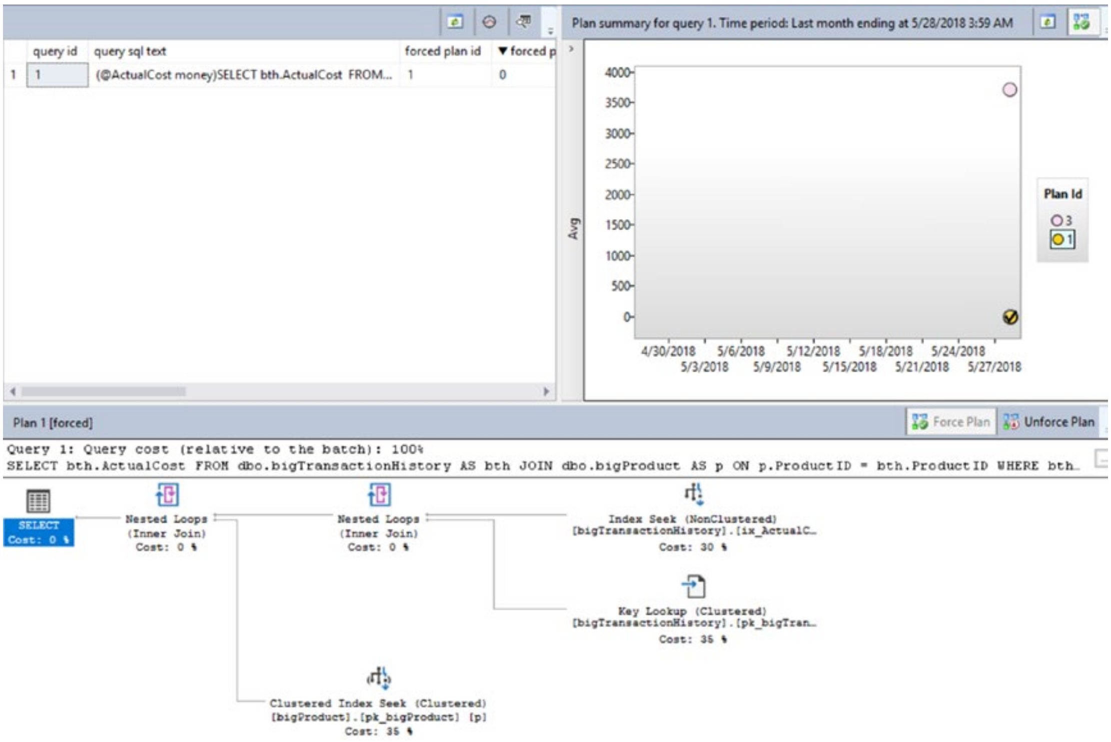
If you restart the server, the information in `sys.dm_db_tuning_recommendations` will be removed. Also, any plans that have been forced will be removed. As soon as a query regresses again, any plan forcing will be automatically reenabled, assuming the Query Store history is there. If this is an issue, you can always force the plan manually.

If a query is forced and then performance degrades, it will be unforced, as already noted. If that query again suffers from degraded performance, plan forcing will be removed, and the query will be marked such that, at least until a server reboot when the information is removed, it will not be forced again.

We can also see the forced plan if we look to the Query Store reports. Figure 25-9 shows the result of the plan forcing from the automated tuning.



	query id	query sql text	forced plan id	▼forced p
1	1	(@ActualCost money)SELECT bth.ActualCost FROM...	1	0

Plan summary for query 1. Time period: Last month ending at 5/28/2018 3:59 AM

Plan 1 [forced]

```
Query 1: Query cost (relative to the batch): 100%
SELECT bth.ActualCost FROM dbo.bigTransactionHistory AS bth JOIN dbo.bigProduct AS p ON p.ProductID = bth.ProductID WHERE bth…
```

```
SELECT
Cost: 0 %

Nested Loops
(Inner Join)
Cost: 0 %

Nested Loops
(Inner Join)
Cost: 0 %

Index Seek (NonClustered)
[bigTransactionHistory].[ix_ActualC…
Cost: 30 %

Key Lookup (Clustered)
[bigTransactionHistory].[pk_bigTran…
Cost: 35 %

Clustered Index Seek (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 35 %
```

Figure 25-9. *The Queries with Forced Plans report showing the result of automated tuning*

These reports won't show you why the plan is forced. However, you can always go to the DMVs for that information if needed.

Azure SQL Database Automatic Index Management

Automatic index management goes to the heart of the concept of Azure SQL Database being positioned as a Platform as a Service (PaaS). A large degree of functionality such as patching, backups, and corruption testing, along with high availability and a bunch of others, are all managed for you inside the Microsoft cloud. It just makes sense that they can also put their knowledge and management of the systems to work on indexes. Further, because all the processing for Azure SQL Database is taking place inside Microsoft's server farms in Azure, they can put their machine learning algorithms to work when monitoring your systems.

Note that Microsoft doesn't gather private information from your queries, data, or any of the information stored there. It simply uses the query metrics to measure behavior. It's important to state this up front because misinformation has been transmitted about these functions.

Before we enable index management, though, let's generate some bad query behavior. I'm using two scripts against the sample database within Azure, AdventureWorksLT. When you provision a database within Azure, the example database, one of your choices in the portal, is simple and easy to immediately implement. That's why I like to use it for examples. To get started, here's a T-SQL script to generate some stored procedures:

```
CREATE OR ALTER PROCEDURE dbo.CustomerInfo
(@Firstname NVARCHAR(50))
AS
SELECT c.FirstName,
       c.LastName,
       c.Title,
       a.City
FROM SalesLT.Customer AS c
    JOIN SalesLT.CustomerAddress AS ca
        ON ca.CustomerID = c.CustomerID
    JOIN SalesLT.Address AS a
        ON a.AddressID = ca.AddressID
WHERE c.FirstName = @Firstname;
GO
```

```
CREATE OR ALTER PROCEDURE dbo.EmailInfo (@EmailAddress nvarchar(50))
AS
SELECT c.EmailAddress,
       c.Title,
       soh.OrderDate
FROM SalesLT.Customer AS c
    JOIN SalesLT.SalesOrderHeader AS soh
        ON soh.CustomerID = c.CustomerID
WHERE c.EmailAddress = @EmailAddress;
GO

CREATE OR ALTER PROCEDURE dbo.SalesInfo (@firstName NVARCHAR(50))
AS
SELECT c.FirstName,
       c.LastName,
       c.Title,
       soh.OrderDate
FROM SalesLT.Customer AS c
    JOIN SalesLT.SalesOrderHeader AS soh
        ON soh.CustomerID = c.CustomerID
WHERE c.FirstName = @firstName
GO

CREATE OR ALTER PROCEDURE dbo.OddName (@FirstName NVARCHAR(50))
AS
SELECT c.FirstName
FROM SalesLT.Customer AS c
WHERE c.FirstName BETWEEN 'Brian'
                  AND     @FirstName
GO
```

