Adaptive Query Processing in SQL Server 2017

[Robert Sheldon](https://www.red-gate.com/simple-talk/author/robert-sheldon/" \o "Posts by Robert Sheldon)

01 October 2018

FROM: <https://www.red-gate.com/simple-talk/sql/database-administration/adaptive-query-processing-in-sql-server-2017/>

As new versions of SQL Server have been released, the Query Optimizer has undergone many improvements. Adaptive Query Processing, new in 2017, represents a new direction. This set of features provides some flexibility for varying join types, multi-statement table valued functions, and memory grants. In this article, Robert Sheldon explains the three features that make up Adaptive Query Processing

SQL Server 2017 now offers adaptive query processing, a new set of features aimed at improving query performance. Adaptive query processing addresses issues related to cardinality estimates in execution plans, leading to better memory allocation, join type selection, and row calculations for multi-statement table valued functions (MSTVFs).

Prior to SQL Server 2017, if a query plan contained incorrect cardinality estimates, the database engine continued to use that plan for each statement execution, as long as the plan remained cached, often resulting in less-than-optimal query performance. For example, the execution plan might allocate too much memory for some queries, while underestimating the memory requirements for others.

The adaptive query processing features attempt to resolve these types of issues by providing more accurate cardinality estimates when calculating query execution plans. SQL Server 2017 enables these features by default on databases configured with a compatibility level of 140 or greater.

If a database has a lower compatibility level, you can use an ALTER DATABASE statement to change the level. For example, the following statement changes the compatibility level of the WideWorldImporters sample database to 140:

|  |  |
| --- | --- |
| 1 | ALTER DATABASE WideWorldImporters SET COMPATIBILITY\_LEVEL = 140; |

The WideWorldImporters database is used for all the examples in this article. If you have this database installed on your system, you should be able to try out the examples without making any changes. If you want to use a different database, you can create SELECT statements comparable to the ones shown in the examples. The same principles should apply to any database with a compatibility level of 140 running on SQL Server 2017.

You can verify a database’s compatibility level by running the following SELECT statement, passing in the name of the database in the WHERE clause:

|  |  |
| --- | --- |
| 1  2 | SELECT compatibility\_level FROM sys.databases  WHERE name = 'WideWorldImporters'; |

If you run this SELECT statement after executing the preceding ALTER DATABASE statement, the SELECT statement should return a value of 140.

Setting the compatibility level on a database is the only step you need to take in SQL Server 2017 to enable the adaptive query processing features for that database. Currently, SQL Server 2017 supports three adaptive query processing types:

Batch mode memory grant feedback

Batch mode adaptive join

Interleaved execution

As already noted, these features are enabled by default. However, you can disable or enable each one individually, without changing the database’s compatibility level. The following sections cover the three features in more detail, including the steps necessary to disable or enable them.

Memory Grant Feedback

SQL Server uses memory to store row data during join and sort operations. When compiling an execution plan, the query engine estimates how much memory is needed to store those rows. If the memory estimate is too small, excess data will spill over to the disk, impacting performance. If the estimate is too large, memory is wasted, impacting the performance of concurrent operations.

The memory grant feedback feature helps remedy this situation by recalculating the row memory requirements when the statement is first executed. If the initial estimate is off, the cached plan is updated. Subsequent executions can then benefit from the new estimate, as long as the query plan remains in cache.

The best way to understand how memory grant feedback works is to see it in action, starting with how SQL Server has traditionally behaved when estimating memory requirements. To demonstrate this behavior, first disable the memory grant feedback feature by running the following ALTER DATABASE statement:

|  |  |
| --- | --- |
| 1 | ALTER DATABASE SCOPED CONFIGURATION SET DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK = ON; |

The statement sets the DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK configuration option to ON, which disables the memory grant feedback features without impacting the database’s compatibility level. To verify that the setting has been updated, run the following SELECT statement:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM sys.database\_scoped\_configurations; |

The SELECT statement returns data about the database’s scoped configuration settings, as shown in Figure 1.

