

```

while(1 -ne 0)
{
    foreach($row in $DatDataSet.Tables[0])
    {
        $name = $row[0]
        $email = $row[1]
        $SqlConnection.Open()
        $Proccmd.Parameters["@FirstName"].Value = $name
        $Proccmd.ExecuteNonQuery() | Out-Null
        $EmailCmd.Parameters["@EmailAddress"].Value = $email
        $EmailCmd.ExecuteNonQuery() | Out-Null
        $SalesCmd.Parameters["@FirstName"].Value = $name
        $SalesCmd.ExecuteNonQuery() | Out-Null
        $OddCmd.Parameters["@FirstName"].Value = $name
        $OddCmd.ExecuteNonQuery() | Out-Null
        $SqlConnection.Close()
    }
}

```

These scripts will enable us to generate the necessary load to cause the automatic index management to fire. The PowerShell script must be run for approximately 12 to 18 hours before a sufficient amount of data can be collected within Azure. However, there are some requirements and settings you must change first.

For automatic index management to work, you must have the Query Store enabled on the Azure SQL Database. The Query Store is enabled by default in Azure, so you'll only need to turn it back on if you have turned it off. To ensure that it is enabled, you can run the following script:

```
ALTER DATABASE CURRENT SET QUERY_STORE = ON;
```

With the Query Store enabled, you'll now need to navigate to the Overview screen of your database. Figure 25-2 shows the full screen. For a reminder, at the bottom of the screen are a number of options, one of which is "Automatic tuning," as shown in Figure 25-10.

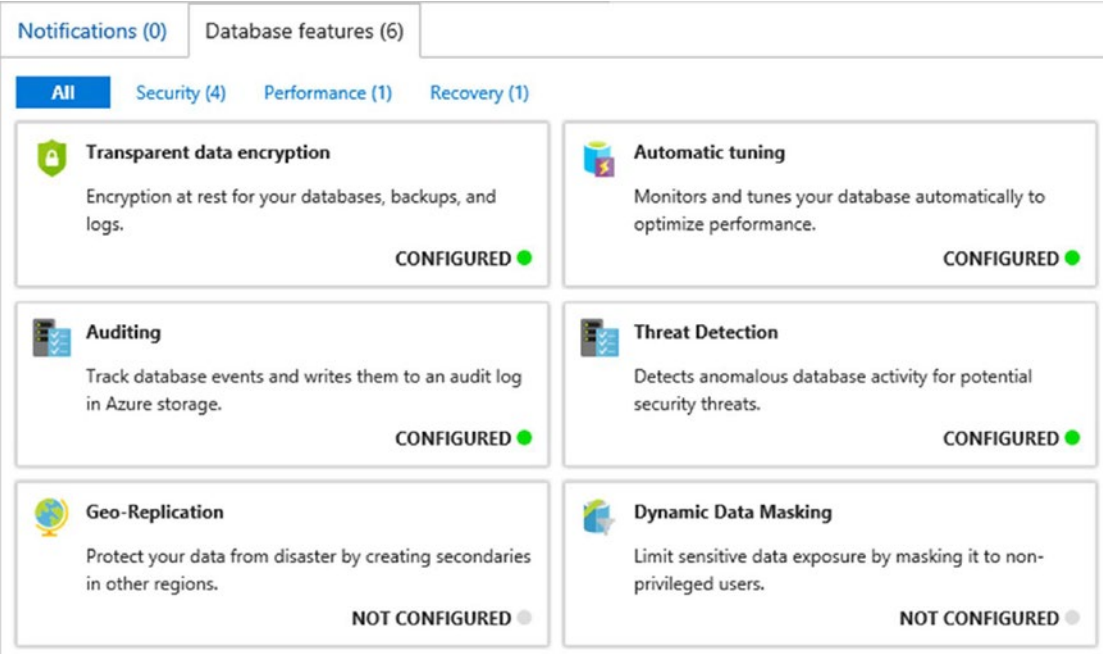


Figure 25-10. Database features in Azure SQL Database including “Automatic tuning”

Automatic tuning is the selection in the upper right. Just remember, Azure is subject to change, so your screen may look different from mine. Clicking the “Automatic tuning” button will open the screen shown in Figure 25-11.