Next, here is a PowerShell script to call these procedures multiple times:

```
$SqlConnection = New-Object System.Data.SqlClient.SqlConnection
$SqlConnection.ConnectionString = 'Server=qpf.database.windows.net;Database
=QueryPerformanceTuning;trusted_connection=false;user=UserName;password=You
rPassword'
```

```
## load customer names
$DatCmd = New-Object System.Data.SqlClient.SqlCommand
$DatCmd.CommandText = "SELECT c.FirstName, c.EmailAddress
FROM SalesLT.Customer AS c;"
$DatCmd.Connection = $SqlConnection
$DatDataSet = New-Object System.Data.DataSet
$SqlAdapter = New-Object System.Data.SqlClient.SqlDataAdapter
$SqlAdapter.SelectCommand = $DatCmd
$SqlAdapter.Fill($DatDataSet)

$Proccmd = New-Object System.Data.SqlClient.SqlCommand
$Proccmd.CommandType = [System.Data.CommandType]'StoredProcedure'
$Proccmd.CommandText = "dbo.CustomerInfo"
$Proccmd.Parameters.Add("@FirstName",[System.Data.SqlDbType]"nvarchar")
$Proccmd.Connection = $SqlConnection

$EmailCmd = New-Object System.Data.SqlClient.SqlCommand
$EmailCmd.CommandType = [System.Data.CommandType]'StoredProcedure'
$EmailCmd.CommandText = "dbo.EmailInfo"
$EmailCmd.Parameters.Add("@EmailAddress",[System.Data.SqlDbType]"nvarchar")
$EmailCmd.Connection = $SqlConnection

$SalesCmd = New-Object System.Data.SqlClient.SqlCommand
$SalesCmd.CommandType = [System.Data.CommandType]'StoredProcedure'
$SalesCmd.CommandText = "dbo.SalesInfo"
$SalesCmd.Parameters.Add("@FirstName",[System.Data.SqlDbType]"nvarchar")
$SalesCmd.Connection = $SqlConnection

$OddCmd = New-Object System.Data.SqlClient.SqlCommand
$OddCmd.CommandType = [System.Data.CommandType]'StoredProcedure'
$OddCmd.CommandText = "dbo.OddName"
$OddCmd.Parameters.Add("@FirstName",[System.Data.SqlDbType]"nvarchar")
$OddCmd.Connection = $SqlConnection
```

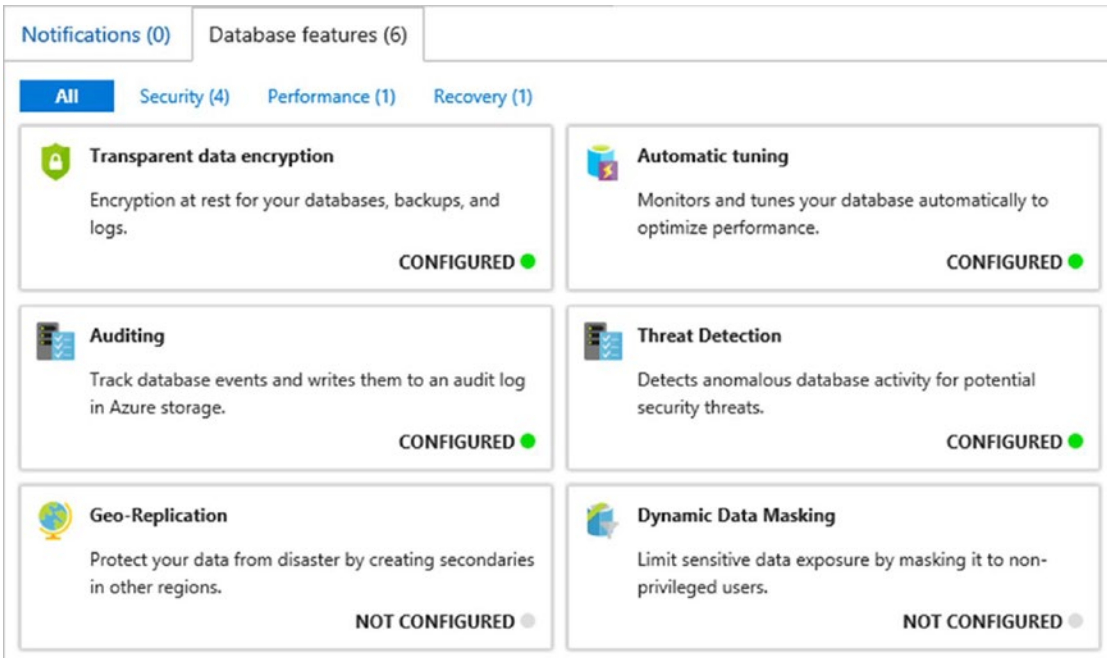
```
while(1 -ne 0)
{
    foreach($row in $DatDataSet.Tables[0])
        {

        $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.



Transparent data encryption
Encryption at rest for your databases, backups, and logs.
CONFIGURED ●

Automatic tuning
Monitors and tunes your database automatically to optimize performance.
CONFIGURED ●

Auditing
Track database events and writes them to an audit log in Azure storage.
CONFIGURED ●

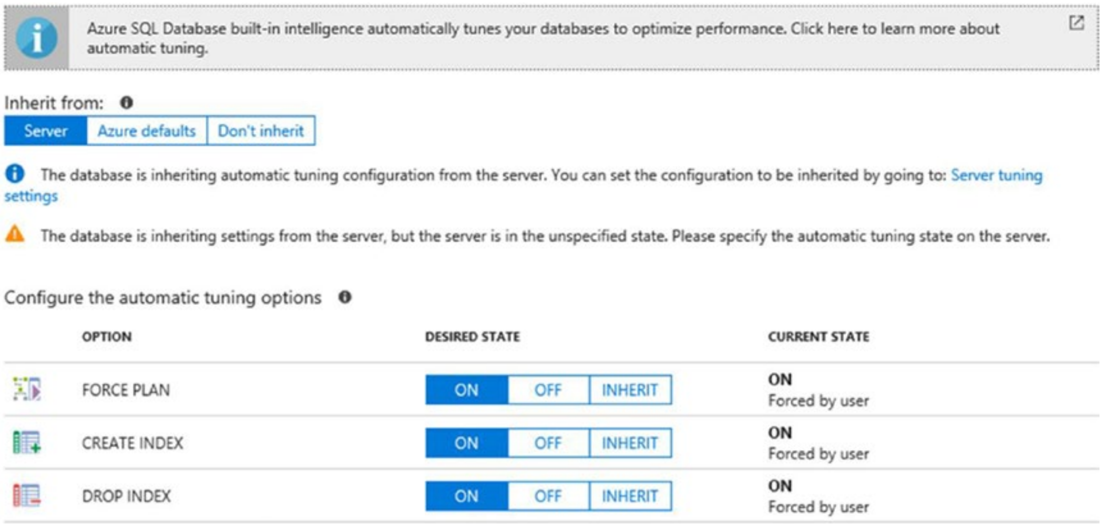
Threat Detection
Detects anomalous database activity for potential security threats.
CONFIGURED ●

Geo-Replication
Protect your data from disaster by creating secondaries in other regions.
NOT CONFIGURED ○

Dynamic Data Masking
Limit sensitive data exposure by masking it to non-privileged users.
NOT CONFIGURED ○

***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.


Azure SQL Database built-in intelligence automatically tunes your databases to optimize performance. Click here to learn more about automatic tuning.

Inherit from: [✓ Server] Azure defaults Don't inherit

The database is inheriting automatic tuning configuration from the server. You can set the configuration to be inherited by going to: Server tuning settings

The database is inheriting settings from the server, but the server is in the unspecified state. Please specify the automatic tuning state on the server.

Configure the automatic tuning options

OPTION	DESIRED STATE	CURRENT STATE
FORCE PLAN	[✓ ON] OFF INHERIT	ON Forced by user
CREATE INDEX	[✓ ON] OFF INHERIT	ON Forced by user
DROP INDEX	[✓ ON] OFF INHERIT	ON Forced by user

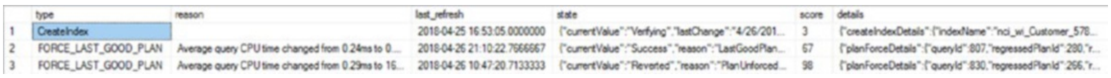
***Figure 25-11.** Enabling "Automatic tuning" within Azure SQL Database*

In this case, I have enabled all three options, so not only will I get the last good plan through automatic tuning as described in the earlier section, but I also have now turned on automatic index management.

With these features enabled, we can now run the PowerShell script for at least 12 hours. You can validate whether you have received an index by querying `sys.dm_db_tuning_recommendations` as we did earlier. Here I'm using the simple script that just retrieves the core information from the DMV:

```
SELECT ddtr.type,
       ddtr.reason,
       ddtr.last_refresh,
       ddtr.state,
       ddtr.score,
       ddtr.details
FROM sys.dm_db_tuning_recommendations AS ddtr;
```

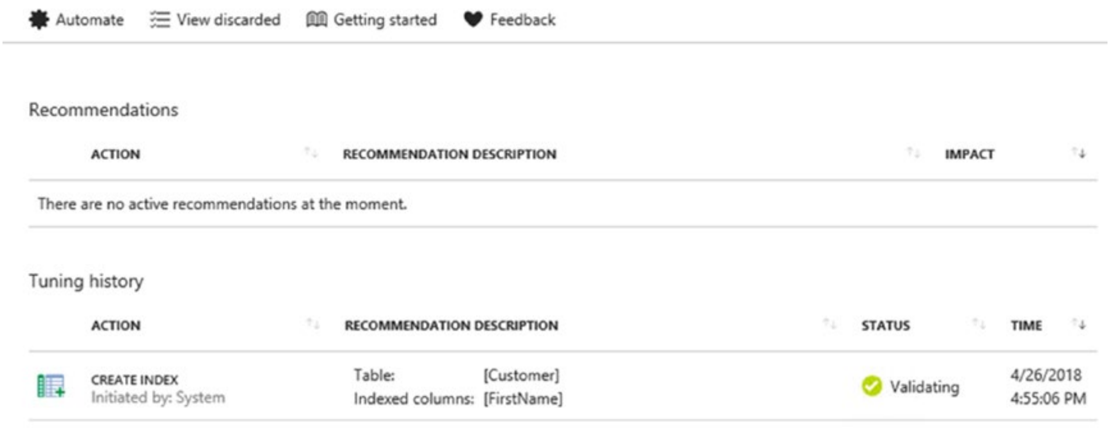
The results on my Azure SQL Database look something like Figure 25-12.



	type	reason	last_refresh	state	score	details
1	CreateIndex		2018-04-25 16:53:05.0000000	{"currentValue":"Verifying","lastChange":"4/26/201...	3	{"createIndexDetails":{"indexName":"nci_wi_Customer_578...
2	FORCE_LAST_GOOD_PLAN	Average query CPU time changed from 0.24ms to 0...	2018-04-26 21:10:22.7666667	{"currentValue":"Success","reason":"LastGoodPlan...	67	{"planForceDetails":{"queryId":807,"regressedPlanId":280,"r...
3	FORCE_LAST_GOOD_PLAN	Average query CPU time changed from 0.29ms to 16...	2018-04-26 10:47:20.7133333	{"currentValue":"Reverted","reason":"PlanUnforced...	98	{"planForceDetails":{"queryId":830,"regressedPlanId":256,"r...

Figure 25-12. *Results of automatic tuning within Azure SQL Database*