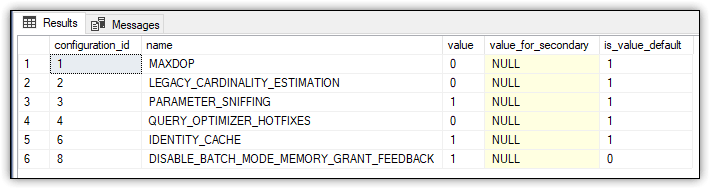


Figure 1. Disabling memory grant feedback

Row 6 of the results includes the DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK option. Notice that the option’s current value is 1 (ON) and that the default value is 0 (OFF), indicating that memory grant feedback is enabled by default (but only for databases with a compatibility level of 140 or greater).

Next, run the following SELECT statement with the Actual Execution Plan enabled:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT il.StockItemID, il.Quantity, il.ExtendedPrice, iv.InvoiceDate  FROM Sales.InvoiceLines il INNER JOIN Sales.Invoices iv    ON il.InvoiceID = iv.InvoiceID  WHERE iv.InvoiceDate BETWEEN '2013-01-01' AND '2015-12-31'  ORDER BY il.StockItemID, il.Quantity DESC; |

After the statement runs, go to the execution plan and hover over the Select operator to display the operator’s details, which are shown in Figure 2.

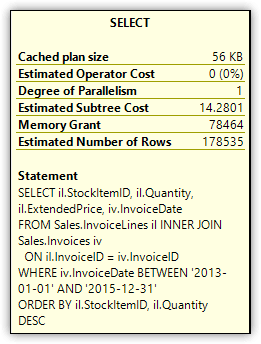


Figure 2. Memory Grant attribute of the Select operator

The Memory Grant attribute indicates that 78,464 KB of memory is required for the query’s row data. No matter how many times you rerun the SELECT statement, you should receive the same Memory Grant total, as long as the query plan remains cached. Even if you’re receiving a different total than the one shown here, the behaviour should be the same.

With this in mind, you can now test the memory grant feedback feature by re-enabling the feature and then re-executing the SELECT statement. To re-enable the feature, run the following ALTER DATABASE statement, which sets the DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK option to OFF:

|  |  |
| --- | --- |
| 1 | ALTER DATABASE SCOPED CONFIGURATION SET DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK = OFF; |

When you turn the DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK option to OFF, the option is no longer listed in the sys.database\_scoped\_configurations table. Only when you set the option to ON is it included in the table. This is true for all the scoped configuration options specific to enabling or disabling adaptive query processing features.

After you re-enable the memory grant feedback feature, you should rerun the example SELECT statement. Before you do that, however, clear the execution plan from cache. (If appropriate, you should clear the cache between each example to ensure you see the correct behaviour when you test these statements. But don’t do this on a production server. In fact, you should never be testing new features on a production server.) One approach to clearing the cache is to run the following T-SQL statement:

|  |  |
| --- | --- |
| 1 | DBCC FREEPROCCACHE; |

SQL Server provides several methods for clearing the cache, so pick whichever one works for you. The DBCC FREEPROCCACHE statement is a fairly straightforward approach, as long as it’s okay for all query plans to be cleared from the cache. If it’s not, you’ll have to specify the specific plan you want to remove.

After you re-enable the memory grant feedback feature and clear the cache, run the following SELECT statement two or more times (which is the same SELECT statement as above):

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT il.StockItemID, il.Quantity, il.ExtendedPrice, iv.InvoiceDate  FROM Sales.InvoiceLines il INNER JOIN Sales.Invoices iv    ON il.InvoiceID = iv.InvoiceID  WHERE iv.InvoiceDate BETWEEN '2013-01-01' AND '2015-12-31'  ORDER BY il.StockItemID, il.Quantity DESC; |

The first time you rerun this statement, you should receive the same results as before, with the Memory Grant attribute showing a total of 78,464 KB of memory, or something close to that. However, when you then rerun the statement, the total should be much lower. On my system, the subsequent executions resulted in a Memory Grant total of 14,592 KB, as shown in Figure 3.

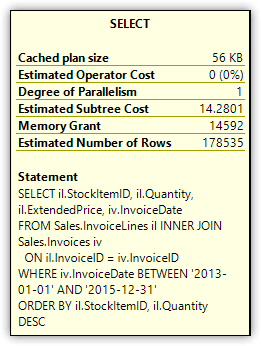


Figure 3. Memory Grant attribute of the Select operator

When I tested the memory grant feedback feature on my system, I reran the above SELECT statement numerous times. Although I generally received the same Memory Grant total described here, in some cases I would get the original estimate or even another value. For the most part, however, the feature worked as advertised, despite the relatively few inconsistencies.