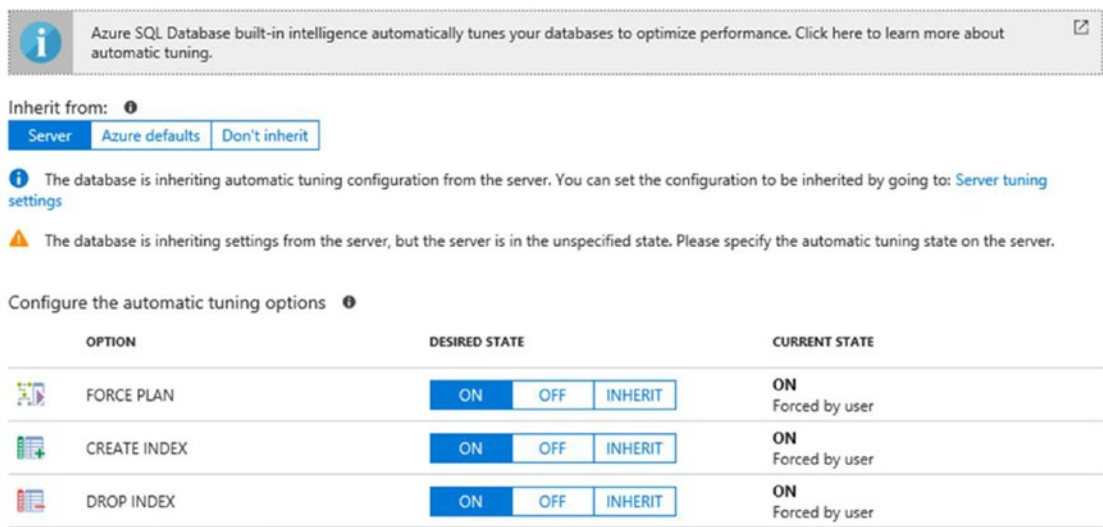






Figure 25-11. Enabling “Automatic tuning” within Azure SQL Database

 Automate	 View discarded	 Getting started	 Feedback
--	--	---	--

Recommendations

ACTION	RECOMMENDATION DESCRIPTION	IMPACT
--------	----------------------------	--------

There are no active recommendations at the moment.

Tuning history



ACTION	RECOMMENDATION DESCRIPTION	STATUS	TIME
 CREATE INDEX Initiated by: System	Table: [Customer] Indexed columns: [FirstName]	 Validating	4/26/2018 4:55:06 PM

Figure 25-13. Performance recommendations and tuning history

Since we have enabled all automatic tuning, there are no recommendations currently. The automated tuning has taken effect. However, we can still drill down and gather additional information. Click the **CREATE INDEX** choice to open a new window. When the automatic tuning has not yet been validated, the window will open by default in the Estimated Impact view, shown in Figure 25-14.