As you can see, there have been multiple tuning events on my system. The first one is the one we're interested in for this example, the `CreateIndex` type. You can also look to the Azure portal to retrieve the behavior of automatic tuning. On the left side of your portal screen, near the bottom, you should see a Support + Troubleshooting choice labeled "Performance recommendations." Selecting that will open a window like Figure 25-13.



Recommendations

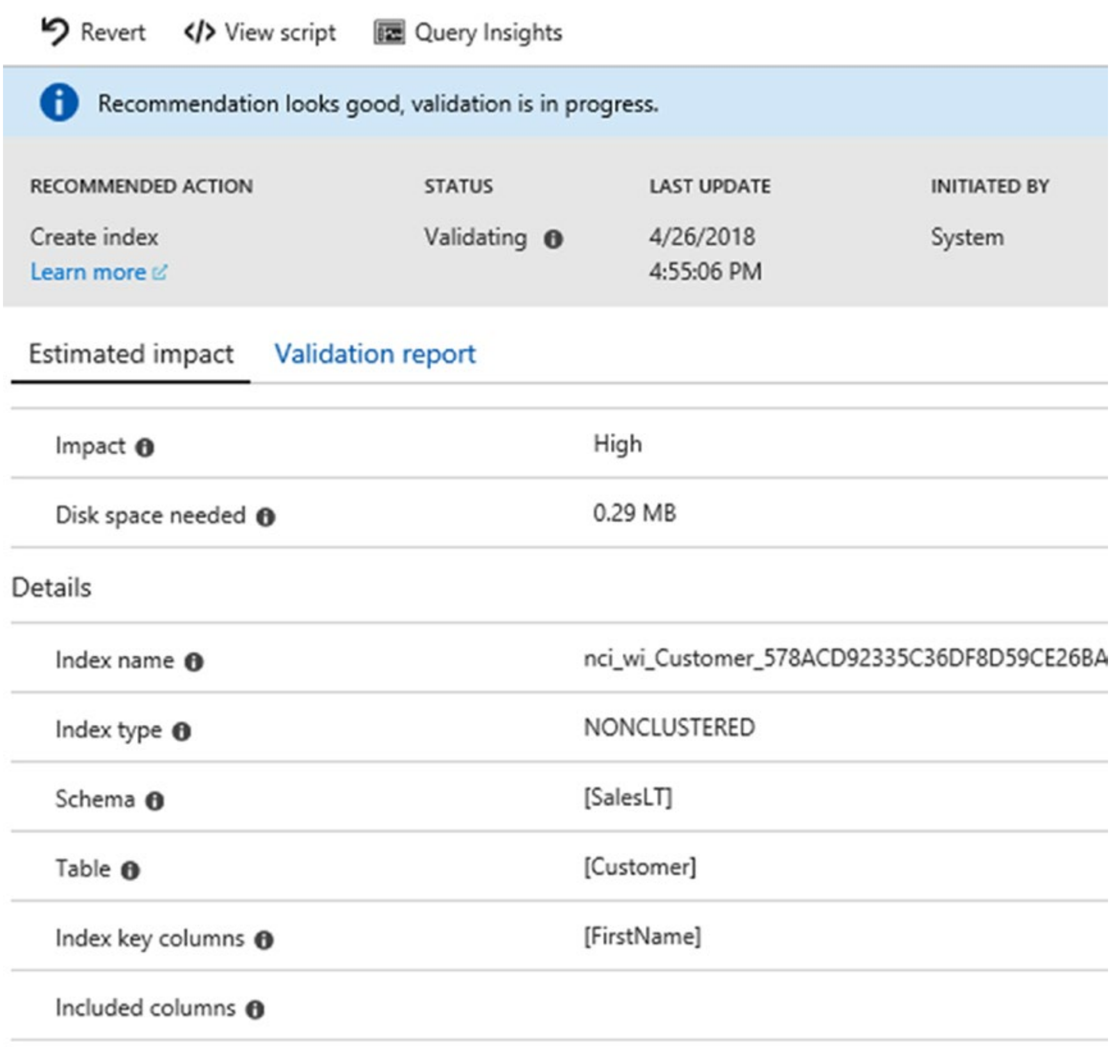
ACTION	RECOMMENDATION DESCRIPTION	IMPACT
There are no active recommendations at the moment.		

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.



Recommendation looks good, validation is in progress.

RECOMMENDED ACTION	STATUS	LAST UPDATE	INITIATED BY
Create index Learn more	Validating	4/26/2018 4:55:06 PM	System

Impact	High
Disk space needed	0.29 MB

Details

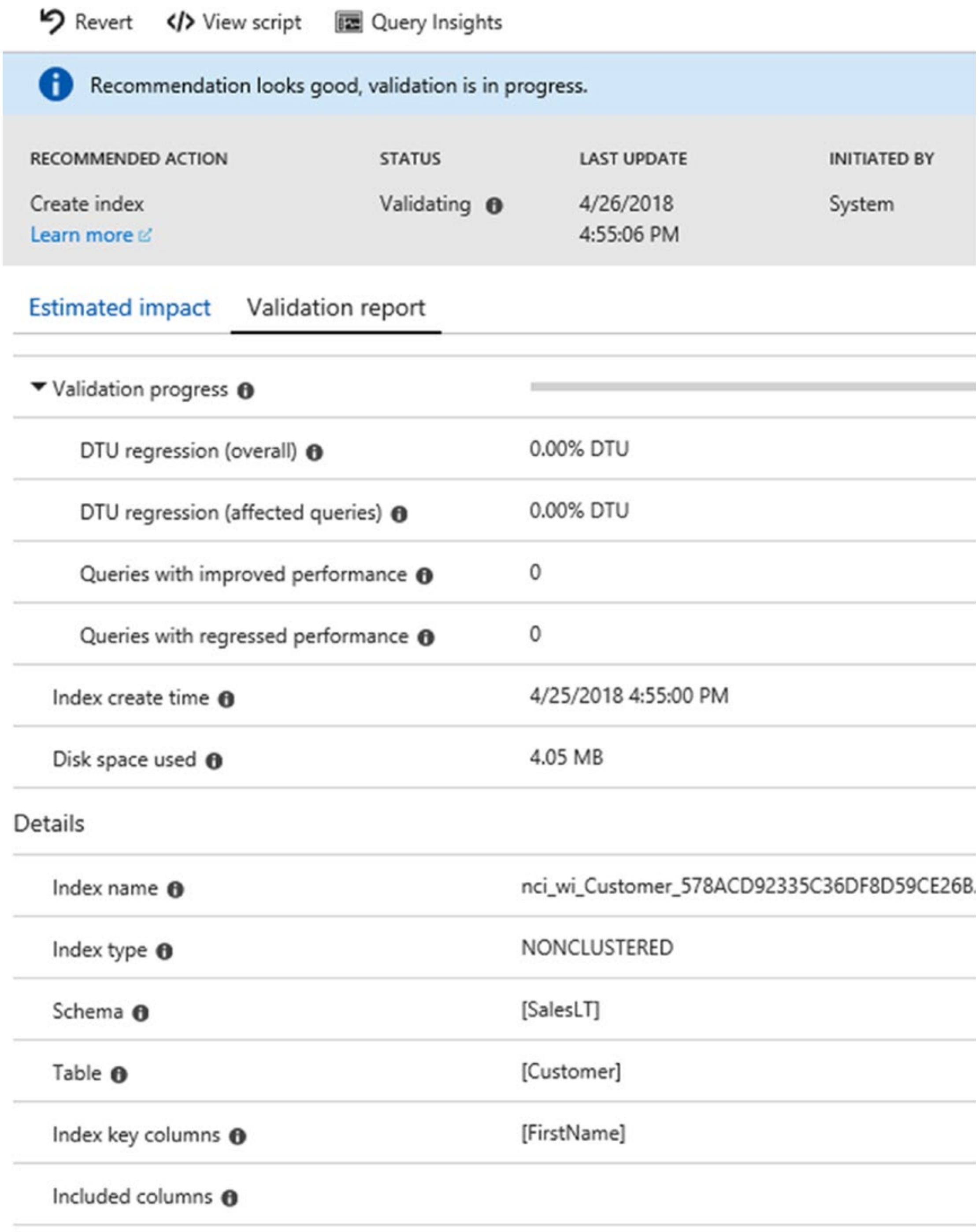
Index name	nci_wi_Customer_578ACD92335C36DF8D59CE26BA
Index type	NONCLUSTERED
Schema	[SalesLT]
Table	[Customer]
Index key columns	[FirstName]
Included columns	

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.



Recommendation looks good, validation is in progress.

RECOMMENDED ACTION	STATUS	LAST UPDATE	INITIATED BY
Create index Learn more	Validating	4/26/2018 4:55:06 PM	System

Validation progress	
