory	936
CompileTime	10
Degree of Parallelism	1
Estimated Number of Rows	200.389
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	1.72657
Memory Grant	1120
⊞ **MemoryGrantInfo**	
⊞ **MissingIndexes**	
Optimization Level	FULL
⊞ **OptimizerHardwareDependentProperties**	
⊞ **OptimizerStatsUsage**	
QueryHash	0xF8A00C5C8F55128E
QueryPlanHash	0xD9B144CE7202CCE2
⊞ **QueryTimeStats**	
RetrievedFromCache	true
SecurityPolicyApplied	False
⊞ **Set Options**	ANSI_NULLS: True, ANSI_PADDIN
Statement	SELECT soh.AccountNumber,

Figure 7-4. *Select operator properties*

It's worth noting that in actual execution plans produced in SQL Server 2017 Management Studio, you can also see the execution time statistics for the query as part of the query plan. They're visible in Figure 7-4 in the section QueryTimeStats. This provides an additional mechanism for measuring query performance. You can also see wait statistics within the execution plan when those statistics exceed 1ms. Any waits less than that won't show up in an execution plan.

Identifying the Costly Steps in an Execution Plan

The most immediate approach in the execution plan is to find out which steps are relatively costly. These steps are the starting point for your query optimization. You can choose the starting steps by adopting the following techniques:

- Each node in an execution plan shows its relative estimated cost in the complete execution plan, with the total cost of the whole plan being 100 percent. Therefore, focus attention on the nodes with the highest relative cost. For example, the execution plan in Figure 7-2 has one step with 59 percent estimated cost.
- An execution plan may be from a batch of statements, so you may also need to find the most costly estimated statement. In Figure 7-2 you can see at the top of the plan the text "Query 1." In a batch situation, there will be multiple plans, and they will be numbered in the order they occurred within the batch.
- Observe the thickness of the connecting arrows between nodes. A thick connecting arrow indicates a large number of rows being transferred between the corresponding nodes. Analyze the node to the left of the arrow to understand why it requires so many rows. Check the properties of the arrows too. You may see that the estimated rows and the actual rows are different. This can be caused by out-of-date statistics, among other things. If you see thick arrows through much of the plan and then a thin arrow at the end, it might be possible to modify the query or indexes to get the filtering done earlier in the plan.
- Look for hash join operations. For small result sets, a nested loop join is usually the preferred join technique. You will learn more about hash joins compared to nested loop joins later in this chapter. Just remember that hash joins are not necessarily bad, and loop joins are not necessarily good. It does depend on the amounts of data being returned by the query.
- Look for key lookup operations. A lookup operation for a large result set can cause a large number of random reads. I will cover key lookups in more detail in Chapter 11.
- There may be warnings, indicated by an exclamation point on one of the operators, which are areas of immediate concern. These can be caused by a variety of issues, including a join without join criteria or an index or a table with missing statistics. Usually resolving the warning situation will help performance.

- Look for steps performing a sort operation. This indicates that the data was not retrieved in the correct sort order. Again, this may not be an issue, but it is an indicator of potential problems, possibly a missing or incorrect index. Ensuring that data is sorted in a specified manner using `ORDER BY` is not problematic, but sorts can lead to reduced performance.
- Watch for operators that may be placing additional load on the system such as table spools. They may be necessary for the operation of the query, or they may indicate an improperly written query or badly designed indexes.
- The default cost threshold for parallel query execution is an estimated cost of 5, and that's very low. Watch for parallel operations where they are not warranted. Just remember that the estimated costs are numbers assigned by the query optimizer representing a mathematical model of CPU and I/O but are not actual measures.

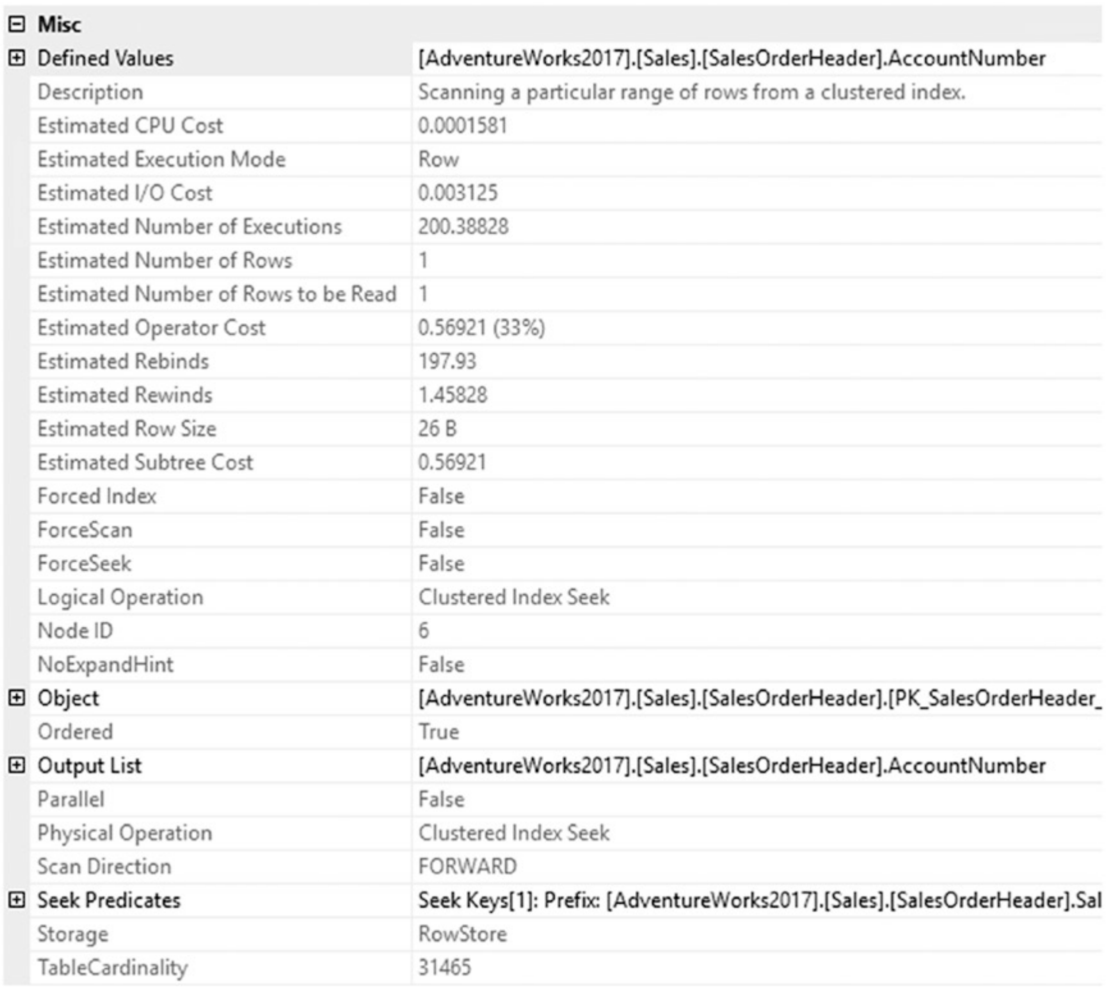
Analyzing Index Effectiveness

To examine a costly step in an execution plan further, you should analyze the data retrieval mechanism for the relevant table or index. First, you should check whether an index operation is a seek or a scan. Usually, for best performance, you should retrieve as few rows as possible from a table, and an index *seek* is frequently the most efficient way of accessing a small number of rows. A *scan* operation usually indicates that a larger number of rows have been accessed. Therefore, it is generally preferable to seek rather than scan. However, this is not saying that seeks are inherently good and scans are inherently bad. The mechanisms of data retrieval need to accurately reflect the needs of the query. A query retrieving all rows from a table will benefit from a scan where a seek for the same query would lead to poor performance. The key here is understanding the details of the operations through examination of the properties of the operators to understand why the optimizer made the choices that it did.

Next, you want to ensure that the indexing mechanism is properly set up. The query optimizer evaluates the available indexes to discover which index will retrieve data from the table in the most efficient way. If a desired index is not available, the optimizer uses the next best index. For best performance, you should always ensure that the best index is used in a data retrieval operation. You can judge the index effectiveness (whether the best index is used or not) by analyzing the Argument section of a node detail for the following:

- A data retrieval operation
- A join operation

Let's look at the data retrieval mechanism for the `SalesOrderHeader` table in the estimated execution plan. Figure 7-5 shows the operator properties.