You can also disable the memory grant feedback feature on a per-statement basis by including an OPTION clause that specifies the hint DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK, as shown in the following SELECT statement:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT il.StockItemID, il.Quantity, il.ExtendedPrice, iv.InvoiceDate  FROM Sales.InvoiceLines il INNER JOIN Sales.Invoices iv    ON il.InvoiceID = iv.InvoiceID  WHERE iv.InvoiceDate BETWEEN '2013-01-01' AND '2015-12-31'  ORDER BY il.StockItemID, il.Quantity DESC  OPTION (USE HINT ('DISABLE\_BATCH\_MODE\_MEMORY\_GRANT\_FEEDBACK')); |

The Memory Grant attribute should once again show a total of 78,464 KB of memory (or something similar), no matter how often you rerun the statement, at least until the plan is recached.

Adaptive Joins

When a SELECT statement includes a join condition, the query engine attempts to determine the best join type to use based on the estimated number of rows. Prior to SQL Server 2017, if an execution plan chose a bad join type, there was little that could be done, outside of specifying a query hint or specific join type.

The new adaptive join feature helps to remedy this situation by choosing a different join type during statement execution, if necessary. After the first input has been scanned, the execution plan determines whether to change the join type to a hash join or nested loop join based on a calculated threshold.

You can see how this feature works by comparing the old method to the new, similar to the approach taken when testing the memory grant feedback feature. To disable the adaptive join feature, run the following ALTER DATABASE statement, setting the DISABLE\_BATCH\_MODE\_ADAPTIVE\_JOINS option to ON:

|  |  |
| --- | --- |
| 1 | ALTER DATABASE SCOPED CONFIGURATION SET DISABLE\_BATCH\_MODE\_ADAPTIVE\_JOINS = ON; |

Not surprisingly, the DISABLE\_BATCH\_MODE\_ADAPTIVE\_JOINS option is specific to adaptive joins. However, the adaptive query processing options all work the same. To disable a feature, set its related option to ON, and to enable the feature, set the option to OFF.

To verify that the option has been set to ON and the feature disabled, you can run the following SELECT statement:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM sys.database\_scoped\_configurations |

The SELECT statement returns the results shown in Figure 3, which indicate that the DISABLE\_BATCH\_MODE\_ADAPTIVE\_JOINS option has been set to 1 (ON) and that the default is 0 (OFF).

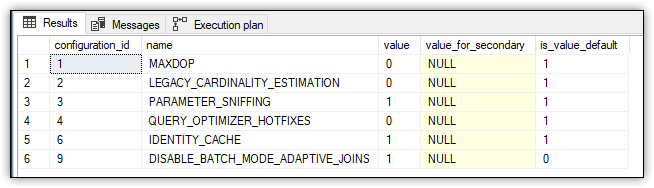


Figure 4. Disabling adaptive joins

Next, run the following SELECT statement, only this time enable Live Query Statistics:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT iv.InvoiceID, il.InvoiceLineID, il.StockItemID, il.Quantity  FROM Sales.Invoices iv INNER JOIN Sales.InvoiceLines il    ON iv.InvoiceID = il.InvoiceID  WHERE il.Quantity > 100; |

The execution plan should look similar to the one shown in Figure 5, which shows a columnstore index scan, a nonclustered index scan, and a hash join.

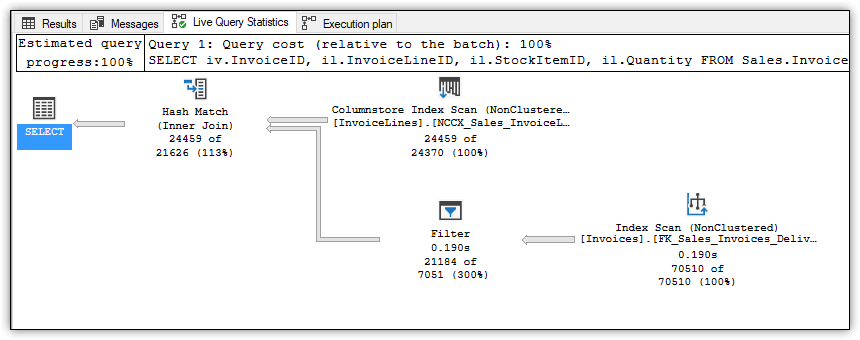


Figure 5. Executing a non-adaptive join

Because Live Query Statistics were enabled, the plan also shows the counts for the number of rows compared to the number of estimated rows, all of which should look fairly straightforward. In fact, I’ve included this example only to compare it with the query plan when adaptive joins are enabled.

