(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.

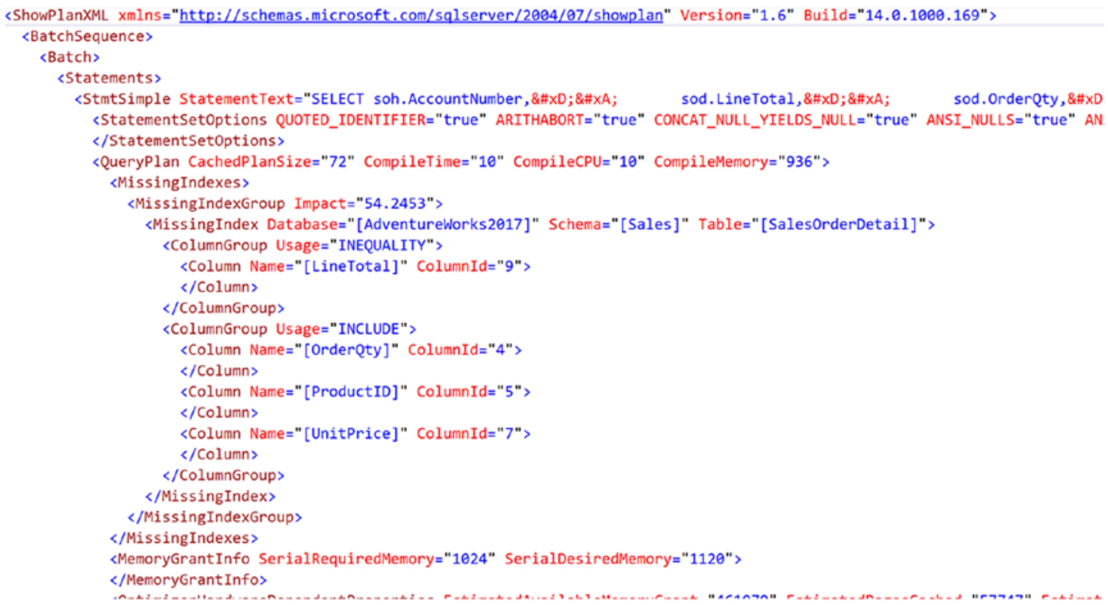
Note You should make sure your database is set to Compatibility Mode 140 so that it accurately reflects updates to SQL Server 2017.

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.

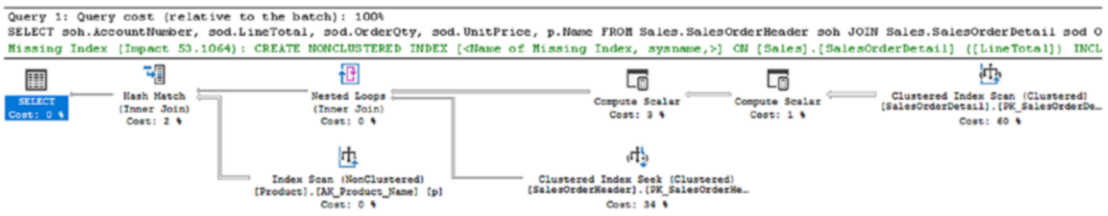


```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.6" Build="14.0.1000.169">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="SELECT soh.AccountNumber,&#xD;&#xA;       sod.LineTotal,&#xD;&#xA;       sod.OrderQty,&#xD
          <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*

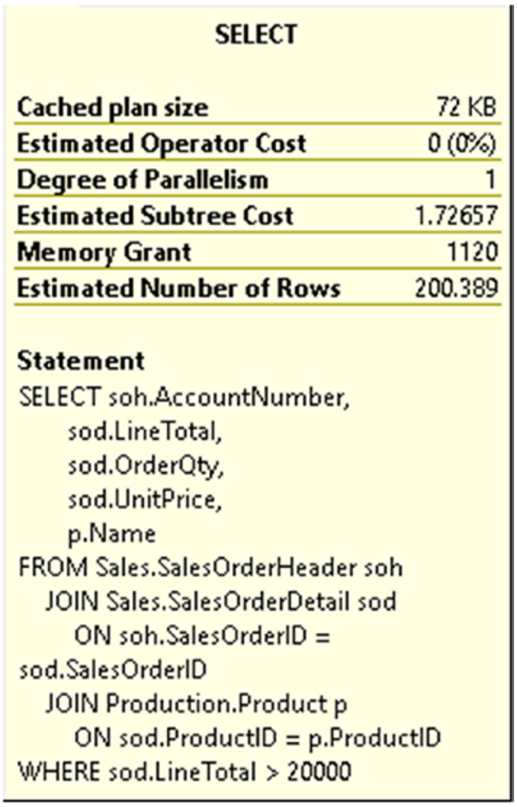