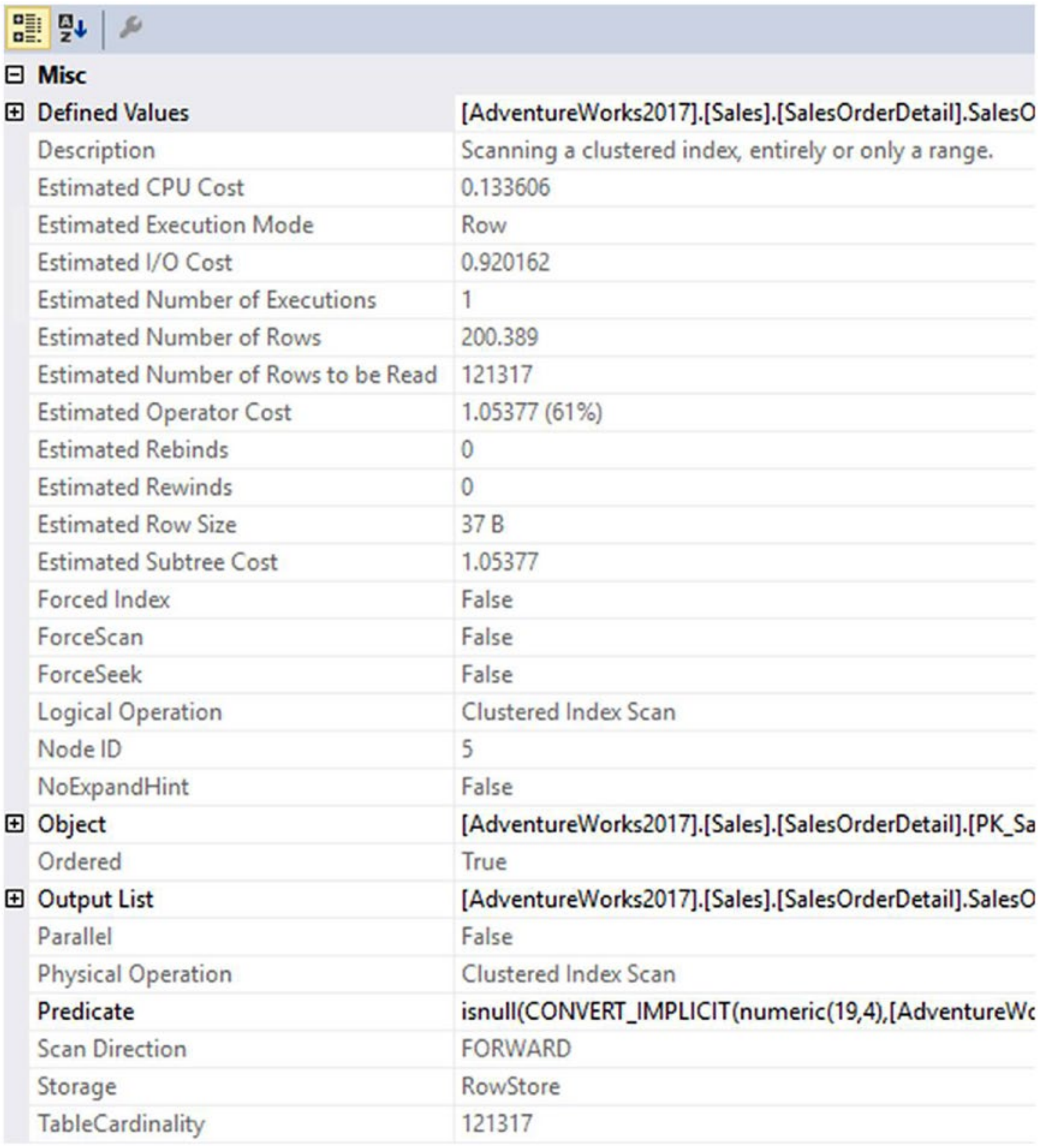
Misc	
⊞ Defined Values	[AdventureWorks2017].[Sales].[SalesOrderHeader].AccountNumber
Description	Scanning a particular range of rows from a clustered index.
Estimated CPU Cost	0.0001581
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	200.38828
Estimated Number of Rows	1
Estimated Number of Rows to be Read	1
Estimated Operator Cost	0.56921 (33%)
Estimated Rebinds	197.93
Estimated Rewinds	1.45828
Estimated Row Size	26 B
Estimated Subtree Cost	0.56921
Forced Index	False
ForceScan	False
ForceSeek	False
Logical Operation	Clustered Index Seek
Node ID	6
NoExpandHint	False
⊞ Object	[AdventureWorks2017].[Sales].[SalesOrderHeader].[PK_SalesOrderHeader_
Ordered	True
⊞ Output List	[AdventureWorks2017].[Sales].[SalesOrderHeader].AccountNumber
Parallel	False
Physical Operation	Clustered Index Seek
Scan Direction	FORWARD
⊞ Seek Predicates	Seek Keys[1]: Prefix: [AdventureWorks2017].[Sales].[SalesOrderHeader].Sal
Storage	RowStore
TableCardinality	31465

Figure 7-5. *Data retrieval mechanism for the SalesOrderHeader table*

In the operator properties for the `SalesOrderHeader` table, the `Object` property specifies the index used, `PK_SalesOrderHeader_SalesOrderID`. It uses the following naming convention: `[Database].[Owner].[Table Name].[Index Name]`. The `Seek Predicates` property specifies the column, or columns, used to find keys in the index. The `SalesOrderHeader` table is joined with the `SalesOrderDetail` table on the

SalesOrderld column. The SEEK works on the fact that the join criteria, SalesOrderld, is the leading edge of the clustered index and primary key, PK_SalesOrderHeader.

Sometimes you may have a different data retrieval mechanism. Instead of the Seek Predicates property you saw in Figure 7-5, Figure 7-6 shows a simple predicate, indicating a totally different mechanism for retrieving the data.



Misc	
Defined Values	[AdventureWorks2017].[Sales].[SalesOrderDetail].SalesO
Description	Scanning a clustered index, entirely or only a range.
Estimated CPU Cost	0.133606
Estimated Execution Mode	Row
Estimated I/O Cost	0.920162
Estimated Number of Executions	1
Estimated Number of Rows	200.389
Estimated Number of Rows to be Read	121317
Estimated Operator Cost	1.05377 (61%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	37 B
Estimated Subtree Cost	1.05377
Forced Index	False
ForceScan	False
ForceSeek	False
Logical Operation	Clustered Index Scan
Node ID	5
NoExpandHint	False
Object	[AdventureWorks2017].[Sales].[SalesOrderDetail].[PK_Sa
Ordered	True
Output List	[AdventureWorks2017].[Sales].[SalesOrderDetail].SalesO
Parallel	False
Physical Operation	Clustered Index Scan
Predicate	isnull(CONVERT_IMPLICIT(numeric(19,4),[AdventureWc
Scan Direction	FORWARD
Storage	RowStore
TableCardinality	121317

Figure 7-6. *A variation of the data retrieval mechanism, a scan*

In the properties in Figure 7-6, there is no seek predicate. Because of the function being performed on the column, the ISNULL, and the CONVERT_IMPLICIT, the entire table must be checked for the existence of the Predicate value.

```
isnull(CONVERT_IMPLICIT(numeric(19,4),[AdventureWorks2017].[Sales].
[SalesOrderDetail].[UnitPrice] as [sod].[UnitPrice],0)*((1.0)-CONVERT_IM
PLICIT(numeric(19,4),[AdventureWorks2017].[Sales].[SalesOrderDetail].
[UnitPriceDiscount] as [sod].[UnitPriceDiscount],0))*CONVERT_IMPLICIT(nu
meric(5,0),[AdventureWorks2017].[Sales].[SalesOrderDetail].[OrderQty] as
[sod].[OrderQty],0),(0.000000))>(20000.000000)
```

Because a calculation is being performed on the data, the index doesn't store the results of the calculation, so instead of simply looking information up on the index, you have to scan all the data, perform the calculation, and then check that the data matches the values that we're looking for.

Analyzing Join Effectiveness

In addition to analyzing the indexes used, you should examine the effectiveness of join strategies decided by the optimizer. SQL Server uses four types of joins.

- Hash joins
- Merge joins
- Nested loop joins
- Adaptive joins

In many simple queries affecting a small set of rows, nested loop joins are far superior to both hash and merge joins. As joins get more complicated, the other join types are used where appropriate. None of the join types is by definition bad or wrong. You're primarily looking for places where the optimizer may have chosen a type not compatible with the data in hand. This is usually caused by discrepancies in the statistics available to the optimizer when it's deciding which of the types to use.

Hash Join

To understand SQL Server's hash join strategy, consider the following simple query:

```
SELECT p.Name AS ProductName,
       pc.Name AS ProductCategoryName
FROM Production.Product p
    JOIN Production.ProductCategory pc
        ON p.ProductSubcategoryID = pc.ProductCategoryID;
```

Table 7-1 shows the two tables' indexes and number of rows.

Table 7-1. *Indexes and Number of Rows of the Products and ProductCategory Tables*

Table	Indexes	Number of Rows
`Product`	Clustered index on `ProductID`	504
`ProductCategory`	Clustered index on `ProductCategoryId`	4

Figure 7-7 shows the execution plan for the preceding query.

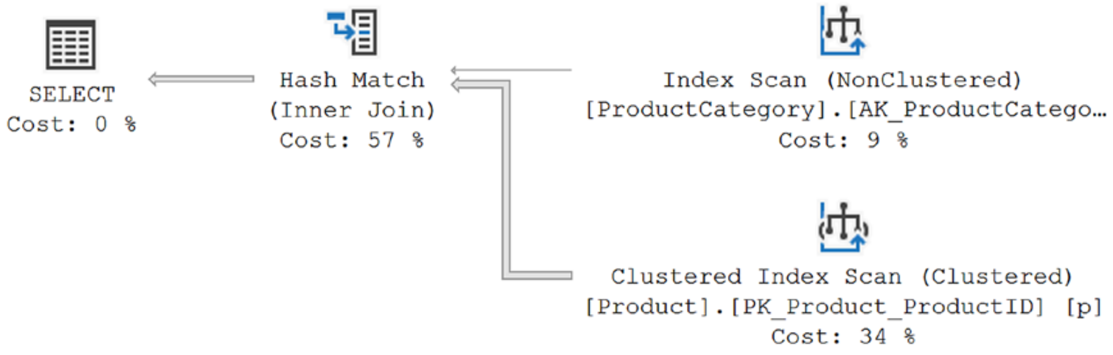

SELECT
Cost: 0 %
Hash Match
(Inner Join)
Cost: 57 %
Index Scan (NonClustered)
[ProductCategory].[AK_ProductCatego…
Cost: 9 %
Clustered Index Scan (Clustered)
[Product].[PK_Product_ProductID] [p]
Cost: 34 %

Figure 7-7. *Execution plan with a hash join*

You can see that the optimizer used a hash join between the two tables.

A hash join uses the two join inputs as a *build input* and a *probe input*. The build input is represented by the top input in the execution plan, and the probe input is the bottom input. Usually the smaller of the two inputs serves as the build input because it's going to be stored on the system, so the optimizer attempts to minimize the memory used.

The hash join performs its operation in two phases: the *build phase* and the *probe phase.* In the most commonly used form of hash join, the *in-memory hash join,* the entire build input is scanned or computed, and then a hash table is built in memory. Each row from the outer input is inserted into a hash bucket depending on the hash value computed for the *hash key* (the set of columns in the equality predicate). A hash is just a mathematical construct run against the values in question and used for comparison purposes.

This build phase is followed by the probe phase. The entire probe input is scanned or computed one row at a time, and for each probe row, a hash key value is computed. The corresponding hash bucket is scanned for the hash key value from the probe input, and the matches are produced. Figure 7-8 illustrates the process of an in-memory hash join.

