

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.

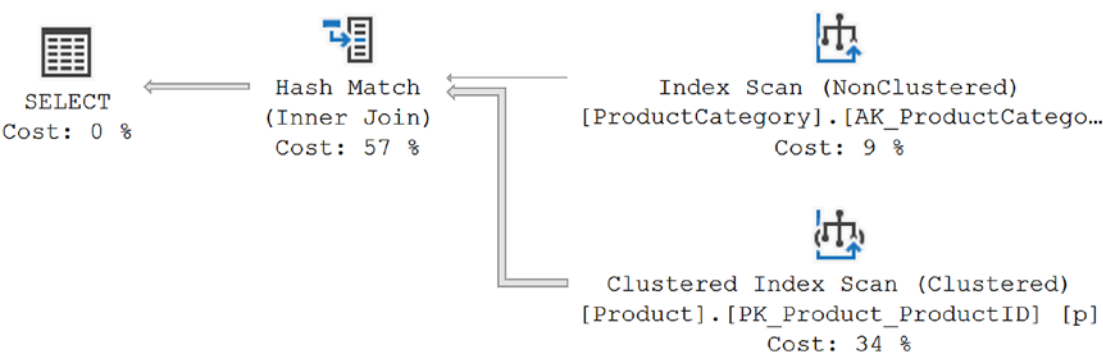


Figure 7-7. *Execution plan with a hash join*

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.

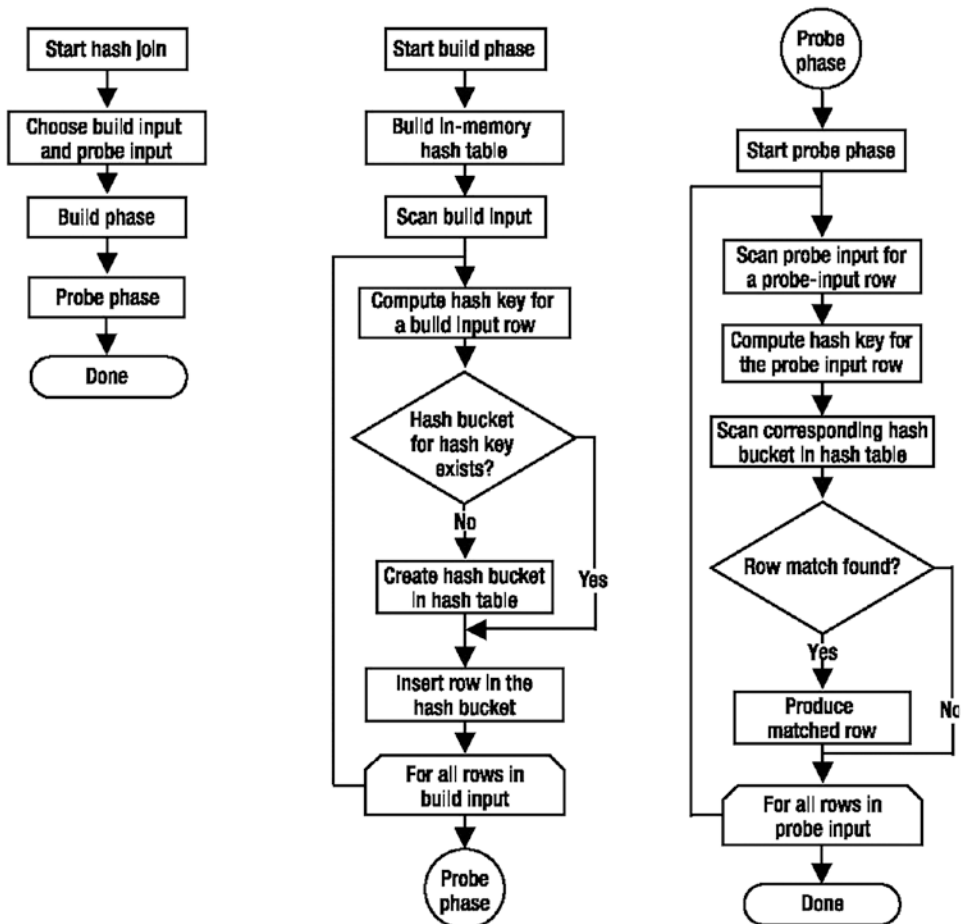


Figure 7-8. Workflow for an in-memory hash join

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.