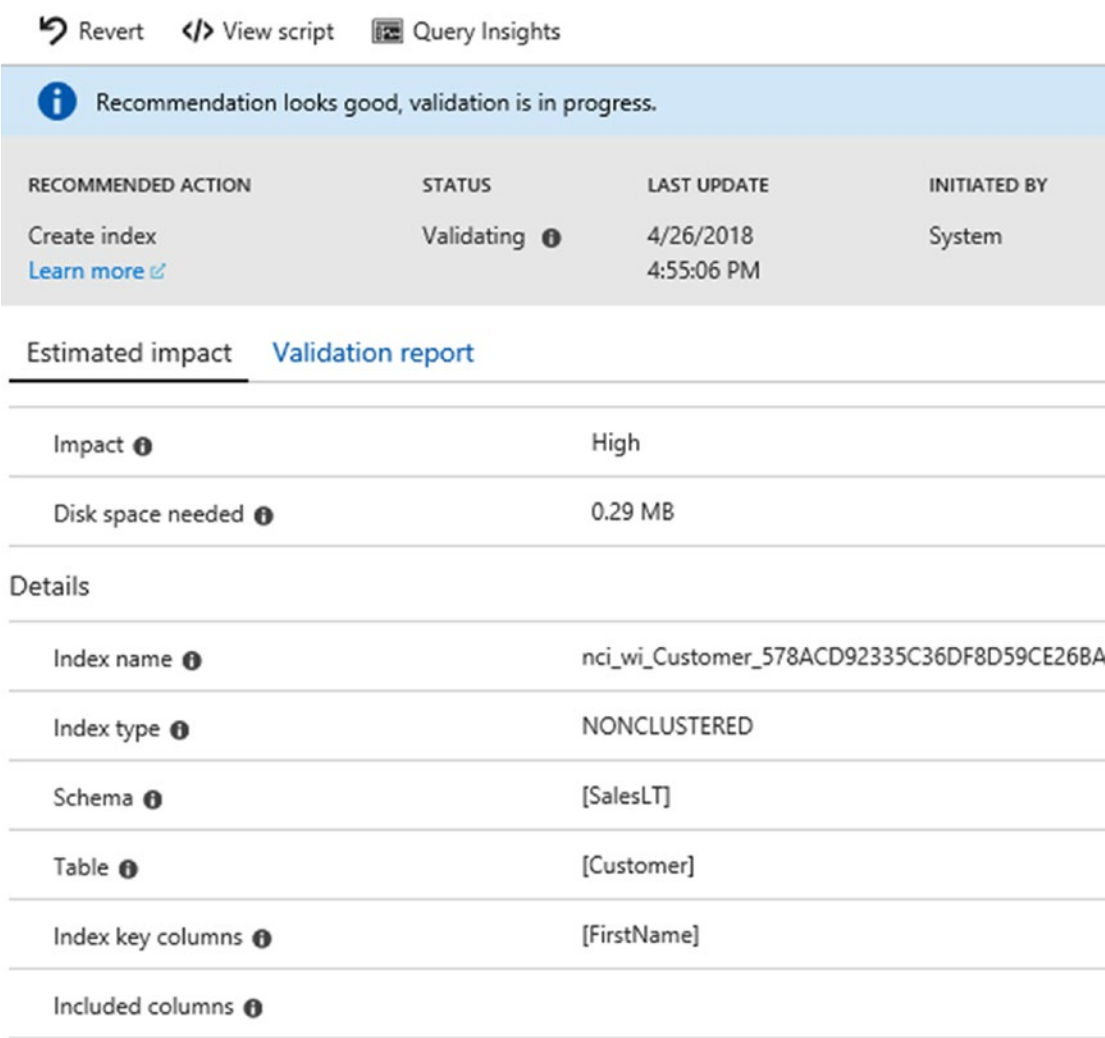


Figure 25-14. *Estimated Impact view of recommended index*

In my case, the index has already been created, and validation that the index is properly supporting the queries is underway. You get a good overview of the recommendation, and it should look familiar since the information is similar to that included in the earlier automatic plan tuning. The differences are in the details where, instead of a suggested plan, we have a suggested index, index type, schema, table, index column or columns, and any included columns.

You can take control of these automated changes by looking at the buttons across the top of the screen. You can remove the changes manually by clicking Revert. You can see the script used to generate the changes, and you can look at the query metrics collected.

My system opens by default on the Validation report, as shown in Figure 25-15.