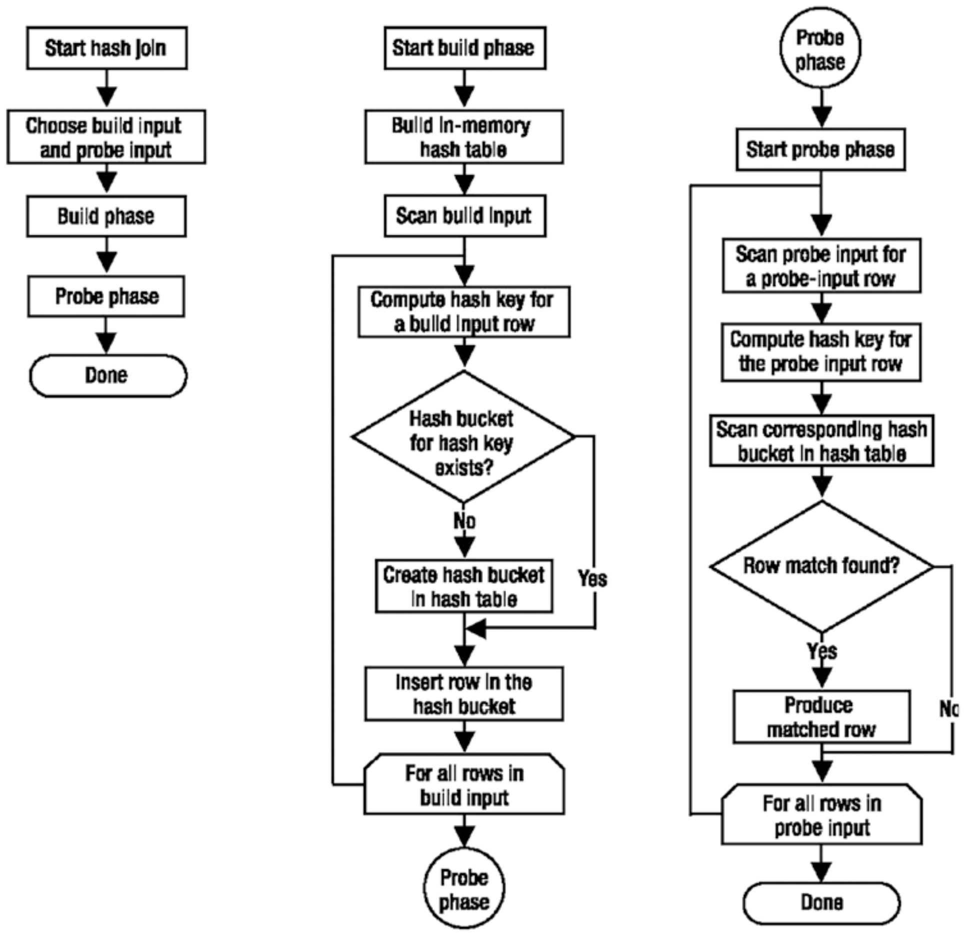

Start hash join
Choose build input and probe input
Build phase
Probe phase
Done
Start build phase
Build in-memory hash table
Scan build input
Compute hash key for a build input row
Hash bucket for hash key exists?
No
Yes
Create hash bucket in hash table
Insert row in the hash bucket
For all rows in build input
Probe phase
Probe phase
Start probe phase
Scan probe input for a probe-input row
Compute hash key for the probe input row
Scan corresponding hash bucket in hash table
Row match found?
Yes
No
Produce matched row
For all rows in probe input
Done

Figure 7-8. *Workflow for an in-memory hash join*

The query optimizer uses hash joins to process large, unsorted, nonindexed inputs efficiently. Let's now look at the next type of join: the merge join.

Merge Join

In the previous case, input from the Product table is larger, and the table is not indexed on the joining column (ProductCategoryID). Using the following simple query, you can see different behavior:

```
SELECT pm.Name AS ProductModelName,
       pmpd.CultureID
FROM Production.ProductModel pm
    JOIN Production.ProductModelProductDescriptionCulture pmpd
        ON pm.ProductModelID = pmpd.ProductModelID;
```

Figure 7-9 shows the resultant execution plan for this query.

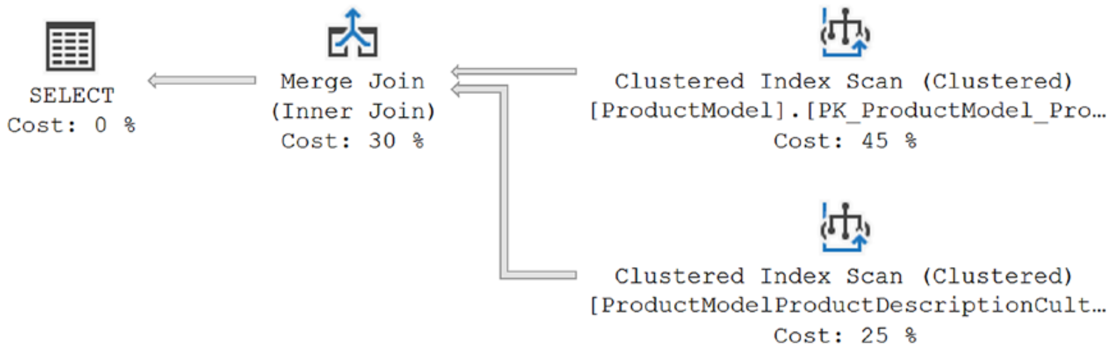
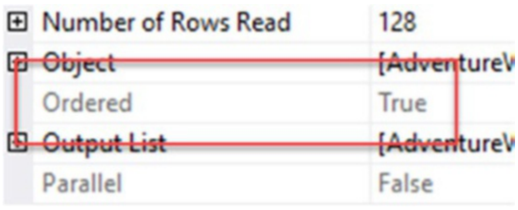

SELECT
Cost: 0 %
Merge Join
(Inner Join)
Cost: 30 %
Clustered Index Scan (Clustered)
[ProductModel].[PK_ProductModel_Pro…
Cost: 45 %
Clustered Index Scan (Clustered)
[ProductModelProductDescriptionCult…
Cost: 25 %

Figure 7-9. *Execution plan with a merge join*

For this query, the optimizer used a merge join between the two tables. A merge join requires both join inputs to be sorted on the merge columns, as defined by the join criterion. If indexes are available on both joining columns, then the join inputs are sorted by the index. Since each join input is sorted, the merge join gets a row from each input and compares them for equality. A matching row is produced if they are equal. This process is repeated until all rows are processed.

In situations where the data is ordered by an index, a merge join can be one of the fastest join operations, but if the data is not ordered and the optimizer still chooses to perform a merge join, then the data has to be ordered by an extra operation, a sort. This can make the merge join slower and more costly in terms of memory and I/O resources. This can be made even worse if the memory allocation is inaccurate and the sort spills to the disk in tempdb.

In this case, the query optimizer found that the join inputs were both sorted (or indexed) on their joining columns. You can see this in the properties of the `Index Scan` operators, as shown in Figure 7-10.



⊞ Number of Rows Read	128
⊞ Object	[AdventureV
Ordered	True
⊞ Output List	[AdventureV
Parallel	False

Figure 7-10. *Properties of Clustered Index Scan showing that the data is ordered*

As a result of the data being ordered by the indexes in use, the merge join was chosen as a faster join strategy than any other join in this situation.

Nested Loop Join

The next type of join I'll cover here is the nested loop join. For better performance, you should always strive to access a limited number of rows from individual tables. To understand the effect of using a smaller result set, decrease the join inputs in your query as follows:

```
SELECT pm.Name AS ProductName,
       pmpd.CultureID
FROM Production.ProductModel pm
    JOIN Production.ProductModelProductDescriptionCulture pmpd
        ON pm.ProductModelID = pmpd.ProductModelID
WHERE pm.Name = 'HL Mountain Front Wheel';
```

Figure 7-11 shows the resultant execution plan of the new query.

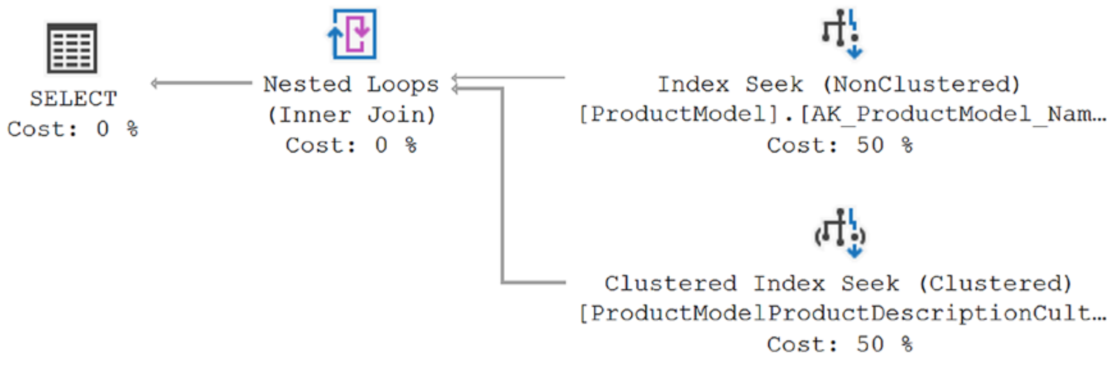

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[ProductModel].[AK_ProductModel_Nam...
Cost: 50 %
Clustered Index Seek (Clustered)
[ProductModelProductDescriptionCult...
Cost: 50 %

Figure 7-11. *Execution plan with a nested loop join*

As you can see, the optimizer used a nested loop join between the two tables.

A nested loop join uses one join input as the outer input table and the other as the inner input table. The outer input table is shown as the top input in the execution plan, and the inner input table is shown as the bottom input table. The outer loop consumes the outer input table row by row. The inner loop, executed for each outer row, searches for matching rows in the inner input table.