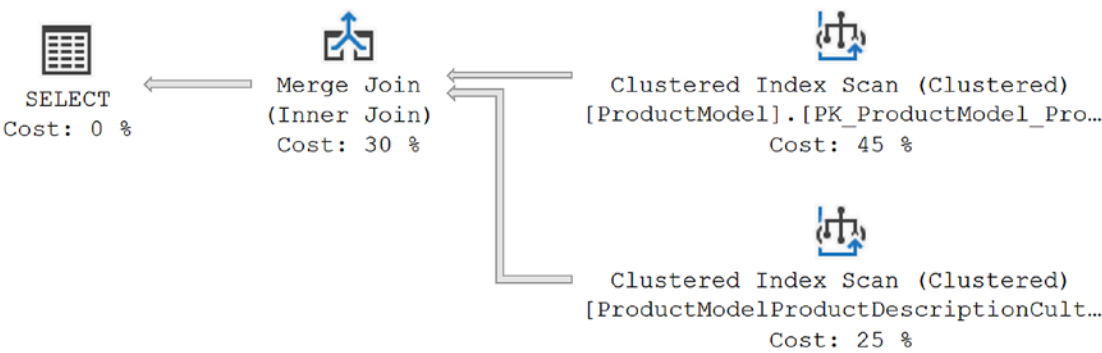


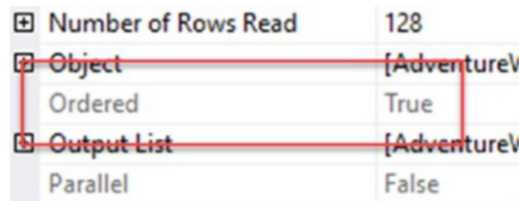
Figure 7-9. Execution plan with a merge join

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	[AdventureW
Ordered	True
Output List	[AdventureW
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.

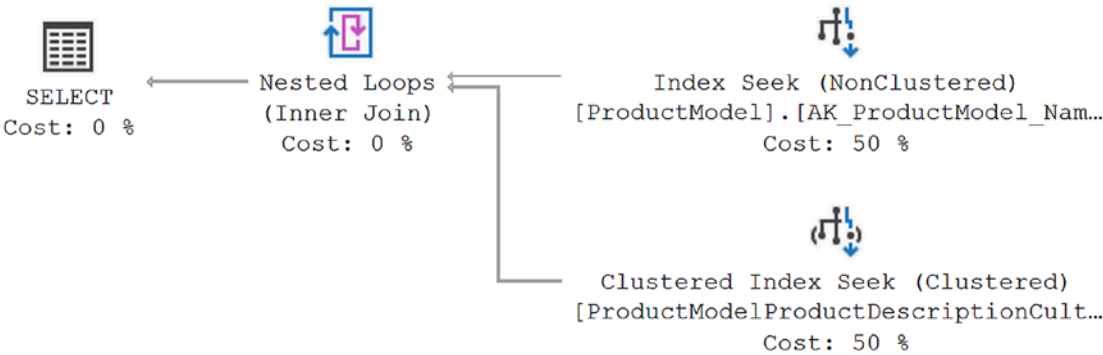


Figure 7-11. Execution plan with a nested loop join

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.