DTU regression (overall)	0.00% DTU
DTU regression (affected queries)	0.00% DTU
Queries with improved performance	0
Queries with regressed performance	0
Index create time	4/25/2018 4:55:00 PM
Disk space used	4.05 MB

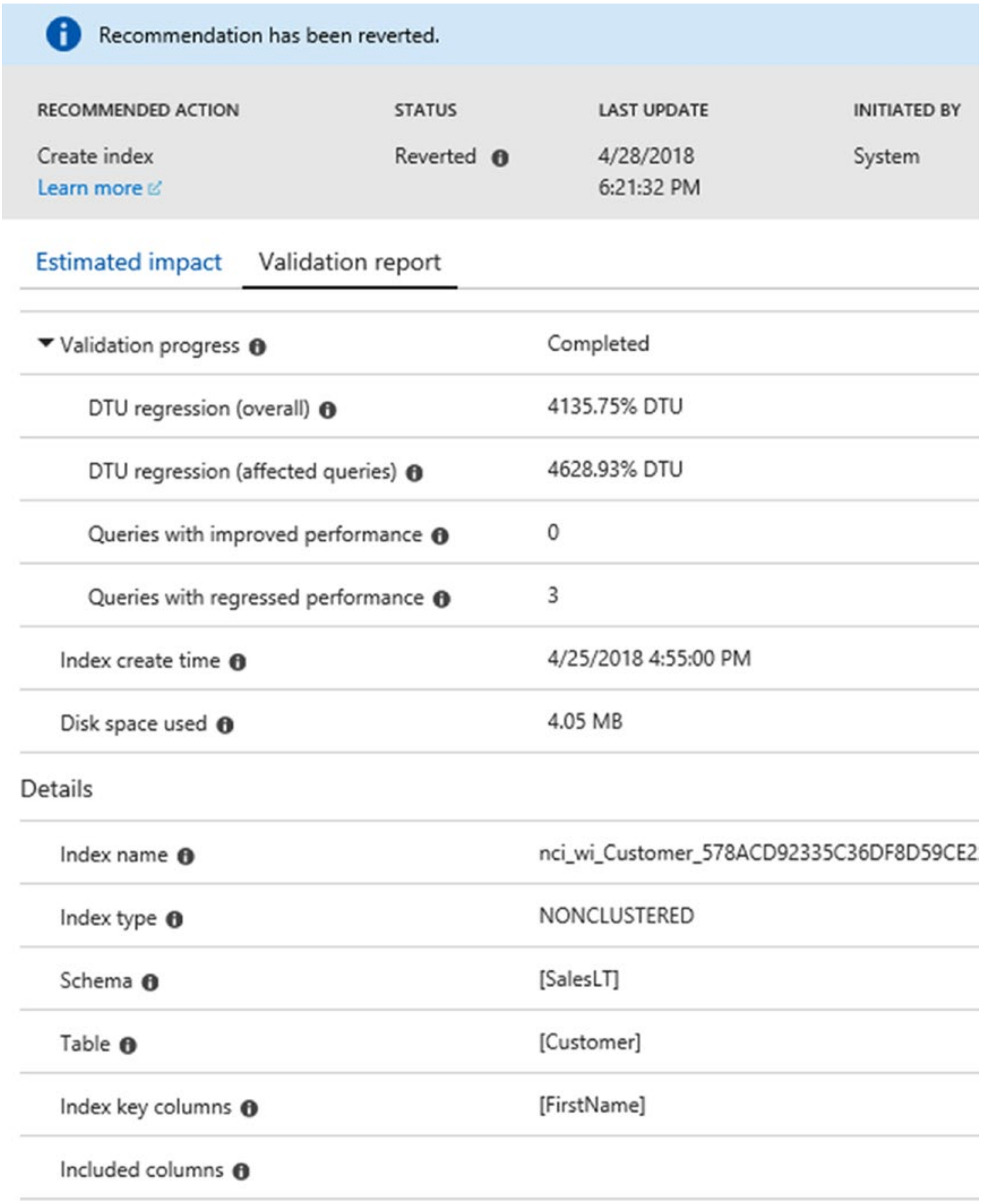
Details

Index name	nci_wi_Customer_578ACD92335C36DF8D59CE26B
Index type	NONCLUSTERED
Schema	[SalesLT]
Table	[Customer]
Index key columns	[FirstName]
Included columns	

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 rows / ~ 900 rows per batch = 11,111 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.

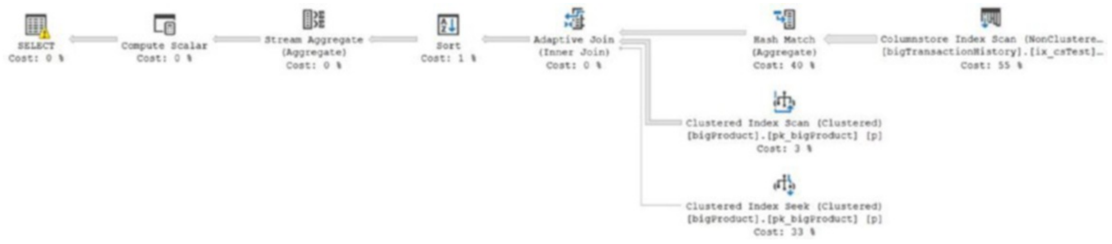
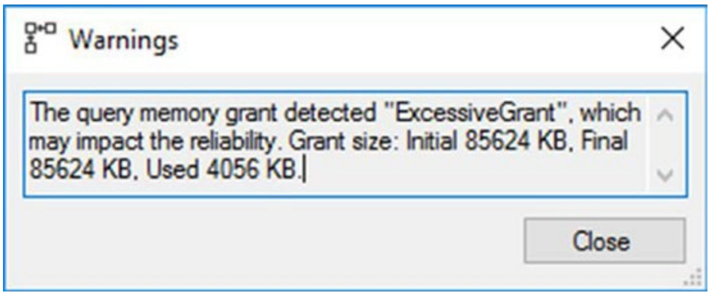

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Stream Aggregate
(Aggregate)
Cost: 0 %
Sort
Cost: 1 %
Adaptive Join
(Inner Join)
Cost: 0 %
Hash Match
(Aggregate)
Cost: 40 %
Columnstore Index Scan (NonClustere…
[bigTransactionHistory].[ix_csTest]…
Cost: 55 %
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 3 %
Clustered Index Seek (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 33 %

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.

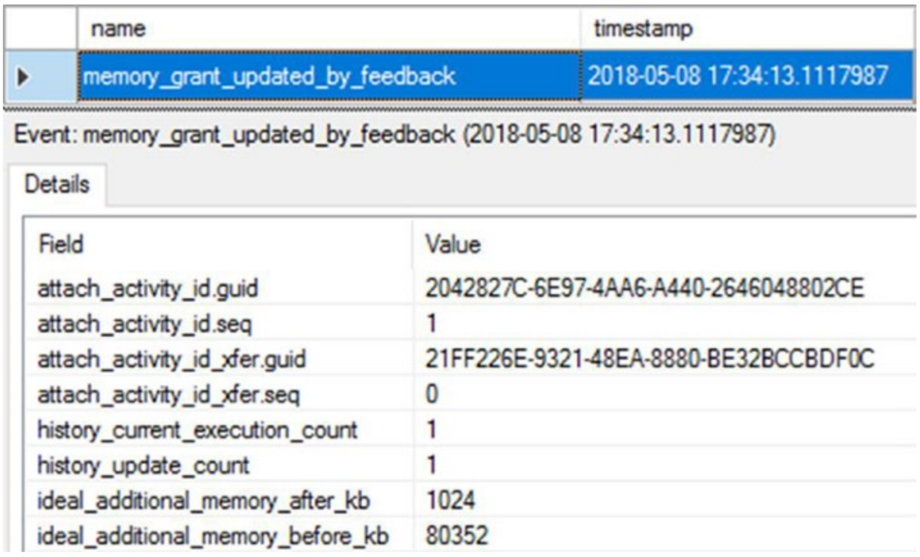


Warnings