Nested loop joins are highly effective if the outer input is quite small and the inner input is larger but indexed. In many simple queries affecting a small set of rows, nested loop joins are far superior to both hash and merge joins. Joins operate by gaining speed through other sacrifices. A loop join can be fast because it uses memory to take a small set of data and compare it quickly to a second set of data. A merge join similarly uses memory and a bit of `tempdb` to do its ordered comparisons. A hash join uses memory and `tempdb` to build out the hash tables for the join. Although a loop join can be faster at small data sets, it can slow down as the data sets get larger or there aren't indexes to support the retrieval of the data. That's why SQL Server has different join mechanisms.

Even for small join inputs, such as in the previous query, it's important to have an index on the joining columns. As you saw in the preceding execution plan, for a small set of rows, indexes on joining columns allow the query optimizer to consider a nested loop join strategy. A missing index on the joining column of an input will force the query optimizer to use a hash join instead.

Table 7-2 summarizes the use of the three join types.

Table 7-2. *Characteristics of the Three Join Types*

Join Type	Index on Joining Columns	Usual Size of Joining Tables	Presorted	Join Clause
Hash	Inner table: Not indexed Outer table: Optional Optimal condition: Small outer table, large inner table	Any	No	Equi-join
Merge	Both tables: Must Optimal condition: Clustered or covering index on both	Large	Yes	Equi-join
Nested loop	Inner table: Must Outer table: Preferable	Small	Optional	All

Note The outer table is usually the smaller of the two joining tables in the hash and loop joins.

I will cover index types, including clustered and covering indexes, in Chapter 8.

Adaptive Join

The adaptive join was introduced in Azure SQL Database and in SQL Server 2017. It's a new join type that can choose between either a nested loop join or a hash join on the fly. As of this writing, it's applicable only to columnstore indexes, but that may change in the future. To see this in action, I'm going to create a table with a clustered columnstore index.

```
SELECT *
INTO dbo.TransactionHistory
FROM Production.TransactionHistory AS th;
```

```
CREATE CLUSTERED COLUMNSTORE INDEX ClusteredColumnStoreTest
ON dbo.TransactionHistory;
```

With this table and index in place and our compatibility mode set correctly, we can run a simple query that takes advantage of the clustered columnstore index.

```
SELECT p.Name,
       th.Quantity
FROM dbo.TransactionHistory AS th
    JOIN Production.Product AS p
        ON p.ProductID = th.ProductID
WHERE th.Quantity > 550;
```

Capturing an actual execution plan from the query, we'll see Figure 7-12.

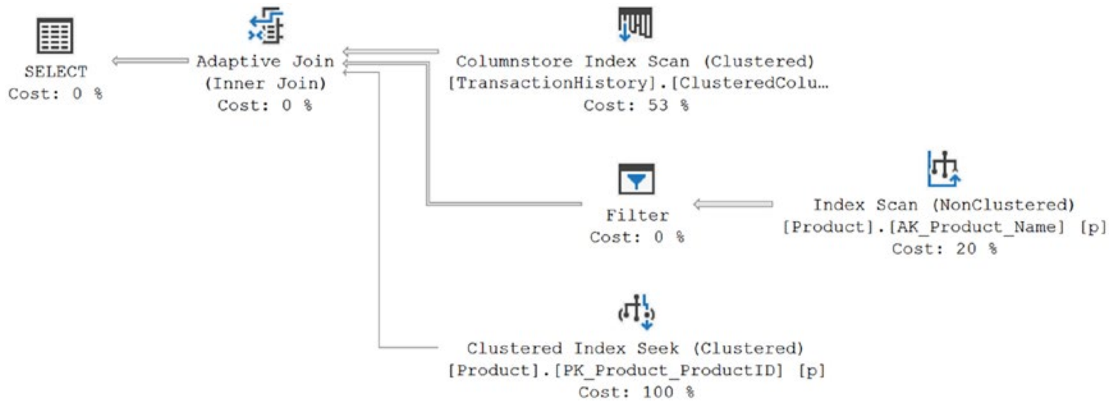
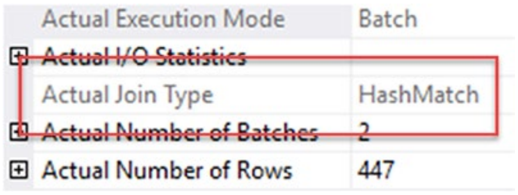

SELECT
Cost: 0 %
Adaptive Join
(Inner Join)
Cost: 0 %
Columnstore Index Scan (Clustered)
[TransactionHistory].[ClusteredColu…
Cost: 53 %
Filter
Cost: 0 %
Index Scan (NonClustered)
[Product].[AK_Product_Name] [p]
Cost: 20 %
Clustered Index Seek (Clustered)
[Product].[PK_Product_ProductID] [p]
Cost: 100 %

Figure 7-12. *Execution plan with an adaptive join*

The hash join or nested loops join used by the adaptive join function exactly as defined earlier. The difference is that the adaptive join can make a determination as to which join type will be more efficient in a given situation. The way it works is that it starts out building an adaptive buffer, which is hidden. If the row threshold is exceeded, rows flow into a regular hash table. The remaining rows are loaded to the hash table, ready for the probe process, just as described. If all the rows are loaded into the adaptive buffer and that number falls below the row threshold, then that buffer is used as the outer reference of a nested loops join.

Each join is shown as a separate branch below the `Adaptive Join` operator, as you can see in Figure 7-12. The first branch below the `Adaptive Join` is for the hash join. In this case, an `Index Scan` operator and a `Filter` operator satisfy the needs of the query should a hash join be used. The second branch below the `adaptive join` is for the nested loops join. Here that would be the `Clustered Index Seek` operation.

The plan is generated and stored in cache, with both possible branches. Then the query engine will determine which of the branches to work down depending on the result set in question. You can see the choice that was made by looking to the properties of the `Adaptive Join` operator, as shown in Figure 7-13.



Actual Execution Mode	Batch
⊞ Actual I/O Statistics	
Actual Join Type	HashMatch
⊞ Actual Number of Batches	2
⊞ Actual Number of Rows	447

Figure 7-13. *Properties of the Adaptive Join operator showing the actual join type*

The threshold at which this join switches between hash match and nested loops is calculated at the time the plan is compiled. That is stored with the plan in the properties as `AdaptiveThresholdRows`. As a query executes and it is determined that it has either met, exceeded, or not met the threshold, processing continues down the correct branch of the adaptive join. No plan recompile is needed for this to happen. Recompiles are discussed further in Chapter 16.

Adaptive joins enhance performance fairly radically when the data set is such that a nested loop would drastically outperform the hash match. While there is a cost associated with building and then not using the hash match, this is offset by the enhanced performance of the nested loops join for smaller data sets. When the data set is large, this process doesn't negatively affect the hash join operation in any way.

While technically this does not represent a fundamentally new type of join, the behavior of dynamically switching between the two core types, nested loops and hash match, in my opinion, makes this effectively a new join type. Add to that the fact that you now have a new operator, the `Adaptive Join` operator, and neither the nested loops nor the hash match is visible, and it certainly looks like a new join type.

Actual vs. Estimated Execution Plans

There are estimated and actual execution plans. To a degree, they are interchangeable. But, the actual plan carries with it information from the execution of the query, specifically the row counts affected and some other information, that is not available in the estimated plans. This information can be extremely useful, especially when trying to understand statistic estimations. For that reason, actual execution plans are preferred when tuning queries.

Unfortunately, you won't always be able to access them. You may not be able to execute a query, say in a production environment. You may have access only to the plan from cache, which contains no runtime information. So, there are situations where the estimated plan is what you will have to work with. However, it's usually preferable to get the actual plans because of the runtime metrics gathered there.

There are other situations where the estimated plans will not work at all. Consider the following stored procedure:

```
CREATE OR ALTER PROC p1
AS
CREATE TABLE t1 (c1 INT);

INSERT INTO t1
SELECT ProductID
FROM Production.Product;

SELECT *
FROM t1;

DROP TABLE t1;
GO
```

You may try to use SHOWPLAN_XML to obtain the estimated XML execution plan for the query as follows:

```
SET SHOWPLAN_XML ON;
GO
EXEC p1 ;
GO
SET SHOWPLAN_XML OFF;
GO
```

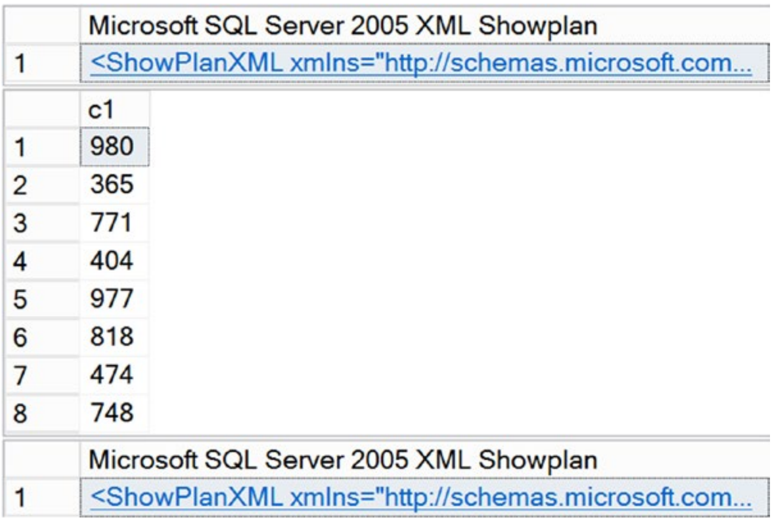
But this fails with the following error:

```
Msg 208, Level 16, State 1, Procedure p1, Line 249
Invalid object name 't1'.
```

Since SHOWPLAN_XML doesn't actually execute the query, the query optimizer can't generate an execution plan for INSERT and SELECT statements on the table (t1) because it doesn't exist until the query is executed. Instead, you can use STATISTICS XML as follows:

```
SET STATISTICS XML ON;
GO
EXEC p1;
GO
SET STATISTICS XML OFF;
GO
```

Since STATISTICS XML executes the query, the table is created and accessed within the query, which is all captured by the execution plan. Figure 7-14 shows the results of the query and the two plans for the two statements within the procedure provided by STATISTICS XML.



```
   Microsoft SQL Server 2005 XML Showplan
1  <ShowPlanXML xmlns="http://schemas.microsoft.com...

   c1
1  980
2  365
3  771
4  404
5  977
6  818
7  474
8  748

   Microsoft SQL Server 2005 XML Showplan
1  <ShowPlanXML xmlns="http://schemas.microsoft.com...
```

Figure 7-14. *STATISTICS PROFILE output*

Tip Remember to switch Query ➤ Show Execution Plan off in Management Studio, or you will see the graphical, rather than textual, execution plan.