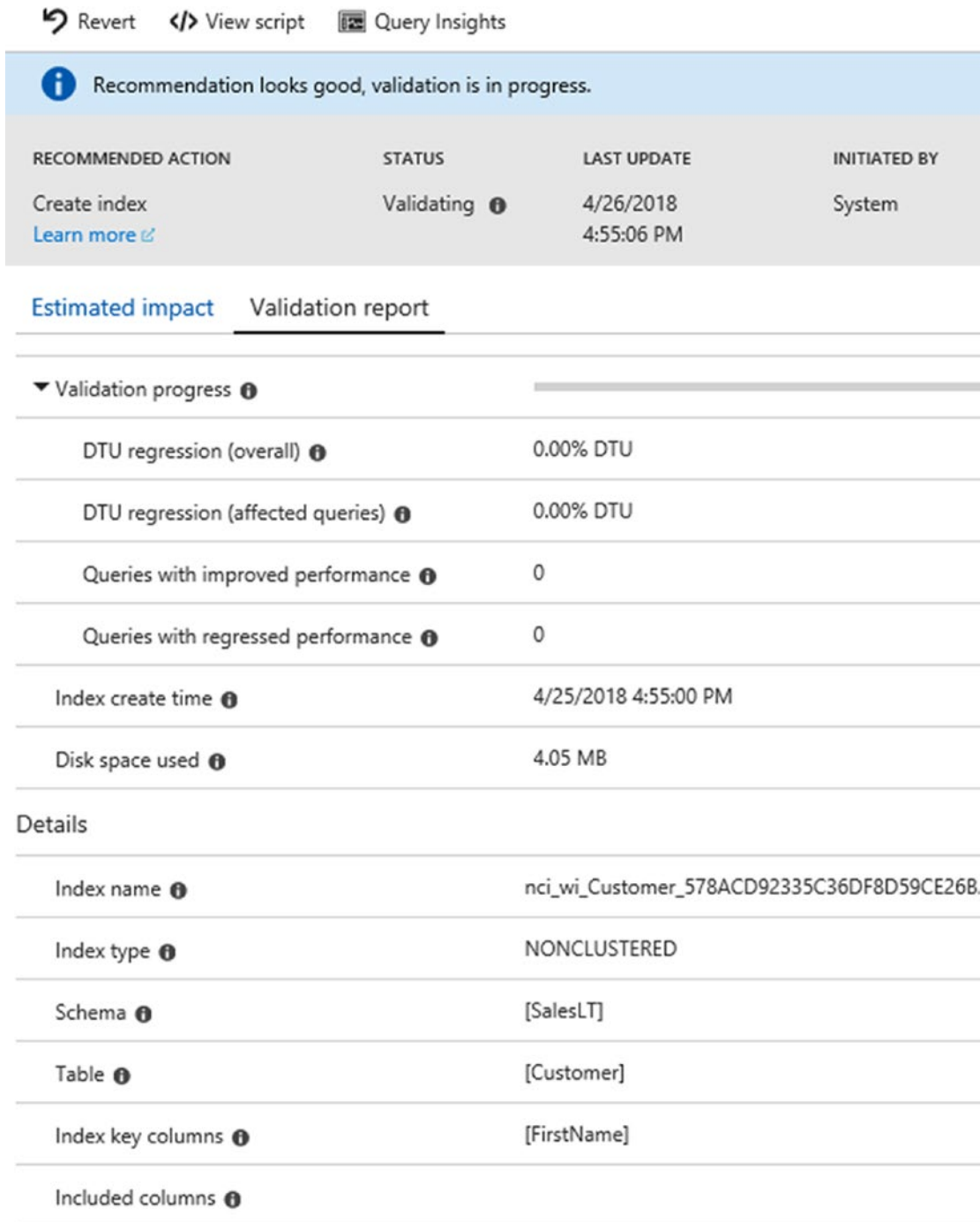




Figure 25-15. Validation in action during automatic index management

From this report you can see that the new index is in place, but it is currently going through an evaluation period. It's currently unclear exactly how long the evaluation period lasts, but it's safe to assume it's probably another 12 to 18 hours of load on the system during which any negative effects from the index will be measured and used to decide whether the index is kept. The exact time over which an evaluation is done is not published information, so even my estimates here could be subject to change.








As a part of demonstrating the behavior, I stopped running queries against the database during the validation period. This meant that any queries measured by the system were unlikely to have any kind of benefit from the new index. Because of that, two days later, the index reverted, meaning it was removed from the system. We can see this in the tuning history, as shown in [Figure 25-16](#).

 Recommendation has been reverted.

RECOMMENDED ACTION	STATUS	LAST UPDATE	INITIATED BY
Create index Learn more	Reverted 	4/28/2018 6:21:32 PM	System

[Estimated impact](#)

[Validation report](#)

▼ Validation progress 	Completed
DTU regression (overall) 	4135.75% DTU
DTU regression (affected queries) 	4628.93% DTU
Queries with improved performance 	0
Queries with regressed performance 	3
Index create time 	4/25/2018 4:55:00 PM
Disk space used 	4.05 MB

Details







Index name 	nci_wi_Customer_578ACD92335C36DF8D59CE2
Index type 	NONCLUSTERED
Schema 	[SalesLT]
Table 	[Customer]
Index key columns 	[FirstName]
Included columns 	

Figure 25-16. The index has been removed after the load changed

Assuming the load was kept in place, however, the index would have been validated as showing a performance improvement for the queries being called.

Adaptive Query Processing

Tuning queries is the purpose of this book, so talking about mechanisms that will make it so you don't have to tune quite so many queries does seem somewhat counterintuitive, but it's worth understanding exactly the places where SQL Server will automatically help you out. The new mechanisms outlined by adaptive query processing are fundamentally about changing the behavior of queries as the queries execute. This can help deal with some fundamental issues related to misestimated row counts and memory allocation. There are currently three types of adaptive query processing, and we'll demonstrate all three in this chapter:

- Batch mode memory grant feedback
- Batch mode adaptive join
- Interleaved execution

We already went over adaptive joins in Chapter 9. We'll deal with the other two mechanisms of adaptive query processing in order, starting with batch mode memory grant feedback.

Batch Mode Memory Grant Feedback

Batch mode, as of this writing, is supported only by queries that involve a columnstore index, clustered or nonclustered. Batch mode itself is worth a short explanation.

During row mode execution within an execution plan, each pair of operators has to negotiate each row being transferred between them. If there are ten rows, there are ten negotiations. If there are ten million rows, there are ten million negotiations. As you can imagine, this gets quite costly. So, in a batch mode operation, instead of processing each row one at a time, the processing occurs in batches, generally distributed up to 900 rows per batch, but there is quite a bit of variation there. This means, instead of ten million negotiations to move ten million rows, there are only this many:

$10000000 \text{ rows} / \sim 900 \text{ rows per batch} = 11,111 \text{ batches}$

Going from ten million negotiations to approximately 11,000 is a radical improvement. Additionally, because processing time has been freed up and because better row estimates are possible, you can get different behaviors within the execution of the query.

The first of the behaviors we'll explore is batch mode memory grant feedback. In this case, when a query gets executed in batch mode, calculations are made as to whether the query had excess or inadequate memory. Inadequate memory is especially a problem because it leads to having to allocate and use the disk to manage the excess, referred to as a *spill*. Having better memory allocation can improve performance. Let's explore an example.

First, for it to work, ensure you still have a columnstore index on your `bigTransactionHistory` table and that the compatibility mode of the database is set to 140.