```
The query memory grant detected "ExcessiveGrant", which
may impact the reliability. Grant size: Initial 85624 KB, Final
85624 KB, Used 4056 KB.
```

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

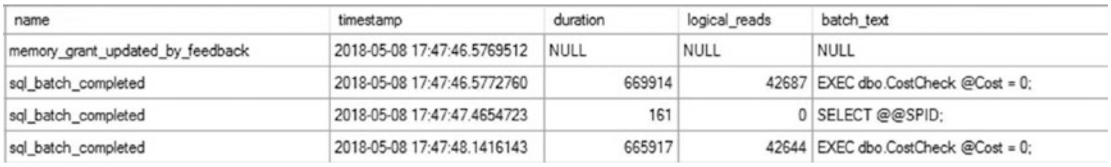
Event: memory_grant_updated_by_feedback (2018-05-08 17:34:13.1117987)

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.

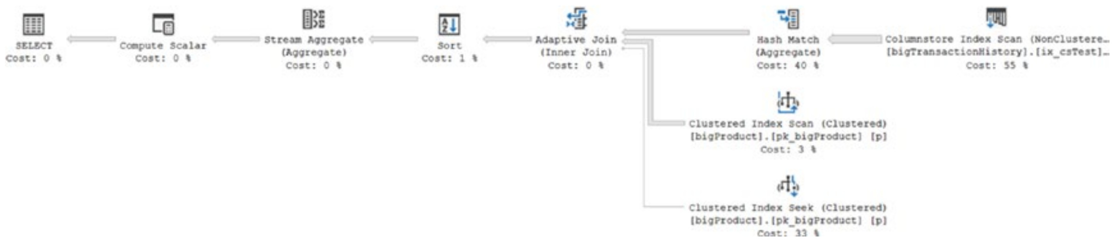

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Stream Aggregate
(Aggregate)
Cost: 0 %
Sort
Cost: 1 %
Adaptive Join
(Inner Join)
Cost: 0 %
Hash Match
(Aggregate)
Cost: 40 %
Columnstore Index Scan (NonClustere…
[bigTransactionHistory].[ix_csTest]…
Cost: 55 %
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 3 %
Clustered Index Seek (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 33 %

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.

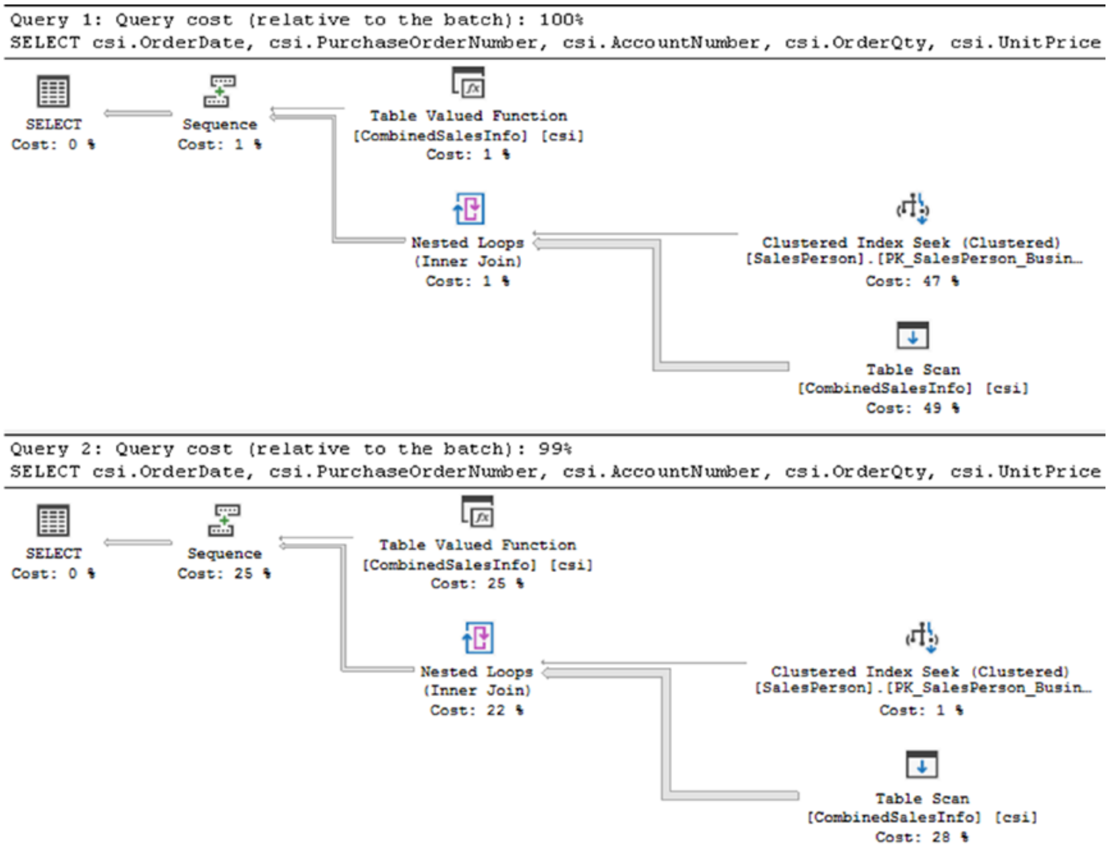
SQL Server 2014 and greater have a different assumption, 100 rows. So if the compatibility level is set to 140 or greater, you'll see the 100-row assumption, except if interleaved execution is enabled.

We can run a query against these functions. However, first we'll want to run it with interleaved execution disabled. Then we'll reenable 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.

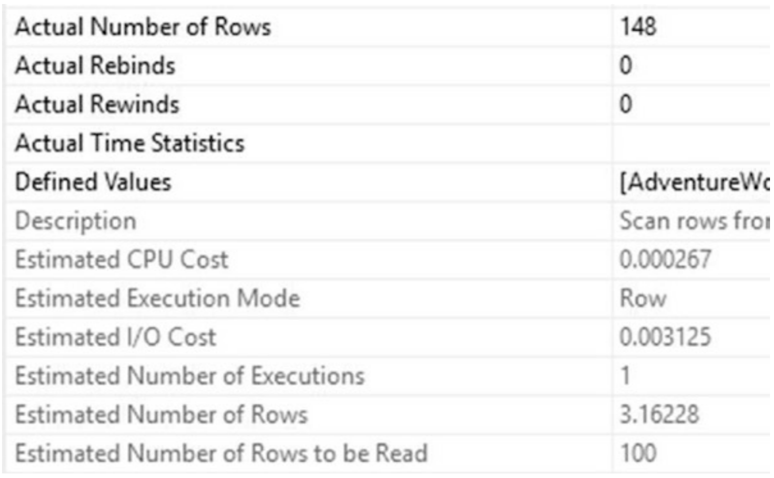

Query 1: Query cost (relative to the batch): 100%
SELECT csi.OrderDate, csi.PurchaseOrderNumber, csi.AccountNumber, csi.OrderQty, csi.UnitPrice

Query 2: Query cost (relative to the batch): 99%
SELECT csi.OrderDate, csi.PurchaseOrderNumber, csi.AccountNumber, csi.OrderQty, csi.UnitPrice

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).

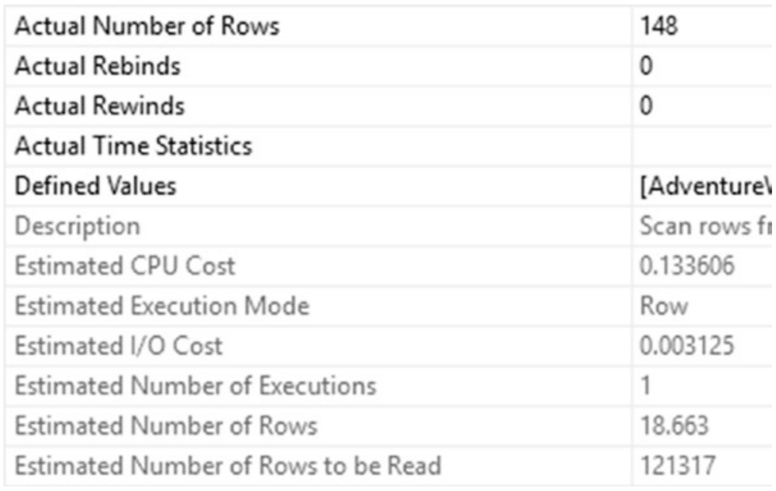


Actual Number of Rows	148
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[AdventureWc
Description	Scan rows froi
Estimated CPU Cost	0.000267
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	1
Estimated Number of Rows	3.16228
Estimated Number 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)
```

```
AS
BEGIN;

    INSERT INTO @return_variable (SalesPersonID,
                                  ShippingCity,
                                  OrderDate,
                                  PurchaseOrderNumber,
                                  AccountNumber,
                                  OrderQty,
                                  UnitPrice)
    SELECT soh.SalesPersonID,
           a.City,
           soh.OrderDate,
           soh.PurchaseOrderNumber,
           soh.AccountNumber,
           sod.OrderQty,
           sod.UnitPrice
    FROM Sales.SalesOrderHeader AS soh
        JOIN Person.Address AS a
            ON a.AddressID = soh.ShipToAddressID
        JOIN Sales.SalesOrderDetail AS sod
            ON sod.SalesOrderID = soh.SalesOrderID
    WHERE soh.SalesPersonID = @SalesPersonID
          AND a.City = @ShippingCity;
    RETURN;
END;
GO
```