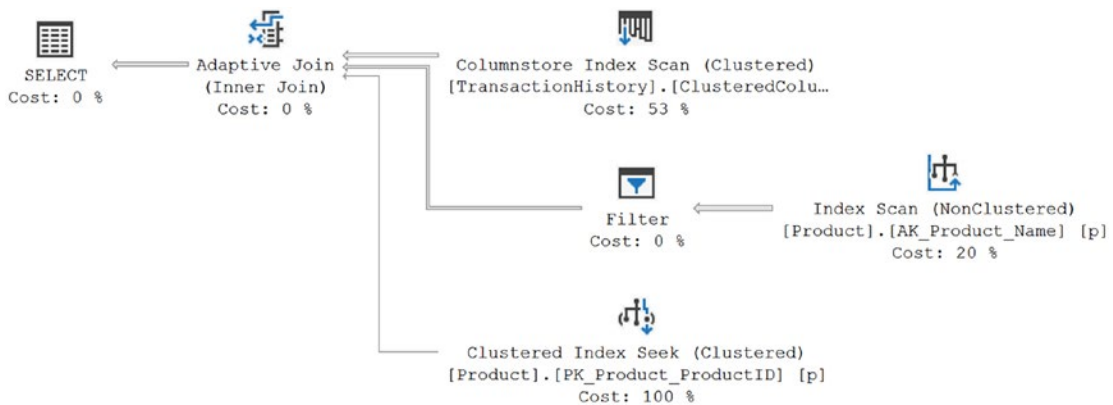


Figure 7-12. Execution plan with an adaptive join

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
<input checked="" type="checkbox"/> Actual I/O Statistics	
Actual Join Type	HashMatch
<input checked="" type="checkbox"/> Actual Number of Batches	2
<input checked="" type="checkbox"/> 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.

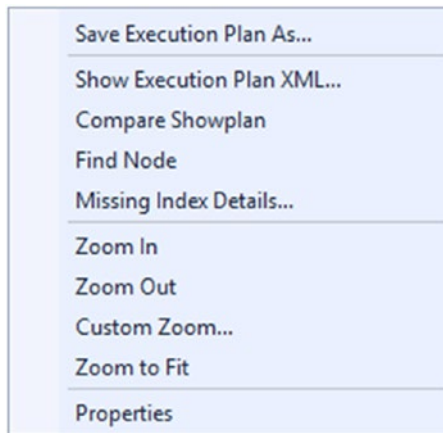


Figure 7-15. Execution plan context menu

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.

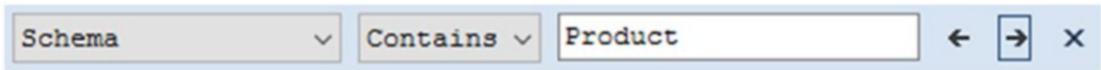


Figure 7-17. Looking for any operators that have a Schema value that contains Product

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.