Before we start, we can also ensure that we can observe the behavior by using Extended Events to capture events directly related to the memory grant feedback process. Here's a script that does that:

```
CREATE EVENT SESSION MemoryGrant
ON SERVER
    ADD EVENT sqlserver.memory_grant_feedback_loop_disabled
    (WHERE (sqlserver.database_name = N'AdventureWorks2017')),
    ADD EVENT sqlserver.memory_grant_updated_by_feedback
    (WHERE (sqlserver.database_name = N'AdventureWorks2017')),
    ADD EVENT sqlserver.sql_batch_completed
    (WHERE (sqlserver.database_name = N'AdventureWorks2017'))
WITH (TRACK_CAUSALITY = ON);
```

The first event, `memory_grant_feedback_loop_disabled`, occurs when the query in question is overly affected by parameter values. Instead of letting the memory grant swing wildly back and forth, the query engine will disable the feedback for some plans. When this happens to a plan, this event will fire during the execution. The second event, `memory_grant_updated_by_feedback`, occurs when the feedback is processed. Let's see that in action.

Here is a procedure with a query that aggregates some of the information from the bigTransactionHistory table:

```
CREATE OR ALTER PROCEDURE dbo.CostCheck (@Cost MONEY)
AS
SELECT p.Name,
       AVG(th.Quantity),
       AVG(th.ActualCost)
FROM dbo.bigTransactionHistory AS th
      JOIN dbo.bigProduct AS p
        ON p.ProductID = th.ProductID
WHERE th.ActualCost = @Cost
GROUP BY p.Name;
GO
```

If we execute this query, passing it the value of 0, and capture the actual execution plan, it looks like Figure 25-17.

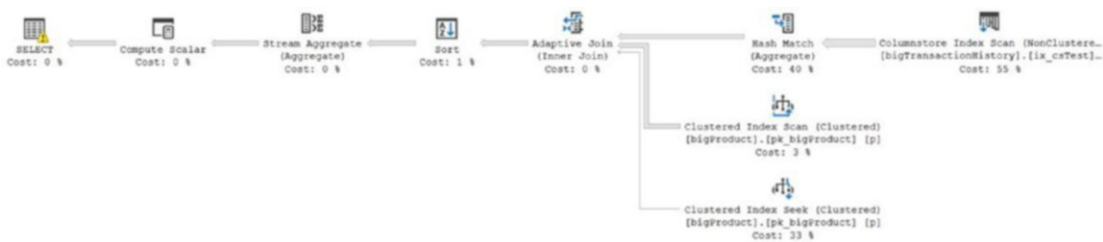


Figure 25-17. Execution plan with a warning on the SELECT operator

What should immediately draw your eye with this query is the warning on the SELECT operator. We can open the Properties window to see all the warnings for a plan. Note that the tooltip only ever shows the first warning, as shown in Figure 25-18.

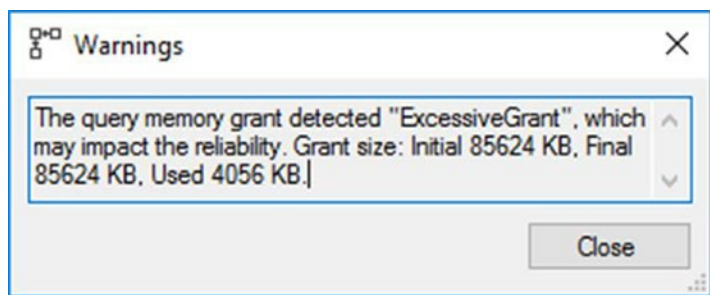


Figure 25-18. Excessive memory grant warning from the execution plan

The definition here is fairly clear. Based on the statistics for the data stored in the columnstore index, SQL Server assumed that to process the data for this query, it would need 85,624KB. The actual memory used was 4,056KB. That’s a more than 81,000KB difference. If queries like this ran a lot with this sort of disparity, we would be facing serious memory pressure without much in the way of benefit. We can also look to the Extended Events to see the feedback process in action. Figure 25-19 shows the `memory_grant_updated_by_feedback` event that fired as part of executing the query.

name	timestamp
memory_grant_updated_by_feedback	2018-05-08 17:34:13.1117987

Event: memory_grant_updated_by_feedback (2018-05-08 17:34:13.1117987)

Details

Field	Value
attach_activity_id.guid	2042827C-6E97-4AA6-A440-2646048802CE
attach_activity_id.seq	1
attach_activity_id_xfer.guid	21FF226E-9321-48EA-8880-BE32BCCBDF0C
attach_activity_id_xfer.seq	0
history_current_execution_count	1
history_update_count	1
ideal_additional_memory_after_kb	1024
ideal_additional_memory_before_kb	80352

Figure 25-19. Extended Events properties for the `memory_grant_updated_by_feedback` event