Instead of using a `WHERE` clause to execute the final query, we would execute it like this:

```
SELECT asi.OrderDate,
       asi.PurchaseOrderNumber,
       asi.AccountNumber,
       asi.OrderQty,
       asi.UnitPrice,
       sp.SalesQuota
```

```
FROM dbo.AllSalesInfo(277,'Odessa') AS asi
    JOIN Sales.SalesPerson AS sp
        ON asi.SalesPersonID = sp.BusinessEntityID;
```

By passing the parameters down to the function, we allow the interleaved execution to have values to measure itself against. Doing it this way returns exactly the same data, but the performance dropped to 65ms and only 1,135 reads. That's pretty amazing for a multistatement function. However, running this as a noninterleaved function also dropped the execution time to 69ms and 1,428 reads. While we are talking about improvements requiring no code or structure changes, those improvements are very minimal.

One additional problem can arise because of the interleaved execution, especially if you pass values as I did in the second query. It's going to create a plan based on the values it has in hand. This effectively acts as if it is parameter sniffing. It's using these hard-coded values to create execution plans directly in support of them, using these values against the statistics as the row count estimates. If your statistics vary wildly, you could be looking at performance problems similar to what we talked about in Chapter 15.

You can also use a query hint to disable the interleaved execution. Simply supply `DISABLE_INTERLEAVED_EXECUTION_TVF` to the query through the hint, and it will disable it only for the query being executed.

Summary

With the addition of the tuning recommendations in SQL Server 2017, along with the index automation in Azure SQL Database, you now have a lot more help within SQL Server when it comes to automation. You'll still need to use the information you've learned in the rest of the book to understand when those suggestions are helpful and when they're simply clues to making your own choices. However, things are even easier because SQL Server can make automatic adjustments without you having to do any work at all through the adaptive query processing. Just remember that all this is helpful, but none of it is a complete solution. It just means you have more tools in your toolbox to help deal with poorly performing queries.

The next chapter discusses methods you can use to automate the testing of your queries through the use of distributed replay.

CHAPTER 26

Database Performance Testing

Knowing how to identify performance issues and knowing how to fix them are great skills to have. The problem, though, is that you need to be able to demonstrate that the improvements you make are real improvements. While you can, and should, capture the performance metrics before and after you tune a query or add an index, the best way to be sure you're looking at measurable improvement is to put the changes you make to work. Testing means more than simply running a query a few times and then putting it into your production system with your fingers crossed. You need to have a systematic way to validate performance improvements using the full panoply of queries that are run against your system in a realistic manner. Introduced with the 2012 version, SQL Server provides such a mechanism through its Distributed Replay tool.

Distributed Replay works with information generated from the SQL Profiler and the trace events created by it. Trace events capture information in a somewhat similar fashion to the Extended Events tool, but trace events are an older (and less capable) mechanism for capturing events within the system. Prior to the release of SQL Server 2012, you could use SQL Server's Profiler tool to replay captured events using a server-side trace. This worked, but the process was extremely limited. For example, the tool could be run only on a single machine, and it dealt with the playback mechanism—a single-threaded process that ran in a serial fashion, rather than what happens in reality. Microsoft has added the capability to run from multiple machines in a parallel fashion to SQL Server. Until Microsoft makes a mechanism to use Distributed Replay through Extended Events output, you'll still be using the trace events for this one aspect of your performance testing.

Distributed Replay is not a widely adopted tool. Most people just skip the idea of implementing repeatable tests entirely. Others may go with some third-party tools that provide a little more functionality. I strongly recommend you do some form of testing to

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_26

ensure your tuning efforts are resulting in positive impact on your systems that you can accurately measure.

This chapter covers the following topics:

- The concepts of database testing
- How to create a server-side trace
- Using Distributed Replay for database testing

Database Performance Testing

The general approach to database performance and load testing is pretty simple. You need to capture the calls against a production system under normal load and then be able to play that load over and over again against a test system. This enables you to directly measure the changes in performance caused by changes to your code or structures. Unfortunately, accomplishing this in the real world is not as simple as it sounds.

To start with, you can't simply capture the recording of queries. Instead, you must first ensure that you can restore your production database to a moment in time on a test system. Specifically, you need to be able to restore to exactly the point at which you start recording the transactions on the system because if you restore to any other point, you might have different data or even different structures. This will cause the playback mechanism to generate errors instead of useful information. This means, to start with, you must have a database that is in Full Recovery mode so that you can take regular full backups as well as log backups in order to restore to a specific point in time when your testing will start.

Once you establish the ability to restore to the appropriate time, you will need to configure your query capture mechanism—a server-side trace definition generated by Profiler, in this case. The playback mechanism will define exactly which events you'll need to capture. You'll want to set up your capture process so that it impacts your system as little as possible.

Next, you'll have to deal with the large amounts of data captured by the trace. Depending on how big your system is, you may have a large number of transactions over a short period of time. All that data has to be stored and managed, and there will be many files.

You can set up this process on a single machine; however, to really see the benefits, you'll want to set up multiple machines to support the playback capabilities of the Distributed Replay tool. This means you'll need to have these machines available to you as part of your testing process. Unfortunately, with all editions except Enterprise, you can have only a single client, so take that into account as you set up your test environment.

Also, you can't ignore the fact that the best data, database, and code to work with is your production system. However, depending on your need for compliance for local and international law, you may have to choose a completely different mechanism for recording your server-side trace. You don't want to compromise the privacy and protection of the data under management within the organization. If this is the case, you may have to capture your load from a QA server or a preproduction server that is used for other types of automated testing. These can be difficult problems to overcome.

When you have all these various parts in place, you can begin testing. Of course, this leads to a new question: what exactly are you doing with your database testing?