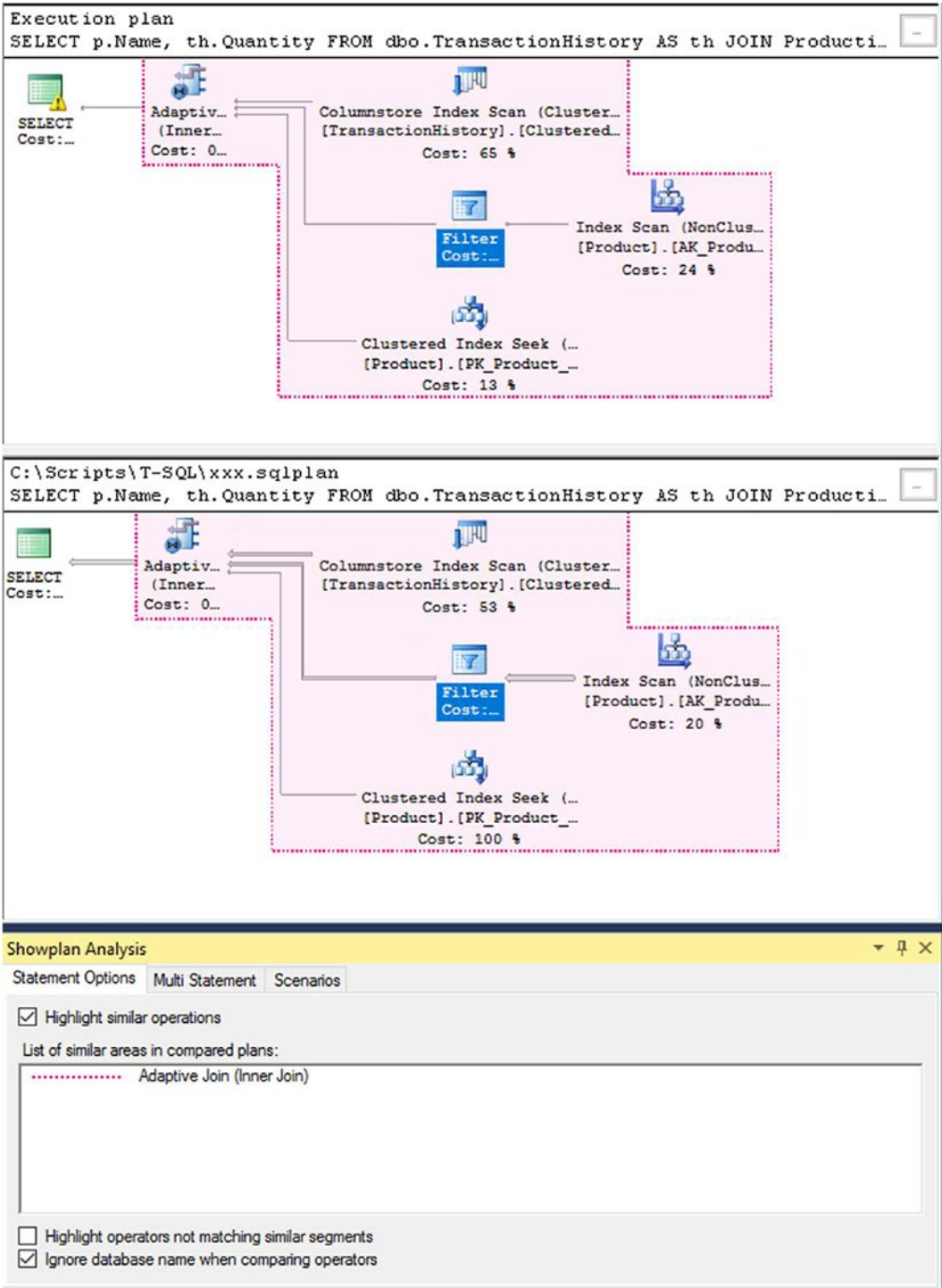


Figure 7-18. Execution plan comparison within SSMS

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.

Properties			
Top Plan		Bottom Plan	
SELECT		SELECT	
> Actual Number of Rows	1	> Actual Number of Rows	447
Cached plan size	56 KB	Cached plan size	56 KB
CardinalityEstimationModelVer	140	CardinalityEstimationModelVer	140
CompileCPU	2	CompileCPU	2
CompileMemory	392	CompileMemory	400
CompileTime	2	CompileTime	2
Degree of Parallelism	1	Degree of Parallelism	1
Estimated Number of Rows	1	Estimated Number of Rows	447
Estimated Operator Cost	0 (0%)	Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0250929	Estimated Subtree Cost	0.0306292
Memory Grant	1056	Memory Grant	1184
> MemoryGrantInfo		> MemoryGrantInfo	
Optimization Level	FULL	Optimization Level	FULL
> OptimizerHardwareDependent		> OptimizerHardwareDepender	
> OptimizerStatsUsage		> OptimizerStatsUsage	
QueryHash	0xBAD67891E8D72D9A	QueryHash	0xBAD67891E8D72D9A
QueryPlanHash	0xE347E782798D6B42	QueryPlanHash	0xE347E782798D6B42
> QueryTimeStats		> QueryTimeStats	
Reason For Early Termination	Time Out	Reason For Early Termination	Time Out
RetrievedFromCache	true	RetrievedFromCache	true
SecurityPolicyApplied	False	SecurityPolicyApplied	False
> Set Options	ANSI_NULLS: True, ANSI_PADDING	> Set Options	ANSI_NULLS: True, ANSI_PADDING
Statement	SELECT p.Name, th.Quantity	Statement	SELECT p.Name, th.Quantity
> Warnings	The query memory grant detecte	> 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.