The next step, then, is to enable adaptive joins by running the following ALTER DATABASE statement:

|  |  |
| --- | --- |
| 1 | ALTER DATABASE SCOPED CONFIGURATION SET DISABLE\_BATCH\_MODE\_ADAPTIVE\_JOINS = OFF; |

After running this statement, rerun the preceding SELECT statement (again shown here for your convenience):

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT iv.InvoiceID, il.InvoiceLineID, il.StockItemID, il.Quantity  FROM Sales.Invoices iv INNER JOIN Sales.InvoiceLines il    ON iv.InvoiceID = il.InvoiceID  WHERE il.Quantity > 100; |

Now take a look at the execution plan. You’ll find a couple additions, including a Clustered Index Seek operator and, more importantly, the new Adaptive Join operator, as shown in Figure 6.

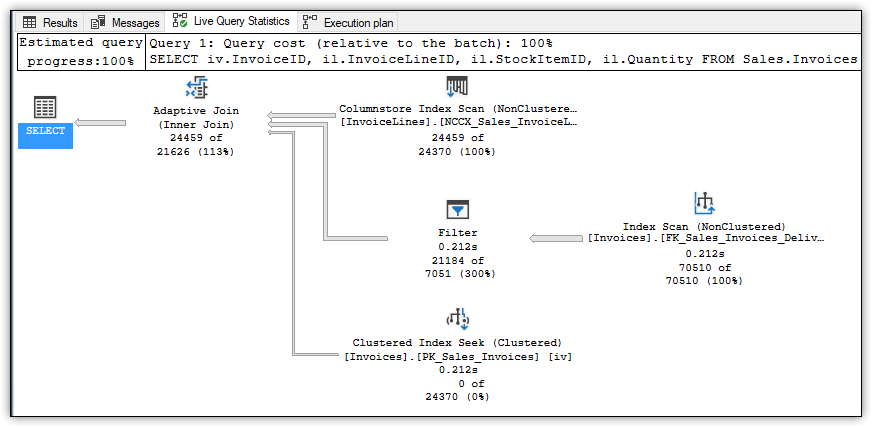


Figure 6. Executing an adaptive join

The Clustered Index Seek operator is included for use by a nested loop join if needed. Notice that 0 of 24370 is specified, indicating that this branch is unused, which implies that a hash join was selected for this operation.

The Adaptive Join operator determines what join type is used by calculating a threshold that determines whether to perform a hash join or a nested loop join, based on the row count. In this case, that threshold is 159.754, and the row count is 24,459. If the row count is greater than or equal to the threshold, the query plan uses a hash join. Otherwise, the plan uses a nested loop join.

If you hover over the Adaptive Join operator to display the details, you’ll see that they include three important attributes:

Estimated Join Type, which is set to HashMatch

Adaptive Threshold Rows, which is set to 159.754

Is Adaptive, which is set to True

Figure 7 shows the details for the Adaptive Join operator after running the SELECT statement with the adaptive join feature enabled.

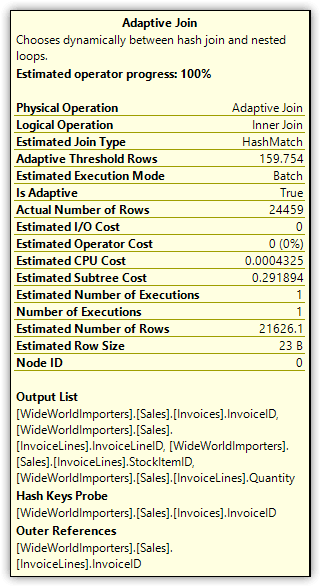


Figure 7. Attributes of the Adaptive Join operator

Suppose you now run the following UPDATE statement against the InvoiceLines table:

|  |  |
| --- | --- |
| 1  2 | UPDATE Sales.InvoiceLines SET Quantity = 361  WHERE InvoiceLineID = 41606; |

Next, run the previous SELECT statement again, only this time specify a Quantity value of 360 in the WHERE clause:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT iv.InvoiceID, il.InvoiceLineID, il.StockItemID, il.Quantity  FROM Sales.Invoices iv INNER JOIN Sales.InvoiceLines il    ON iv.InvoiceID = il.InvoiceID  WHERE il.Quantity > 360; |

This time, the details for the Adaptive Join operator will show the join type as NestedLoops and the threshold as 104.24.