Execution plans show two different flows of information. Reading from the left side, you can see the logical flow, starting with the `SELECT` operator and proceeding through each of the execution steps. Starting from the right side and reading the other way is the physical flow of information, pulling 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

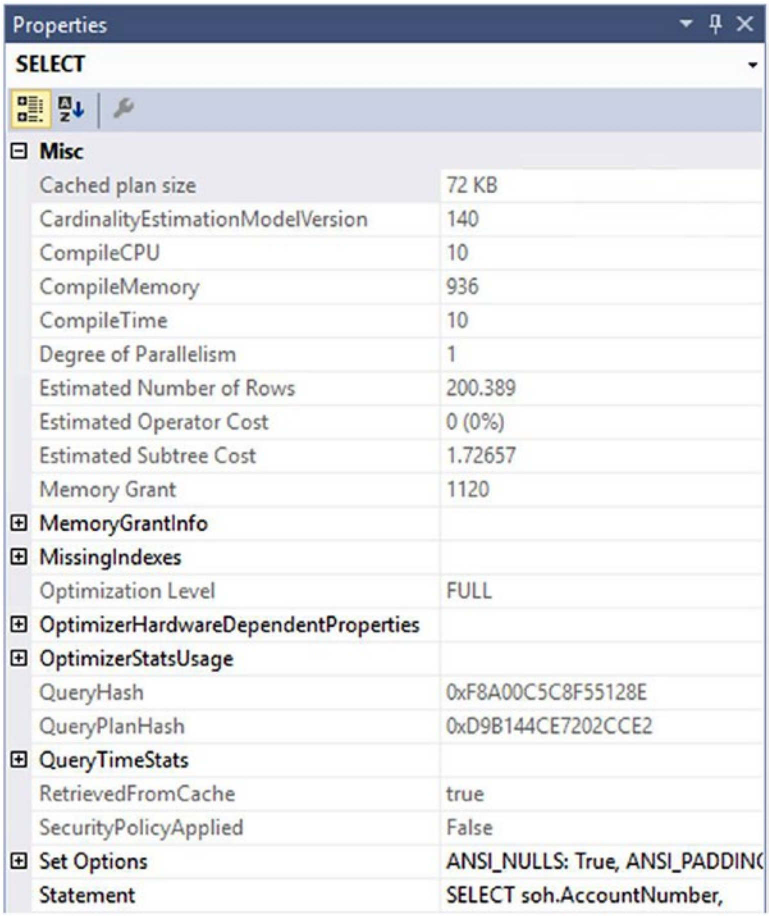
Cached plan size	72 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	1.72657
Memory Grant	1120
Estimated Number of Rows	200.389

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.



Properties

SELECT

⊟ 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.

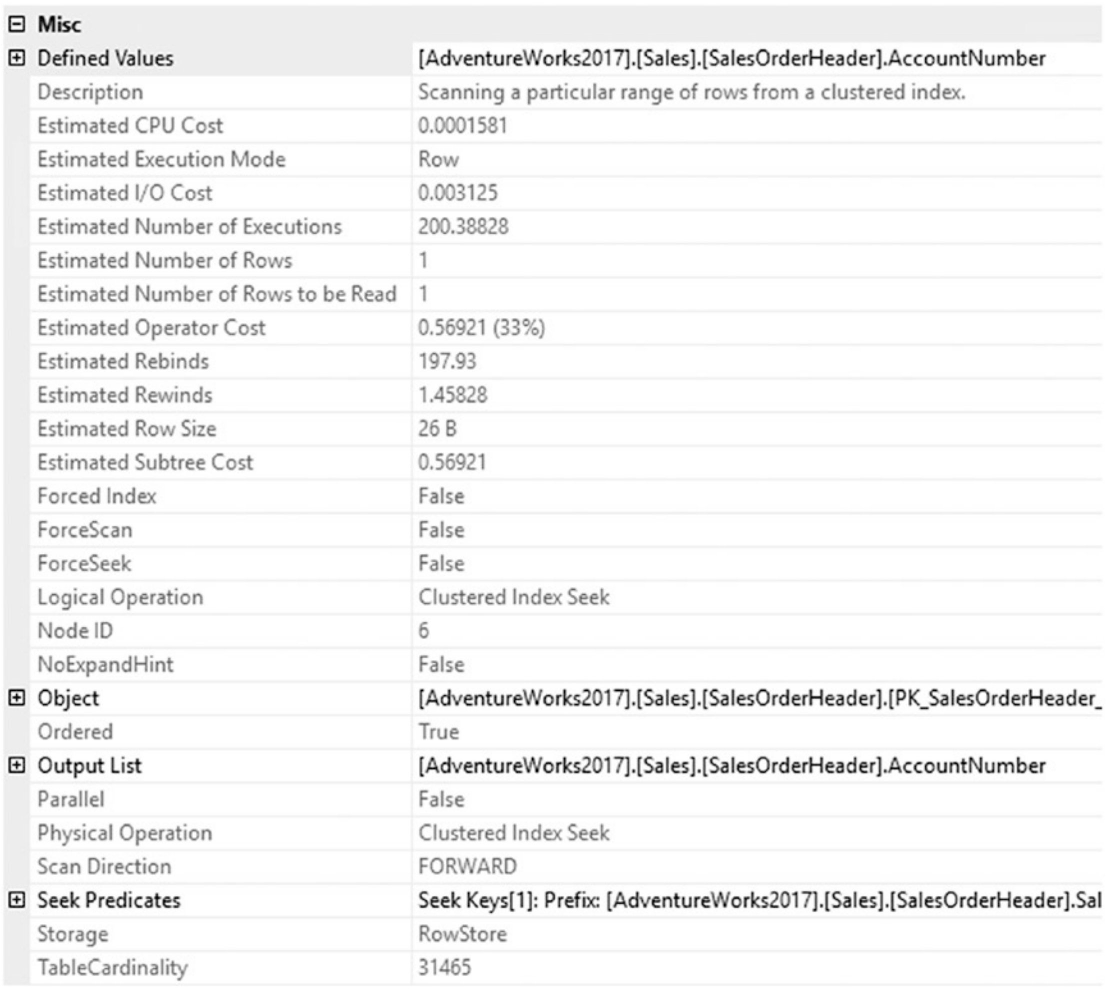
Analyzing Index Effectiveness

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.