You can see in Figure 25-19 some important information. The activity_id values show that this event occurred before the others in the Extended Events session since the seq value is 1. If you’re running the code, you’ll see that your sql_batch_completed had a seq value of 2. This means the memory grant adjustments occur before the query completes execution, although you still get the warning in the plan. These adjustments are for subsequent executions of the query. In fact, let’s execute the query again and look at the results of the query execution in Extended Events, as shown in Figure 25-20.

name	timestamp	duration	logical_reads	batch_text
memory_grant_updated_by_feedback	2018-05-08 17:47:46.5769512	NULL	NULL	NULL
sql_batch_completed	2018-05-08 17:47:46.5772760	669914	42687	EXEC dbo.CostCheck @Cost = 0;
sql_batch_completed	2018-05-08 17:47:47.4654723	161	0	SELECT @@SPID;
sql_batch_completed	2018-05-08 17:47:48.1416143	665917	42644	EXEC dbo.CostCheck @Cost = 0;

Figure 25-20. Extended Events showing the memory grant feedback occurs only once

The other interesting thing to note is that if you capture the execution plan again, as shown in Figure 25-21, you are no longer seeing the warning.

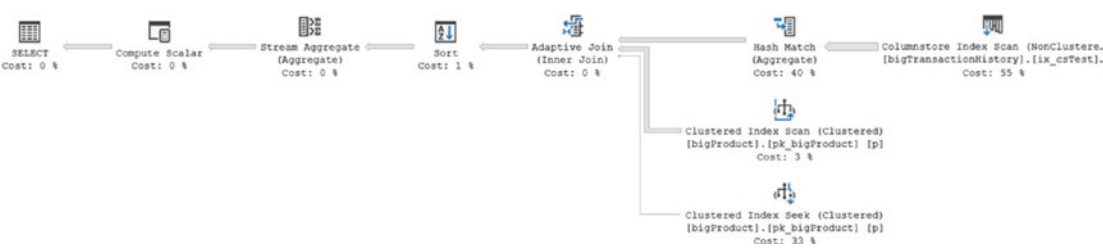


Figure 25-21. The same execution plan but without a warning

If we were to continue running this procedure using these parameter values, you wouldn’t see any other changes. However, if we were to modify the parameter values as follows:

```
EXEC dbo.CostCheck @Cost = 15.035;
```

we wouldn’t see any changes at all. This is because that while the result sets are quite different, 1 row versus 9000, the memory requirements are not so wildly different as what we saw in the first execution of the first query. However, if we were to clear the memory cache and then execute the procedure using these values, you would again see the memory_grant_updated_by_feedback firing.

If you are experiencing issues with some degree of thrash caused by changing the memory grant, you can disable it on a database level using DATABASE SCOPED CONFIGURATION as follows:

```
ALTER DATABASE SCOPED CONFIGURATION SET DISABLE_BATCH_MODE_MEMORY_GRANT_
FEEDBACK = ON;
```

To reenable it, just use the same command to set it to OFF. There is also a query hint that you can use to disable the memory feedback for a single query. Simply add DISABLE_BATCH_MODE_MEMORY_GRANT_FEEDBACK to the USE part of the query hint.

Interleaved Execution

While my recommendation of avoiding the use of multistatement table-valued functions remains the same, you may find yourself forced to deal with them. Prior to SQL Server 2017, the only real option for making these run faster was to rewrite the code to not use them at all. However, SQL Server 2017 now has interleaved execution for these objects. The way it works is that the optimizer will identify that it is dealing with one of these multistatement functions. It will pause the optimization process. The part of the plan dealing with the table-valued function will execute, and accurate row counts will be returned. These row counts will then be used through the rest of the optimization process. If you have more than one multistatement function, you'll get multiple executions until all such objects have more accurate row counts returned.

To see this in action, I want to create the following multistatement functions:

```
CREATE OR ALTER FUNCTION dbo.SalesInfo ()
RETURNS @return_variable TABLE (SalesOrderID INT,
                                OrderDate DATETIME,
                                SalesPersonID INT,
                                PurchaseOrderNumber dbo.OrderNumber,
                                AccountNumber dbo.AccountNumber,
                                ShippingCity NVARCHAR(30))
AS
BEGIN;
    INSERT INTO @return_variable (SalesOrderID,
                                OrderDate,
                                SalesPersonID,
```

```

        PurchaseOrderNumber,
        AccountNumber,
        ShippingCity)

SELECT soh.SalesOrderID,
       soh.OrderDate,
       soh.SalesPersonID,
       soh.PurchaseOrderNumber,
       soh.AccountNumber,
       a.City
FROM Sales.SalesOrderHeader AS soh
     JOIN Person.Address AS a
       ON soh.ShipToAddressID = a.AddressID;
RETURN;
END;
GO

CREATE OR ALTER FUNCTION dbo.SalesDetails ()
RETURNS @return_variable TABLE (SalesOrderID INT,
                                SalesOrderDetailID INT,
                                OrderQty SMALLINT,
                                UnitPrice MONEY)
AS
BEGIN;
    INSERT INTO @return_variable (SalesOrderID,
                                SalesOrderDetailID,
                                OrderQty,
                                UnitPrice)

    SELECT sod.SalesOrderID,
           sod.SalesOrderDetailID,
           sod.OrderQty,
           sod.UnitPrice
    FROM Sales.SalesOrderDetail AS sod;
RETURN;
END;
GO