If you want to return the WorldWideImporters database back to its original state, run the following UPDATE statement:

|  |  |
| --- | --- |
| 1  2 | UPDATE Sales.InvoiceLines SET Quantity = 360  WHERE InvoiceLineID = 41606; |

Be aware that the adaptive join feature comes with extra memory overhead and that it currently supports only SELECT statements (no data modification statements). In addition, the statement must be eligible for both hash joins and nested loop joins to use the adaptive join feature.

Similar to the memory grant feedback feature, you can disable adaptive joins on a per-statement basis by including an OPTION clause and specifying the DISABLE\_BATCH\_MODE\_ ADAPTIVE\_JOINS hint, as shown in the following SELECT statement:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT iv.InvoiceID, il.InvoiceLineID, il.StockItemID, il.Quantity  FROM Sales.Invoices iv INNER JOIN Sales.InvoiceLines il    ON iv.InvoiceID = il.InvoiceID  WHERE il.Quantity > 100  OPTION (USE HINT('DISABLE\_BATCH\_MODE\_ADAPTIVE\_JOINS')); |

When you include the OPTION clause, the SELECT statement will run just like it would in a database with a compatibility level earlier than 140, but without affecting the current compatibility level.

Interleaved Execution

Prior to SQL Server 2017, when a statement included an MSTVF, the execution plan fixed the row estimate at 100, no matter how many rows the function might actually return. For small datasets, this usually wasn’t a problem, but when there was a wide difference between the estimate and the actual count, performance could suffer.

The interleaved execution feature helps address this issue by pausing execution long enough to capture a more accurate cardinality and then using that information for downstream operations. It should be noted, however, that using MSTVFs can still cause performance issues if they contain complex logic and will be joined against a large number of rows.

To see how this feature works, start by running the following CREATE FUNCTION statement, which defines a very simple MSTVF:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | CREATE FUNCTION dbo.GetInvoiceLines (@qty INT)  RETURNS @tbl TABLE(LineID INT, InvoiceID INT, Quantity INT, Total DECIMAL)  WITH SCHEMABINDING  AS  BEGIN    INSERT @tbl    SELECT InvoiceLineID, InvoiceID, Quantity, ExtendedPrice    FROM Sales.InvoiceLines    WHERE Quantity > @qty    RETURN  END;  GO |

Next, disable the interleaved execution feature by running the following ALTER DATABASE statement, setting the DISABLE\_INTERLEAVED\_EXECUTION\_TVF option to ON:

|  |  |
| --- | --- |
| 1 | ALTER DATABASE SCOPED CONFIGURATION SET DISABLE\_INTERLEAVED\_EXECUTION\_TVF = ON; |

This is just like you saw in the earlier examples, except that it’s specific to interleaved executions. Also like before, to verify that the option has been set to ON and the feature disabled, you can run the following SELECT statement:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM sys.database\_scoped\_configurations; |

The SELECT statement returns the results shown in Figure 8, which indicate that the DISABLE\_INTERLEAVED\_EXECUTION\_TVF option has been set to 1 (ON) and that the default is 0 (OFF).

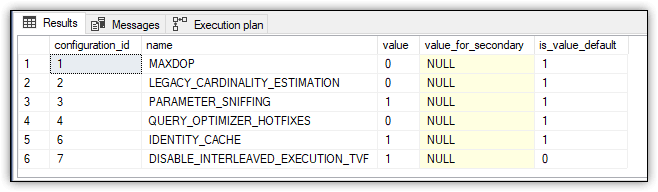


Figure 8. Disabling interleaved execution

Next, run the following SELECT statement with the Actual Execution Plan enabled:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT il.LineID, il.Quantity, il.Total, iv.InvoiceDate  FROM dbo.GetInvoiceLines(100) il INNER JOIN Sales.Invoices iv    ON il.InvoiceID = iv.InvoiceID  WHERE il.Total > 1000; |

The SELECT statement joins the GetInvoiceLines function to the Sales.Invoices table, passing in 100 as the function’s parameter value. Next, go to the execution plan and hover over the Table Valued Function operator. The operator details should show a value of 100 for the Estimated Number of Rows attribute, as shown in Figure 9.