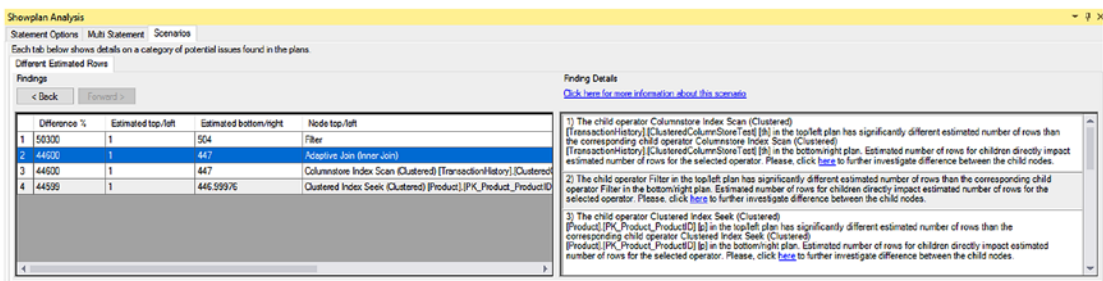


Figure 7-20. *Different Estimated Rows scenario in Showplan Analysis window*

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.

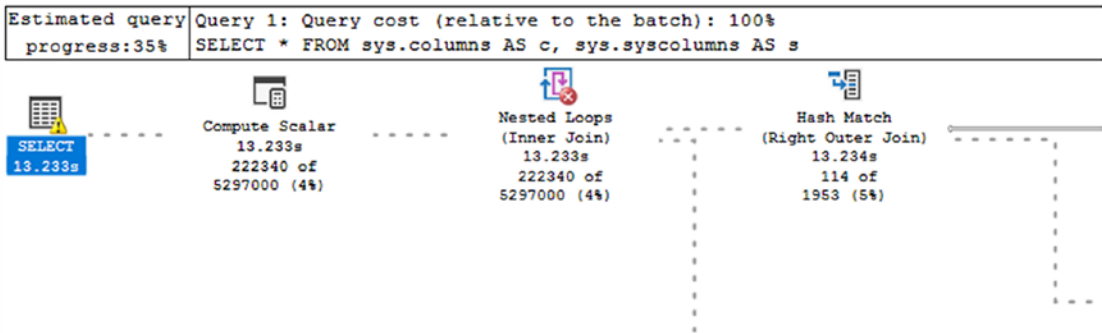


Figure 7-22. Live execution plan showing rows moving between operators

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.

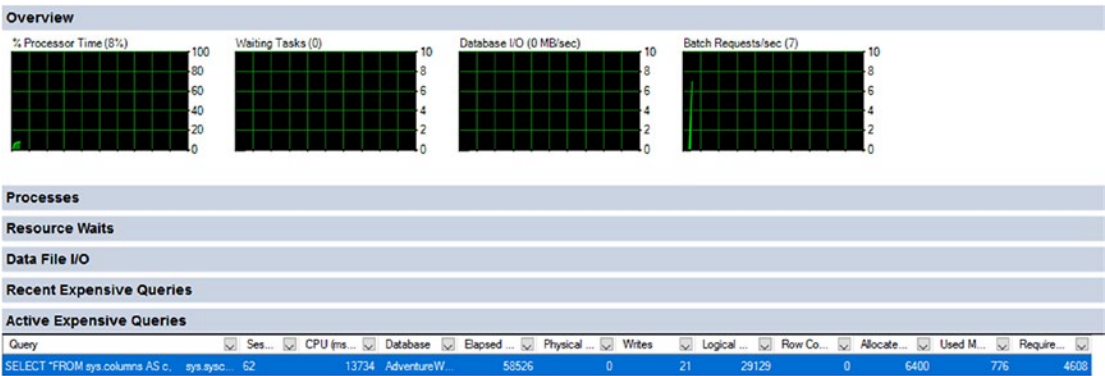


Figure 7-23. Activity Monitor showing Active Expensive Queries

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.