Plan Cache

Another place to access execution plans is to read them directly from the memory space where they are stored, the plan cache. Dynamic management views and functions are provided from SQL Server to access this data. All plans stored in the cache are estimated plans. To see a listing of execution plans in cache, run the following query:

```
SELECT p.query_plan,
       t.text
FROM sys.dm_exec_cached_plans r
    CROSS APPLY sys.dm_exec_query_plan(r.plan_handle) p
    CROSS APPLY sys.dm_exec_sql_text(r.plan_handle) t;
```

The query returns a list of XML execution plan links. Opening any of them will show the execution plan. These execution plans are the compiled plans, but they contain no execution metrics. Working further with columns available through the dynamic management views will allow you to search for specific procedures or execution plans.

While not having the runtime data is somewhat limiting, having access to execution plans, even as the query is executing, is an invaluable resource for someone working on performance tuning. As mentioned earlier, you might not be able to execute a query in a production environment, so getting any plan at all is useful.

Covered in Chapter 11, you can also retrieve plans from the Query Store. Like the plans stored in cache, these are all estimated plans.

Execution Plan Tooling

While you've just started to see execution plans in action, you've only seen part of what's available to you to understand how these plans work. In addition to the XML information presented in the plans within SSMS as the graphical plans and their inherent properties, Management Studio offers some additional plan functionality that is worth knowing about in your quest to understand what any given execution plan is showing you about query performance.

Find Node

First, you can actually search within the operators of a plan to find particular values within the properties. Let's take the original query that we started the chapter with and generate a plan for it. Here is the query:

```
SELECT soh.AccountNumber,
       sod.LineTotal,
       sod.OrderQty,
       sod.UnitPrice,
       p.Name
FROM Sales.SalesOrderHeader soh
    JOIN Sales.SalesOrderDetail sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product p
        ON sod.ProductID = p.ProductID
WHERE sod.LineTotal > 20000;
```

After we generate the execution plan, using any means you prefer, right-click within the execution plan. A context menu comes up with lots of interesting resources for controlling the plan, as shown in Figure 7-15.

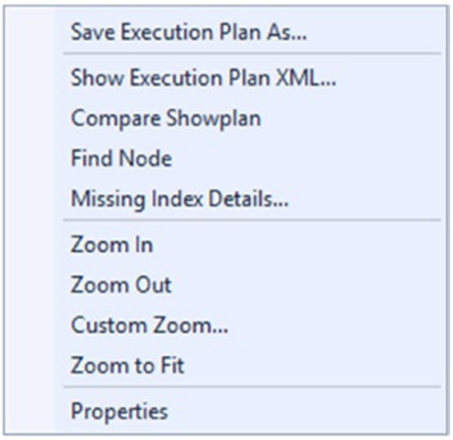


Figure 7-15. *Execution plan context menu*

If we select the Find Node menu choice, a new interface appears in the upper-right corner of the execution plan, similar to Figure 7-16.

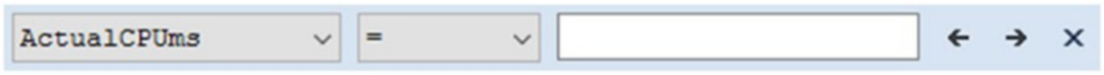


Figure 7-16. *Find Node interface*

On the left side are all the properties for all the operators. You can pick any property you want to search for. You can then choose an operator. The default shown in Figure 7-16 is the equal operator. There is also a `Contains` operator. Finally you type in a value. Clicking the left or right arrow will find the operator that matches your criteria. Clicking again will move to the next operator, if any, allowing you to work your way through an execution plan that is large and complex without having to visually search the properties of each operator on your own.

For example, we can look for any of the operators that reference the schema `Product`, as shown in Figure 7-17.

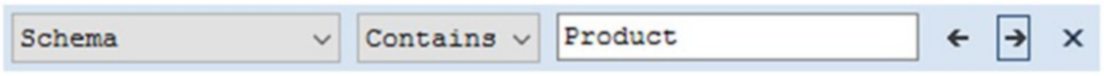


Figure 7-17. *Looking for any operators that have a Schema value that contains Product*

Clicking the right arrow will take you to the first operator that references the `Product` schema. In the example, it would first go to the `SELECT` operator, then the `Adaptive Join` operator, the `Filter` operator, and then both the `Index Scan` and the `Index Seek` operators. The only operator it would not select is `Columnstore Index Scan` because it's in the `TransactionHistory` schema.

Compare Plans

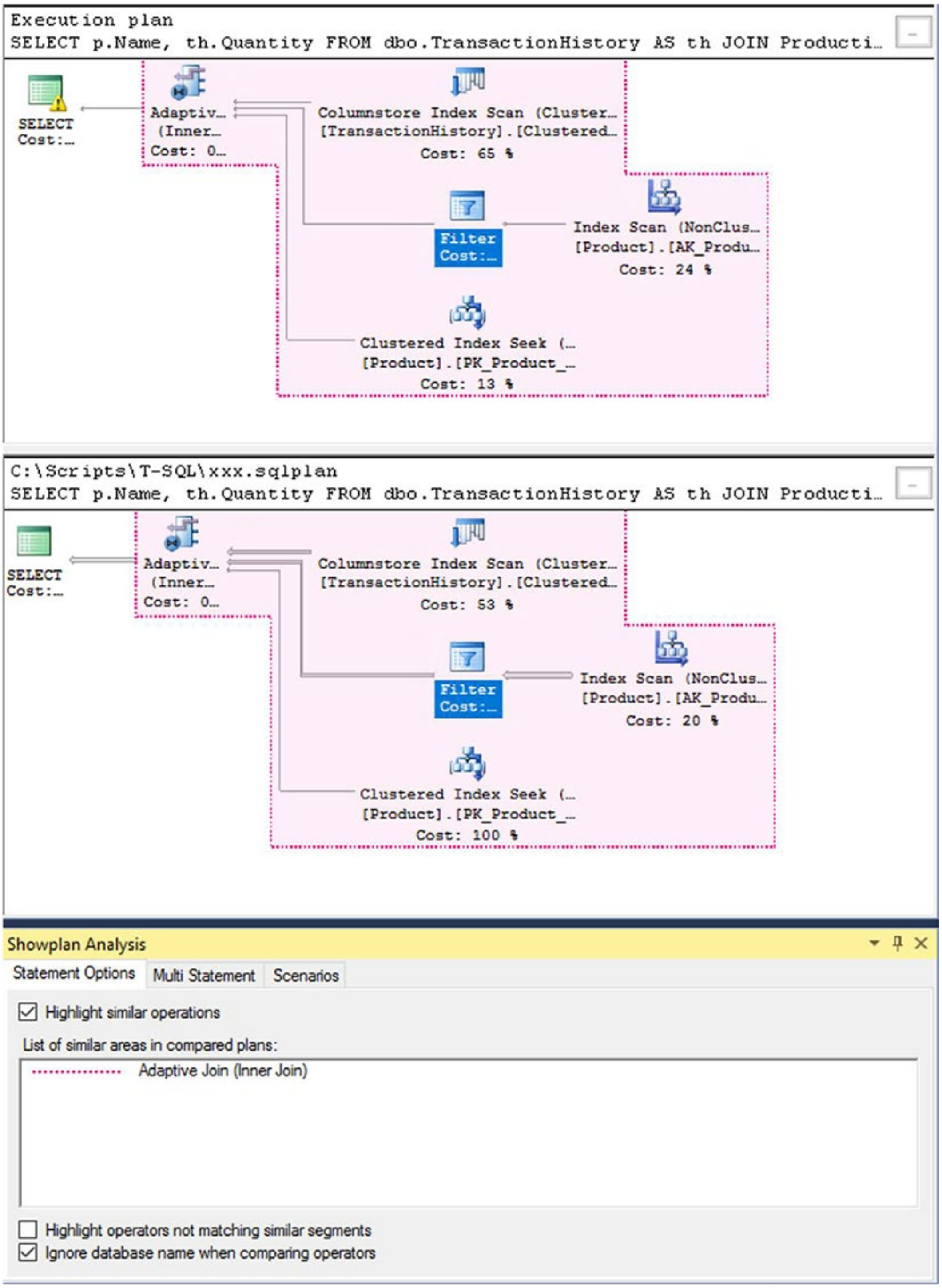
Sometimes you may be wondering what the difference is between two execution plans when it's not easily visible within the graphical plans. If we were to run the following queries, the plans would essentially look identical:

```
SELECT p.Name,
       th.Quantity
FROM dbo.TransactionHistory AS th
```

```
    JOIN Production.Product AS p
        ON p.ProductID = th.ProductID
WHERE th.Quantity > 550;

SELECT p.Name,
       th.Quantity
FROM dbo.TransactionHistory AS th
    JOIN Production.Product AS p
        ON p.ProductID = th.ProductID
WHERE th.Quantity > 35000;
```

There actually are some distinct differences in these plans, but they also look similar. Determining exactly what the differences are just using your eyes to compare them could lead to a lot of mistakes. Instead, we'll right-click in one of the plans and bring up the context menu from Figure 7-15. Use the top option to save one of the plans to a file. This is necessary. Then, right-click within the other plan to get the context menu again. Select the choice Compare Showplan. This will open a new window within SSMS that will look a lot like Figure 7-18.