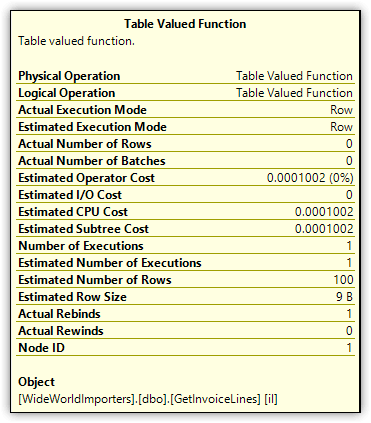


Figure 9. Estimated Number of Rows attribute of the Table Valued Function operator

Although the Table Valued Function operator estimates 100 rows, the function actually returns 24,459 rows, a substantial difference between the two amounts. You can see this amount by viewing the details for the Table Scan operator (the Number of Rows attribute) or by querying the function directly.

To see how interleaved execution changes this behaviour, first re-enable the feature by setting the DISABLE\_INTERLEAVED\_EXECUTION\_TVF option to OFF:

|  |  |
| --- | --- |
| 1 | ALTER DATABASE SCOPED CONFIGURATION SET DISABLE\_INTERLEAVED\_EXECUTION\_TVF = OFF; |

Next, rerun the SELECT statement from above, passing in the same parameter value (100) when calling the function:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT il.LineID, il.Quantity, il.Total, iv.InvoiceDate  FROM dbo.GetInvoiceLines(100) il INNER JOIN Sales.Invoices iv    ON il.InvoiceID = iv.InvoiceID  WHERE il.Total > 1000; |

Finally, go to the execution plan and hover over the Table Valued Function operator. The operator details should now show a value of 24459 for the Estimated Number of Rows attribute, as shown in Figure 10.

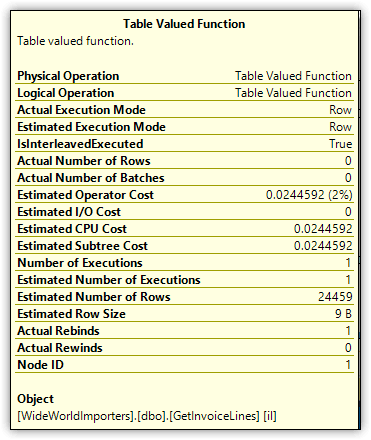


Figure 10. Estimated Number of Rows attribute of the Table Valued Function operator

Being able to return a more accurate row estimate for MSTVFs can help boost query performance, especially when the function returns a large number of rows. In some cases, however, you might want to disable this feature on a per-statement basis, as you saw with the other adaptive query processing features:

|  |  |
| --- | --- |
| 1  2  3  4  5 | SELECT il.LineID, il.Quantity, il.Total, iv.InvoiceDate  FROM dbo.GetInvoiceLines(100) il INNER JOIN Sales.Invoices iv    ON il.InvoiceID = iv.InvoiceID  WHERE il.Total > 1000  OPTION (USE HINT('DISABLE\_INTERLEAVED\_EXECUTION\_TVF')); |

When you include this OPTION clause and specify the hint DISABLE\_INTERLEAVED\_EXECUTION\_TVF, the Table Valued Function operator will once again show an estimate of 100 rows.

Adaptive Query Processing

Depending on the type of queries you’re running, the adaptive query processing capabilities can deliver a noticeable boost in query performance, especially as the size of your workloads grow. It’s unclear at this point whether Microsoft will be enhancing these features anytime soon, but it seems likely we’ll see some improvements. For example, Microsoft might eventually extend the adaptive join capabilities to data modification statements or extend the interleaved execution capabilities beyond MSTVFs. In fact, Microsoft has already released the public preview of the new Table Variable Deferred Compilation feature in Azure SQL Database and SQL Server 2019.

If you’re moving to SQL Server 2017, you should consider updating the compatibility levels of those databases that might benefit from adaptive query processing. Just be sure to fully test the databases to make sure you haven’t introduced any new issues. If you’re uncertain whether your organization will be moving to SQL Server 2017, you might try out the adaptive query processing features when testing other new features to help you determine whether an upgrade is worthwhile.