```

```

CREATE OR ALTER FUNCTION dbo.CombinedSalesInfo ()
RETURNS @return_variable TABLE (SalesPersonID INT,
                                ShippingCity NVARCHAR(30),
                                OrderDate DATETIME,
                                PurchaseOrderNumber dbo.OrderNumber,
                                AccountNumber dbo.AccountNumber,
                                OrderQty SMALLINT,
                                UnitPrice MONEY)

AS
BEGIN;
    INSERT INTO @return_variable (SalesPersonID,
                                ShippingCity,
                                OrderDate,
                                PurchaseOrderNumber,
                                AccountNumber,
                                OrderQty,
                                UnitPrice)

    SELECT si.SalesPersonID,
           si.ShippingCity,
           si.OrderDate,
           si.PurchaseOrderNumber,
           si.AccountNumber,
           sd.OrderQty,
           sd.UnitPrice
    FROM dbo.SalesInfo() AS si
         JOIN dbo.SalesDetails() AS sd
           ON si.SalesOrderID = sd.SalesOrderID;
RETURN;
END;
GO

```

These are the types of anti-patterns (or code smells) I see so frequently when working with multistatement functions. One function calls another, which joins to a third, and so on. Since the optimizer will do one of two things with these functions, depending on the version of the cardinality estimation engine in use, you have no real choices. Prior to SQL Server 2014 the optimizer assumed one row for these objects.

SQL Server 2014 and greater have a different assumption, 100 rows. So if the compatibility level is set to 140 or greater, you'll see the 100-row assumption, except if interleaved execution is enabled.

We can run a query against these functions. However, first we'll want to run it with interleaved execution disabled. Then we'll reenale it, clear the cache, and execute the query again as follows:

```
ALTER DATABASE SCOPED CONFIGURATION SET DISABLE_INTERLEAVED_EXECUTION_TVF = ON;
GO
```

```
SELECT csi.OrderDate,
       csi.PurchaseOrderNumber,
       csi.AccountNumber,
       csi.OrderQty,
       csi.UnitPrice,
       sp.SalesQuota
FROM dbo.CombinedSalesInfo() AS csi
     JOIN Sales.SalesPerson AS sp
       ON csi.SalesPersonID = sp.BusinessEntityID
WHERE csi.SalesPersonID = 277
     AND csi.ShippingCity = 'Odessa';
GO
```

```
ALTER DATABASE SCOPED CONFIGURATION SET DISABLE_INTERLEAVED_EXECUTION_TVF = OFF;
GO
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
```

```
SELECT csi.OrderDate,
       csi.PurchaseOrderNumber,
       csi.AccountNumber,
       csi.OrderQty,
       csi.UnitPrice,
       sp.SalesQuota
FROM dbo.CombinedSalesInfo() AS csi
     JOIN Sales.SalesPerson AS sp
       ON csi.SalesPersonID = sp.BusinessEntityID
```

```

WHERE csi.SalesPersonID = 277
      AND csi.ShippingCity = 'Odessa';
GO

```

The resulting execution plans are different, but the differences are subtle, as you can see in Figure 25-22.