☑ Highlight similar operations

List of similar areas in compared plans:

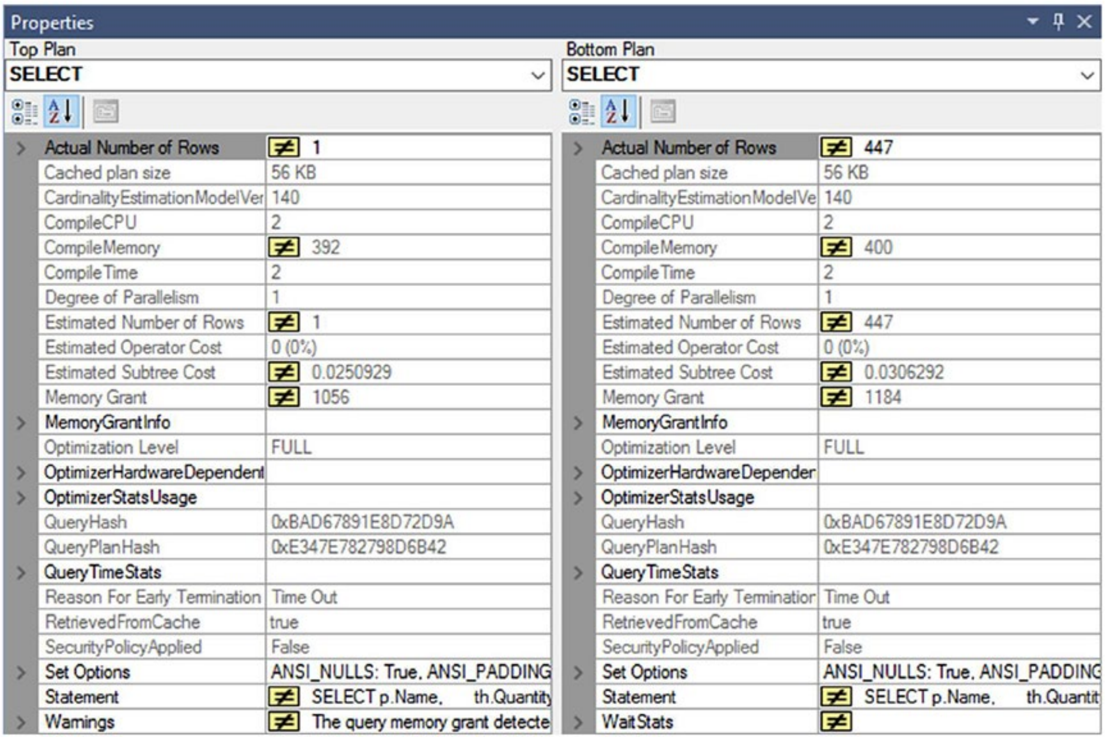
.... Adaptive Join (Inner Join)

☐ Highlight operators not matching similar segments
☑ Ignore database name when comparing operators

***Figure 7-18.** Execution plan comparison within SSMS*

What you're seeing are plans that are similar but with distinct differences. The area highlighted in pink are the similarities. Areas of the plan that are not highlighted, the SELECT operator in this case, are the larger differences. You can control the highlighting using the Statement options at the bottom of the screen.

Further, you can explore the properties of the operators. Right-clicking one and selecting the Properties menu choice will open a window like Figure 7-19.



Top Plan

SELECT

Actual Number of Rows	≠ 1
Cached plan size	56 KB
CardinalityEstimationModelVer	140
CompileCPU	2
CompileMemory	≠ 392
CompileTime	2
Degree of Parallelism	1
Estimated Number of Rows	≠ 1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	≠ 0.0250929
Memory Grant	≠ 1056
MemoryGrantInfo	
Optimization Level	FULL
OptimizerHardwareDependent	
OptimizerStatsUsage	
QueryHash	0xBAD67891E8D72D9A
QueryPlanHash	0xE347E782798D6B42
QueryTimeStats	
Reason For Early Termination	Time Out
RetrievedFromCache	true
SecurityPolicyApplied	False
Set Options	ANSI_NULLS: True, ANSI_PADDING
Statement	≠ SELECT p.Name, th.Quantity
Warnings	≠ The query memory grant detecte

Bottom Plan

SELECT

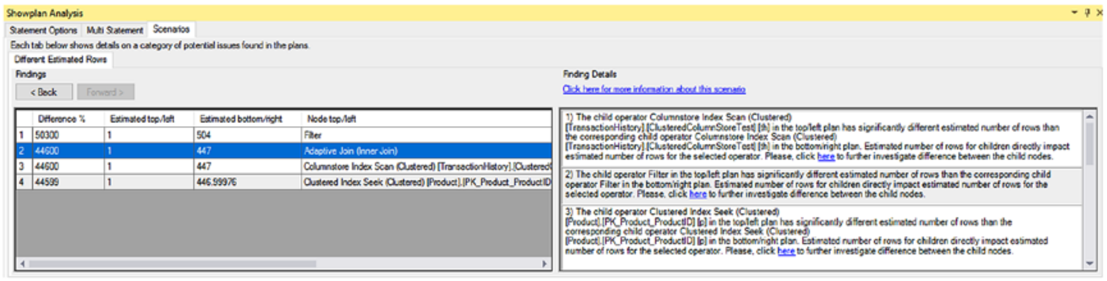
Actual Number of Rows	≠ 447
Cached plan size	56 KB
CardinalityEstimationModelVe	140
CompileCPU	2
CompileMemory	≠ 400
CompileTime	2
Degree of Parallelism	1
Estimated Number of Rows	≠ 447
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	≠ 0.0306292
Memory Grant	≠ 1184
MemoryGrantInfo	
Optimization Level	FULL
OptimizerHardwareDepender	
OptimizerStatsUsage	
QueryHash	0xBAD67891E8D72D9A
QueryPlanHash	0xE347E782798D6B42
QueryTimeStats	
Reason For Early Terminatior	Time Out
RetrievedFromCache	true
SecurityPolicyApplied	False
Set Options	ANSI_NULLS: True, ANSI_PADDING
Statement	≠ SELECT p.Name, th.Quantit
WaitStats	≠

Figure 7-19. *SELECT operator property differences between two plans*

You can see that properties that don't match have that bright yellow "does not equal" symbol on them. This allows you to easily find and see the differences between two execution plans.

Scenarios

Finally, one additional new tool is the ability of Management Studio to analyze your execution plans and point out possible issues with the plan. These are referred to as *scenarios* and are listed on the bottom of the screen shown in Figure 7-18. To see this functionality in action, Figure 7-20 shows the tab selected and one of the operators selected.



Each tab below shows details on a category of potential issues found in the plans.

	Difference %	Estimated top/left	Estimated bottom/right	Node top/left
1	50300	1	504	Filter
2	44600	1	447	Adaptive Join (Inner Join)
3	44600	1	447	Columnstore Index Scan (Clustered) [TransactionHistory].[Clustered
4	44599	1	446.99976	Clustered Index Seek (Clustered) [Product].[PK_Product_ProductID

1) The child operator Columnstore Index Scan (Clustered) [TransactionHistory].[ClusteredColumnStoreTest] [th] in the top/left plan has significantly different estimated number of rows than the corresponding child operator Columnstore Index Scan (Clustered) [TransactionHistory].[ClusteredColumnStoreTest] [th] in the bottom/right plan. Estimated number of rows for children directly impact estimated number of rows for the selected operator. Please, click here to further investigate difference between the child nodes.

2) The child operator Filter in the top/left plan has significantly different estimated number of rows than the corresponding child operator Filter in the bottom/right plan. Estimated number of rows for children directly impact estimated number of rows for the selected operator. Please, click here to further investigate difference between the child nodes.

3) The child operator Clustered Index Seek (Clustered) [Product].[PK_Product_ProductID] [p] in the top/left plan has significantly different estimated number of rows than the corresponding child operator Clustered Index Seek (Clustered) [Product].[PK_Product_ProductID] [p] in the bottom/right plan. Estimated number of rows for children directly impact estimated number of rows for the selected operator. Please, click here to further investigate difference between the child nodes.

Figure 7-20. *Different Estimated Rows scenario in Showplan Analysis window*

Currently Microsoft offers only a single scenario, but more may be available by the time you read this book. The scenario I have currently highlighted is Different Estimated Rows. This is directly related to common problems with missing, incorrect, or out-of-date statistics on columns and indexes. It's a common problem and one that we'll address in several of the chapters in the book, especially Chapter 13. Suffice to say that when there is a disparity between estimated and actual row counts, it can cause performance problems because the plans generated may be incorrect for the actual data.

On the left side of the screen are the operators that may have a disparity between estimated and actual rows. On the right are descriptions about why this disparity has been highlighted. We'll be exploring this in more detail later in the book.

You can also get to the Analysis screen when you capture a plan using XML STATISTICS or when you simply open a file containing a plan. Currently, you can't capture a plan within SSMS and get to the Showplan Analysis screen directly.