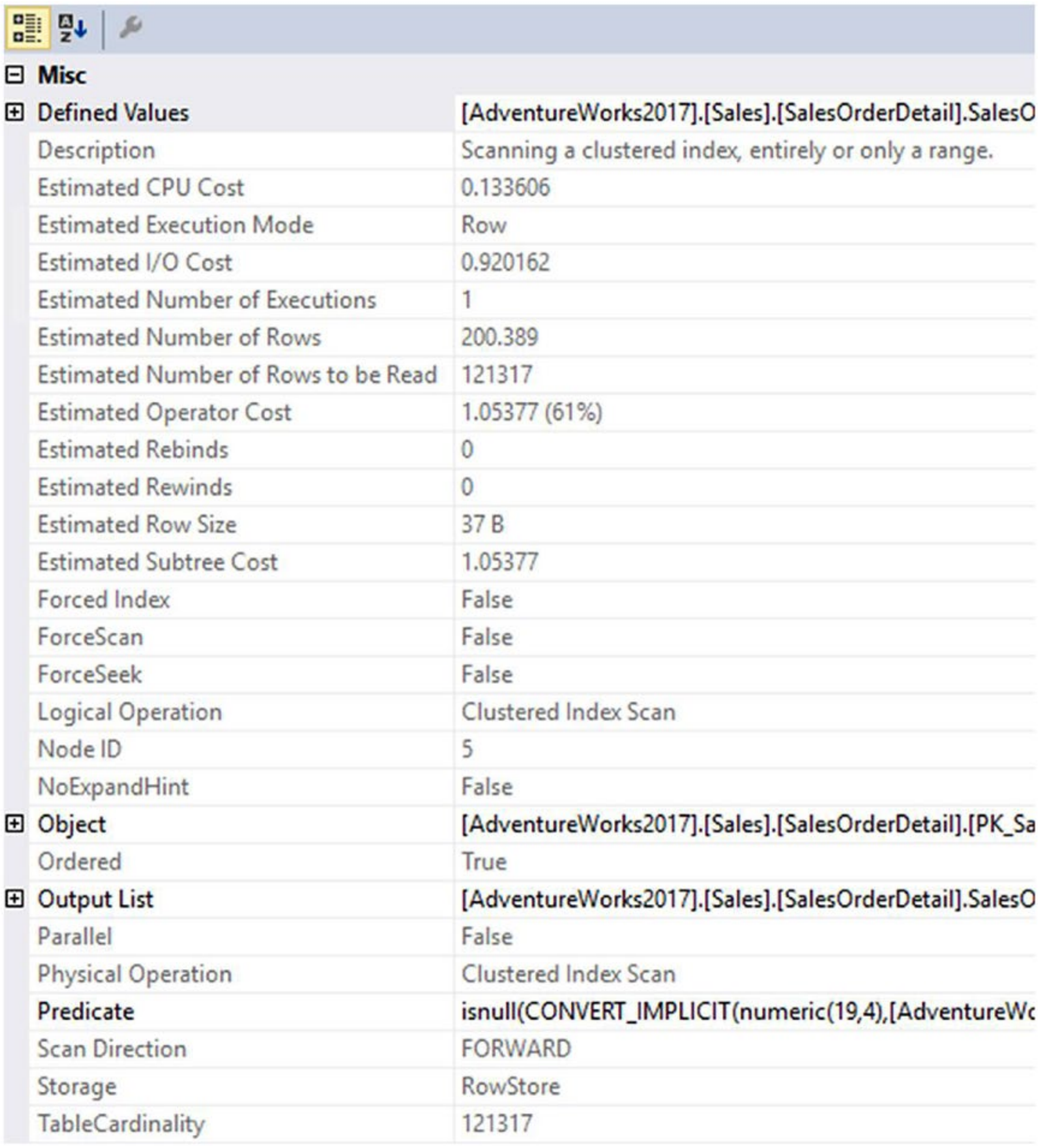
⊟ **Misc**	
⊞ Defined Values	[AdventureWorks2017].[Sales].[SalesOrderHeader].AccountNumber
Description	Scanning a particular range of rows from a clustered index.
Estimated CPU Cost	0.0001581
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	200.38828
Estimated Number 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.