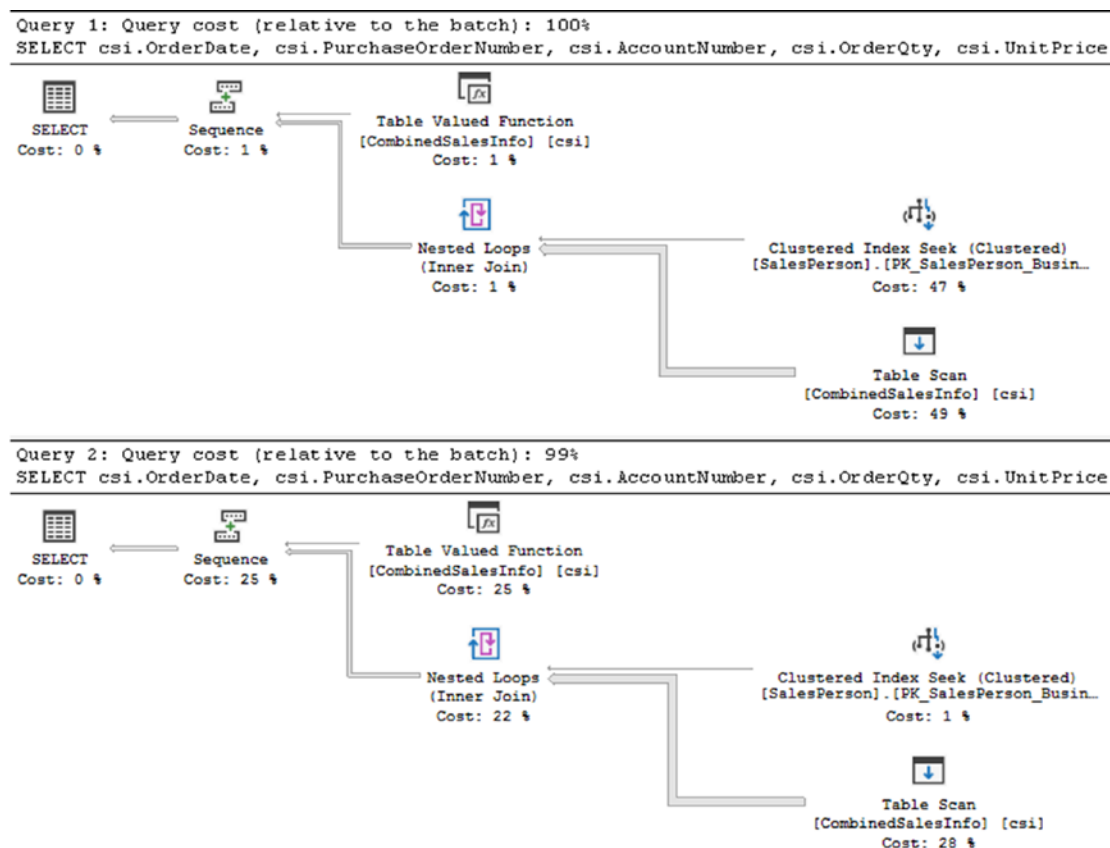


Figure 25-22. Two execution plans, one with interleaved execution

Looking at the plans, it's actually difficult to see the differences since all the operators are the same. However, the differences are in the estimated cost values. At the top, the table-valued function has an estimated cost of 1 percent, suggesting that it's almost free when compared to the Clustered Index Seek and Table Scan operations. In the second plan, though, the Clustered Index Seek, returning an estimated one row, suddenly costs only an estimated 1 percent of the total, and the rest of the cost is rightly

redistributed to the other operations. It's these differences in row estimates that may lead, in some situations, to enhanced performance. However, let's look at the values to see this in action.

The Sequence operator forces each subtree attached to it to fire in order. In this instance, the first to fire would be the table-valued function. It's supplying data to the Table Scan operator at the bottom of both plans. Figure 25-23 shows the properties for the top plan (the plan that executed in the old way).

Actual Number of Rows	148
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[AdventureWc
Description	Scan rows from
Estimated CPU Cost	0.000267
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	1
Estimated Number of Rows	3.16228
Estimated Number of Rows to be Read	100

Figure 25-23. *The properties of the old-style plan*

At the bottom of the image captured in Figure 25-23 you can see the estimated number of rows to be read from the operator is 100. Of these, an expected number of matching rows was anticipated as 3.16228. The actual number of rows is at the top and is 148. That disparity is a major part of what leads to such poor execution times for multistatement functions.

Now, let's look at the same properties for the function that executed in an interleaved fashion, as shown in Figure 25-24.

Actual Number of Rows	148
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[Adventure\
Description	Scan rows fr
Estimated CPU Cost	0.133606
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	1
Estimated Number of Rows	18.663
Estimated Number of Rows to be Read	121317

Figure 25-24. *The properties of the interleaved execution*

The same number of actual rows was returned because these are identical queries against identical result sets. However, look at the Estimated Number of Rows to be Read value. Instead of the hard-coded value of 100, regardless of the data involved, we now have 121317. That is a much more accurate estimate. It resulted in an anticipated 18.663 rows being returned. That's still not the actual value of 148, but it's moving toward a more accurate estimate.

Since these plans are similar, the chances of much of a difference in execution times and reads is unlikely. However, let's get the measures from Extended Events. On average the noninterleaved execution was 1.48 seconds with 341,000 reads. The interleaved execution ran in 1.45 seconds on average and had 340,000 reads. There was a small improvement.

Now, we can actually improve the performance remarkably and still use a multistatement function. Instead of joining functions together, if we were to rewrite the code something like this:

```
CREATE OR ALTER FUNCTION dbo.AllSalesInfo (@SalesPersonID INT,
                                           @ShippingCity VARCHAR(50))
RETURNS @return_variable TABLE (SalesPersonID INT,
                                ShippingCity NVARCHAR(30),
                                OrderDate DATETIME,
                                PurchaseOrderNumber dbo.OrderNumber,
                                AccountNumber dbo.AccountNumber,
                                OrderQty SMALLINT,
                                UnitPrice MONEY)
```