Live Execution Plans

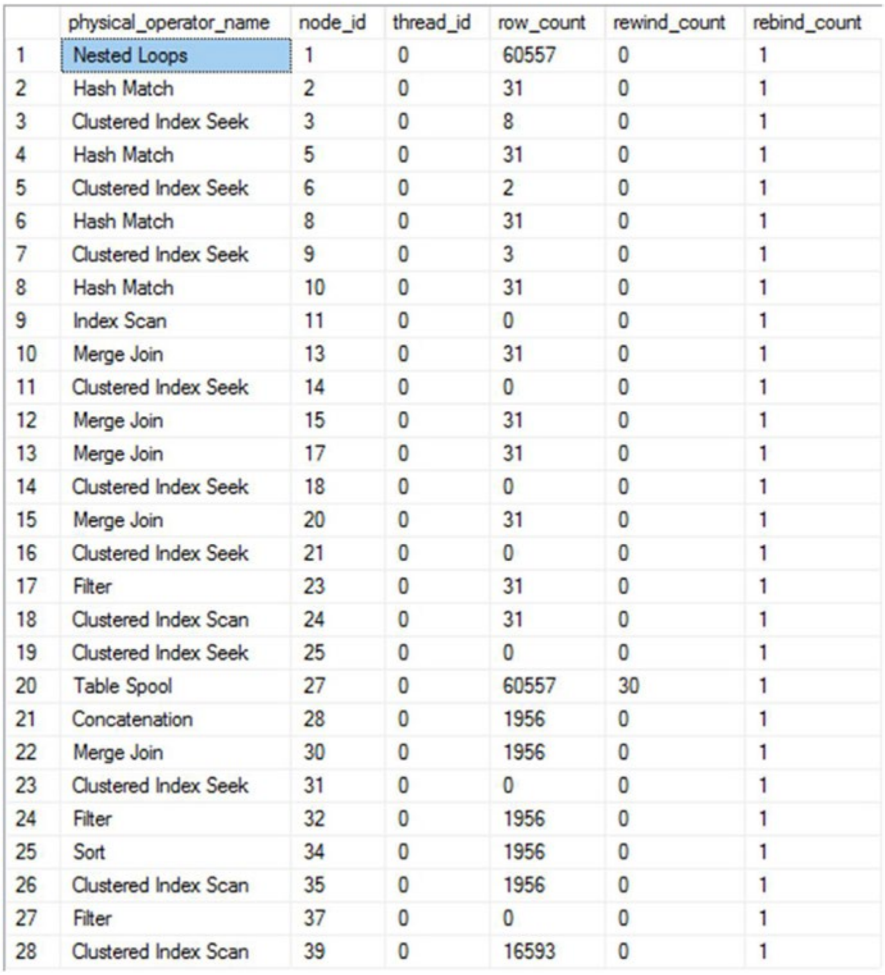
The official name is Live Query Statistics, but what you'll actually see is a live execution plan. Introduced in SQL Server 2014, the DMV `sys.dm_exec_query_profiles` actually allows you to see execution plan operations live, observing the number of rows processed by each operation in real time. However, in SQL Server 2014, and by default in other versions, you must be capturing an actual execution plan for this to work. Further, the query has to be somewhat long-running to see this in action. So, this is a query without JOIN criteria that creates Cartesian products, so it will take a little while to complete:

```
SELECT *
FROM sys.columns AS c,
     sys.syscolumns AS s;
```

Put that into one query window and execute it while capturing an actual execution plan. While it's executing, in a second query window, run this query:

```
SELECT deqp.physical_operator_name,
       deqp.node_id,
       deqp.thread_id,
       deqp.row_count,
       deqp.rewind_count,
       deqp.rebind_count
FROM sys.dm_exec_query_profiles AS deqp;
```

You'll see data similar to Figure 7-21.



	physical_operator_name	node_id	thread_id	row_count	rewind_count	rebind_count
1	Nested Loops	1	0	60557	0	1
2	Hash Match	2	0	31	0	1
3	Clustered Index Seek	3	0	8	0	1
4	Hash Match	5	0	31	0	1
5	Clustered Index Seek	6	0	2	0	1
6	Hash Match	8	0	31	0	1
7	Clustered Index Seek	9	0	3	0	1
8	Hash Match	10	0	31	0	1
9	Index Scan	11	0	0	0	1
10	Merge Join	13	0	31	0	1
11	Clustered Index Seek	14	0	0	0	1
12	Merge Join	15	0	31	0	1
13	Merge Join	17	0	31	0	1
14	Clustered Index Seek	18	0	0	0	1
15	Merge Join	20	0	31	0	1
16	Clustered Index Seek	21	0	0	0	1
17	Filter	23	0	31	0	1
18	Clustered Index Scan	24	0	31	0	1
19	Clustered Index Seek	25	0	0	0	1
20	Table Spool	27	0	60557	30	1
21	Concatenation	28	0	1956	0	1
22	Merge Join	30	0	1956	0	1
23	Clustered Index Seek	31	0	0	0	1
24	Filter	32	0	1956	0	1
25	Sort	34	0	1956	0	1
26	Clustered Index Scan	35	0	1956	0	1
27	Filter	37	0	0	0	1
28	Clustered Index Scan	39	0	16593	0	1

Figure 7-21. *Operator row counts from actively executing query*

Run the query against the `sys.dm_exec_query_profiles` over and over while the problematic query executes. You'll see that the various row counts continue to increment. With this approach, you can gather metrics on actively executing queries.

There is an easier way to see this in action starting with SQL Server Management Studio 2016. Instead of querying the DMV, you can simply click the Include Live Query Statistics button in the query window containing the problematic query. Then, when you execute the query, the view will change to an execution plan, but it will be actively showing you the row counts as they're moving between operators. Figure 7-22 shows a section of a plan.

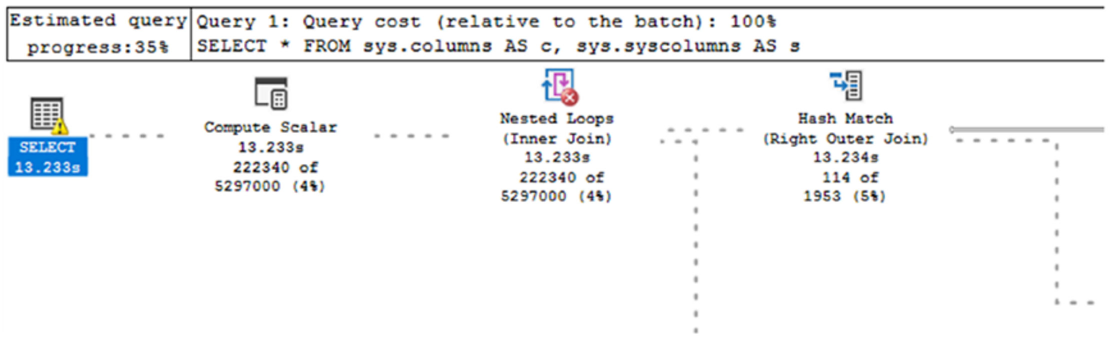

```
Estimated query progress:35%
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM sys.columns AS c, sys.syscolumns AS s
```

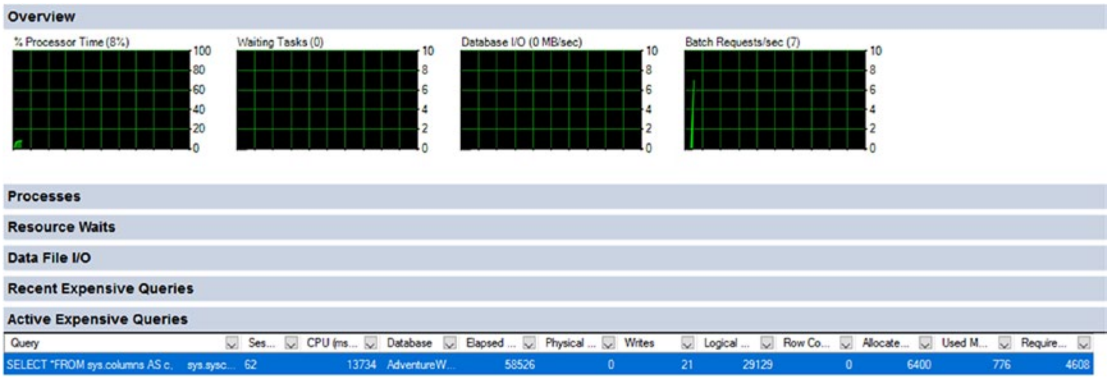
Figure 7-22. *Live execution plan showing rows moving between operators*

Instead of the usual arrows showing the data flow between operators, you get moving dashed lines (obviously, not visible in a book). As operations complete, the dashed lines change to solid lines just as they behave in a regular execution plan.

This is a useful device for understanding what's happening with a long-running query, but the requirement to capture a live execution plan is not convenient if the query is already executing, say on a production server. Further, capturing live execution plans, although useful, is not cost free. So, introduced in SQL Server 2016 SP1 and available in all other versions of SQL Server, a new traceflag was introduced, 7412. Setting that traceflag enables a way to view live query statistics (a live execution plan) on demand. You can also create an Extended Events session and use the `query_thread_profile` event (more on that in the next section). While that is running or the traceflag is enabled, you can get information from `sys.dm_exec_query_profiles` or watch a live execution plan on any query at any time. To see this in action, let's first enable the traceflag on our system.

```
DBCC TRACEON(7412);
```

With it enabled, we'll again run our problematic query. A tool that I don't use often but one that becomes much more attractive with this addition is the Activity Monitor. It's a way to look at activity on your system. You access it by right-clicking the server in the Object Explorer window and selecting Activity Monitor from the context menu. With the traceflag enabled and executing the problematic query, Activity Monitor on my system looks like Figure 7-23.



Query	Ses...	CPU (ms...	Database	Elapsed ...	Physical ...	Writes	Logical ...	Row Co...	Allocate...	Used M...	Require...
SELECT *FROM sys.columns AS c, sys.sysc...	62	13734	AdventureW...	58526	0	21	29129	0	6400	776	4608

Figure 7-23. *Activity Monitor showing Active Expensive Queries*

You'll have to click Active Expensive Queries to see the query running. You can then right-click the query, and you can select Show Live Execution Plan if the query is actively executing.

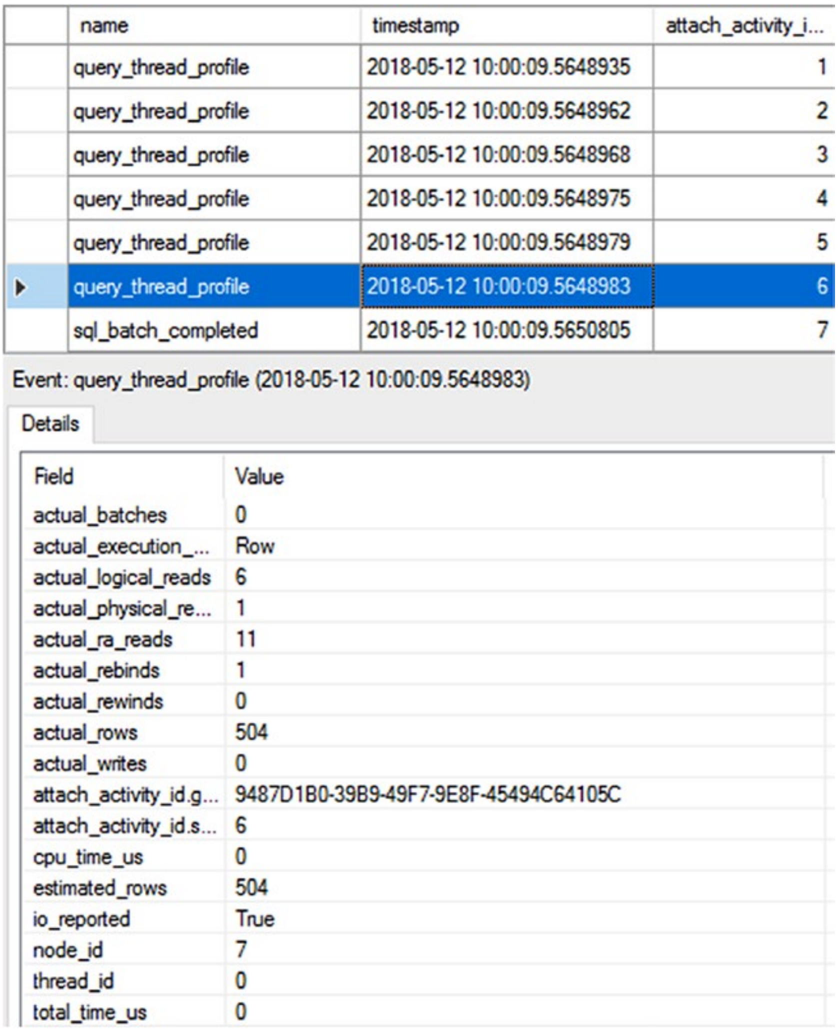
Unfortunately, the naming on all this is somewhat inconsistent. The original DMV refers to query profiles, while the query window in SSMS uses query statistics, the DMV uses thread profiles, and then Activity Monitor talks about live execution plans. They all basically mean the same thing: a way to observe the behavior of operations within an actively executing query. With the new ability to immediately access this information without having to first be actively capturing an execution plan, what was something of an interesting novelty has become an extremely useful tool. You can see precisely which operations are slowing down a long-running query.

Query Thread Profiles

Mentioned earlier, the new Extended Events event query_thread_profile adds new functionality to the system. This event is a debug event. As mentioned in Chapter 6, the debug events should be used sparingly. However, Microsoft does advocate for the use of this event. Running it will allow you to watch live execution plans on long-running queries. However, it does more than that. It also captures row and thread counts for all operators within an execution plan at the end of the execution of that plan. It's very low cost and an easy way to capture those metrics, especially on queries that run fast where you could never really see their active row counts in a live execution plan. This is data that you get with an execution plan, but this is much more low cost than capturing a plan. This is the script for creating a session that captures the query thread profiles as well as the core query metrics:

```
CREATE EVENT SESSION QueryThreadProfile
ON SERVER
    ADD EVENT sqlserver.query_thread_profile
    (WHERE (sqlserver.database_name = N'AdventureWorks2017')),
    ADD EVENT sqlserver.sql_batch_completed
    (WHERE (sqlserver.database_name = N'AdventureWorks2017'))
WITH (TRACK_CAUSALITY = ON)
GO
```

With this session running, if you run a small query, such as the one we used at the start of the "Execution Plan Tooling" section, the output looks like Figure 7-24.



	name	timestamp	attach_activity_i...
	query_thread_profile	2018-05-12 10:00:09.5648935	1
	query_thread_profile	2018-05-12 10:00:09.5648962	2
	query_thread_profile	2018-05-12 10:00:09.5648968	3
	query_thread_profile	2018-05-12 10:00:09.5648975	4
	query_thread_profile	2018-05-12 10:00:09.5648979	5
▶	query_thread_profile	2018-05-12 10:00:09.5648983	6
	sql_batch_completed	2018-05-12 10:00:09.5650805	7

Event: query_thread_profile (2018-05-12 10:00:09.5648983)

Details

Field	Value
actual_batches	0
actual_execution_...	Row
actual_logical_reads	6
actual_physical_re...	1
actual_ra_reads	11
actual_rebinds	1
actual_rewinds	0
actual_rows	504
actual_writes	0
attach_activity_id.g...	9487D1B0-39B9-49F7-9E8F-45494C64105C
attach_activity_id.s...	6
cpu_time_us	0
estimated_rows	504
io_reported	True
node_id	7
thread_id	0
total_time_us	0

Figure 7-24. *Extended Events session showing query_thread_profile information*

You can see the details of the event including estimated rows, actual rows, and a lot of the other information we frequently go to execution plans for as part of evaluating statistics and index use among other things. You can now capture this information on the fly for your queries without having to go through the much costlier process of capturing execution plans. Just remember, this is not a zero-cost operation. It’s just a lower-cost operation. It’s also not going to replace all the uses of an execution plan because the plans show so much more than threads, duration, and row counts.

Query Resource Cost

Even though the execution plan for a query provides a detailed processing strategy and the estimated relative costs of the individual steps involved, if it's an estimated plan, it doesn't provide the actual cost of the query in terms of CPU usage, reads/writes to disk, or query duration. While optimizing a query, you may add an index to reduce the relative cost of a step. This may adversely affect a dependent step in the execution plan, or sometimes it may even modify the execution plan itself. Thus, if you look only at the estimated execution plan, you can't be sure that your query optimization benefits the query as a whole, as opposed to that one step in the execution plan. You can analyze the overall cost of a query in different ways.

You should monitor the overall cost of a query while optimizing it. As explained previously, you can use Extended Events to monitor the `duration`, `cpu`, `reads`, and `writes` information for the query. Extended Events is an extremely efficient mechanism for gathering metrics. You should plan on taking advantage of this fact and use this mechanism to gather your query performance metrics. Just understand that collecting this information leads to large amounts of data that you will have to find a place to maintain within your system.

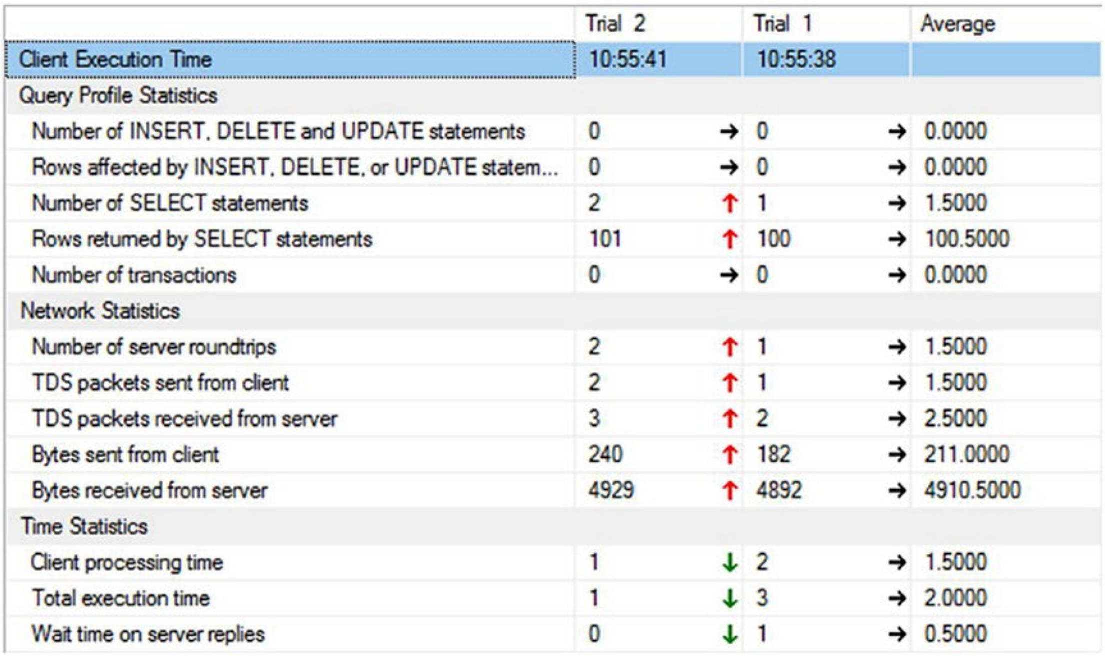
There are other ways to collect performance data that are more immediate and easily accessible than Extended Events. In addition to the ones I detail next, don't forget that we have the DMOs, such as `sys.dm_exec_query_stats` and `sys.dm_exec_procedure_stats`, and the Query Store system views and reports, `sys.query_store_runtime_stats` and `sys.query_store_wait_stats`.

Client Statistics

Client statistics capture execution information from the perspective of your machine as a client of the server. This means that any times recorded include the time it takes to transfer data across the network, not merely the time involved on the SQL Server machine. To use them, simply select Query ➤ Include Client Statistics. Now, each time you run a query, a limited set of data is collected including the execution time, the number of rows affected, the round-trips to the server, and more. Further, each execution of the query is displayed separately on the Client Statistics tab, and a column aggregating the multiple executions shows the averages for the data collected. The statistics will also show whether a time or count has changed from one run to the next, showing up as arrows, as shown in Figure 7-13. For example, consider this query:

```
SELECT TOP 100
    p.Name,
    p.ProductNumber
FROM Production.Product p;
```

The client statistics information for the query should look something like those shown in Figure 7-25.



	Trial 2		Trial 1		Average
Client Execution Time	10:55:41		10:55:38		
Query Profile Statistics					
Number of INSERT, DELETE and UPDATE statements	0	→	0	→	0.0000
Rows affected by INSERT, DELETE, or UPDATE statem...	0	→	0	→	0.0000
Number of SELECT statements	2	↑	1	→	1.5000
Rows returned by SELECT statements	101	↑	100	→	100.5000
Number of transactions	0	→	0	→	0.0000
Network Statistics					
Number of server roundtrips	2	↑	1	→	1.5000
TDS packets sent from client	2	↑	1	→	1.5000
TDS packets received from server	3	↑	2	→	2.5000
Bytes sent from client	240	↑	182	→	211.0000
Bytes received from server	4929	↑	4892	→	4910.5000
Time Statistics					
Client processing time	1	↓	2	→	1.5000
Total execution time	1	↓	3	→	2.0000
Wait time on server replies	0	↓	1	→	0.5000

Figure 7-25. *Client statistics*

Although capturing client statistics can be a useful way to gather data, it's a limited set of data, and there is no way to show how one execution is different from another. You could even run a completely different query, and its data would be mixed in with the others, making the averages useless. If you need to, you can reset the client statistics. Select the Query menu and then the Reset Client Statistics menu item.