A Repeatable Process

As explained in Chapter 1, performance tuning your system is an iterative process that you may have to go through on multiple occasions to get your performance to where you need it to be and keep it there. Since businesses change over time, so will your data distribution, your applications, your data structures, and all the code supporting it. Because of all this, one of the most important things you can do for testing is to create a process that you can run over and over again.

The primary reason you need to create a repeatable testing process is because you can't always be sure that the methods outlined in the preceding chapters of this book will work well in every situation. This no doubt means you need to be able to validate that the changes you have made have resulted in a positive improvement in performance. If not, you need to be able to remove any changes you've made, make a new set of changes, and then repeat the tests, repeating this process iteratively. You may find that you'll need to repeat the entire tuning cycle until you've met your goals for this round.

Because of the iterative nature of this process, you absolutely need to concentrate on automating it as much as possible. This is where the Distributed Replay tool comes into the picture.

Distributed Replay

The Distributed Replay tool consists of three pieces of architecture.

- *Distributed Replay Controller*: This service manages the processes of the Distributed Replay system.
- *Distributed Replay Administrator*: This is an interface to allow you to control the Distributed Replay Controller and the Distributed Replace process.
- *Distributed Replay Client*: This is an interface that runs on one or more machines (up to 16) to make all the calls to your database server.

You can install all three components onto one machine; however, the ideal approach is to have the controller on one machine and then have one or more client machines that are completely separated from the controller so that each of these machines is handling only some of the transactions you'll be making against the test machine. Only for the purposes of illustration, I have all the components running on a single instance.

Begin by installing the Distributed Replay Controller service onto a machine. There is no interface for the Distributed Replay utility. Instead, you'll use XML configuration files to take control of the different parts of the Distributed Replay architecture. You can use the distributed playback for various tasks, such as basic query playback, server-side cursors, or prepared server statements. Since I'm primarily covering query tuning, I'm focus on the queries and prepared server statements (also known as *parameterized queries*). This defines a particular set of events that must be captured. I'll cover how to do that in the next section.

Once the information is captured in a trace file, you will have to run that file through the preprocess event using the Distributed Replay Controller. This modifies the basic trace data into a different format that can be used to distribute to the various Distributed Replay Client machines. You can then fire off a replay process. The reformatted data is sent to the clients, which in turn will create queries to run against the target server. You can capture another trace output from the client machines to see exactly which calls they made, as well as the I/O and CPU of those calls. Presumably you'll also set up standard monitoring on the target server to see how the load you are generating impacts that server.

When you go to run the system against your server, you can choose one of two types of playback: Synchronization mode or Stress mode. In Synchronization mode, you will get an exact copy of the original playback, although you can affect the amount of idle time on the system. This is good for precise performance tuning because it helps you understand how the system is working, especially if you're making changes to structures, indexes, or T-SQL code. Stress mode doesn't run in any particular order, except within a single connection, where queries will be streamed in the correct order. In this case, the calls are made as fast as the client machines can make them—in any order—as fast as the server can receive them. In short, it performs a stress test. This is useful for testing database designs or hardware installations.

One important note, as a general rule, is that you're safest when using the latest version of SQL Server for your replay only with the latest version of trace data. However, you can replay SQL Server 2005 data on SQL Server 2017. Also, Azure SQL Database is not supported by Distributed Replay or trace events, so you won't be able to use any of this with your Azure database.

Capturing Data with the Server-Side Trace

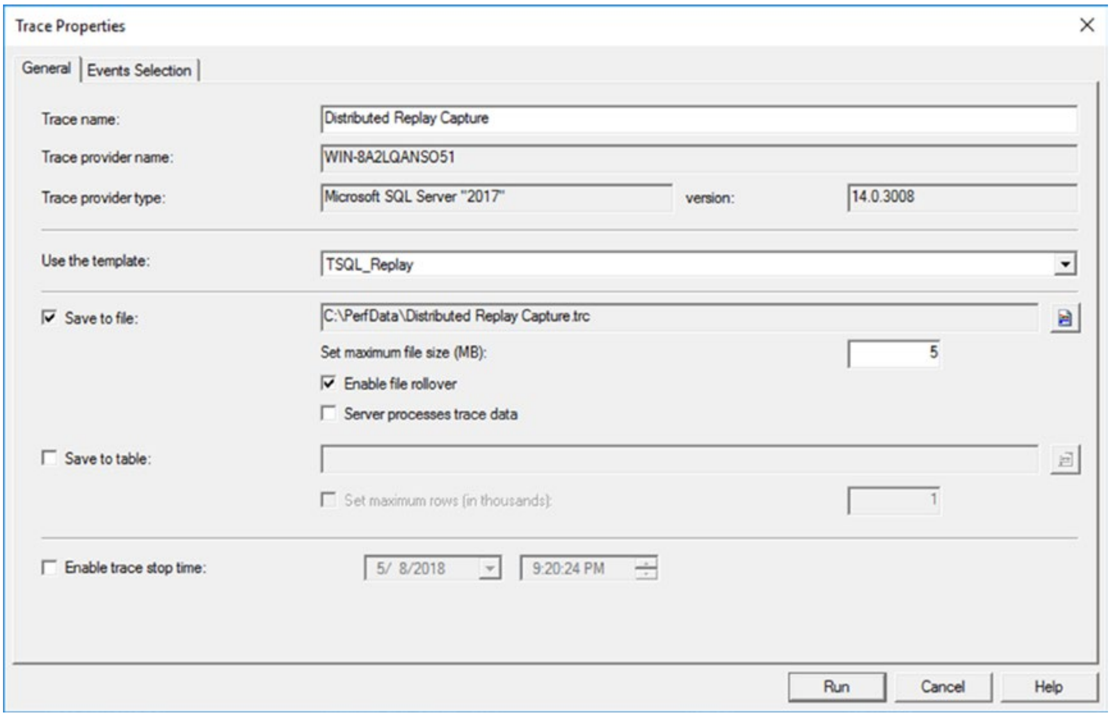
Using trace events to capture data is similar to capturing query executions with Extended Events. To support the Distributed Replay process, you'll need to capture some specific events and specific columns for those events. If you want to build your own trace events, you need to go after the events listed in Table 26-1.

Table 26-1. *Events to Capture*

Events	Columns
`Prepare SQL`	`Event Class`
`Exec Prepared SQL`	`EventSequence`
`SQL:BatchStarting`	`TextData`
`SQL:BatchCompleted`	`Application Name`
`RPC:Starting`	`LoginName`
`RPC:Completed`	`DatabaseName`
`RPC Output Parameter`	`Database ID`
`Audit Login`	`HostName`
`Audit Logout`	`Binary Data`
`Existing Connection`	`SPID`
`Server-side Cursor`	`Start Time`
`Server-side prepared SQL`	`EndTime`
	`IsSystem`

You have two options for setting up these events. First, you can use T-SQL to set up a server-side trace. Second, you can use an external tool called Profiler. While Profiler can connect directly to your SQL Server instance, I strongly recommend against using this tool to capture data. Profiler is best used as a way to supply a template for performing the capture. You should use T-SQL to generate the actual server-side trace.

On a test or development machine, open Profiler and select TSQL_Replay from the Template list, as shown in Figure 26-1.

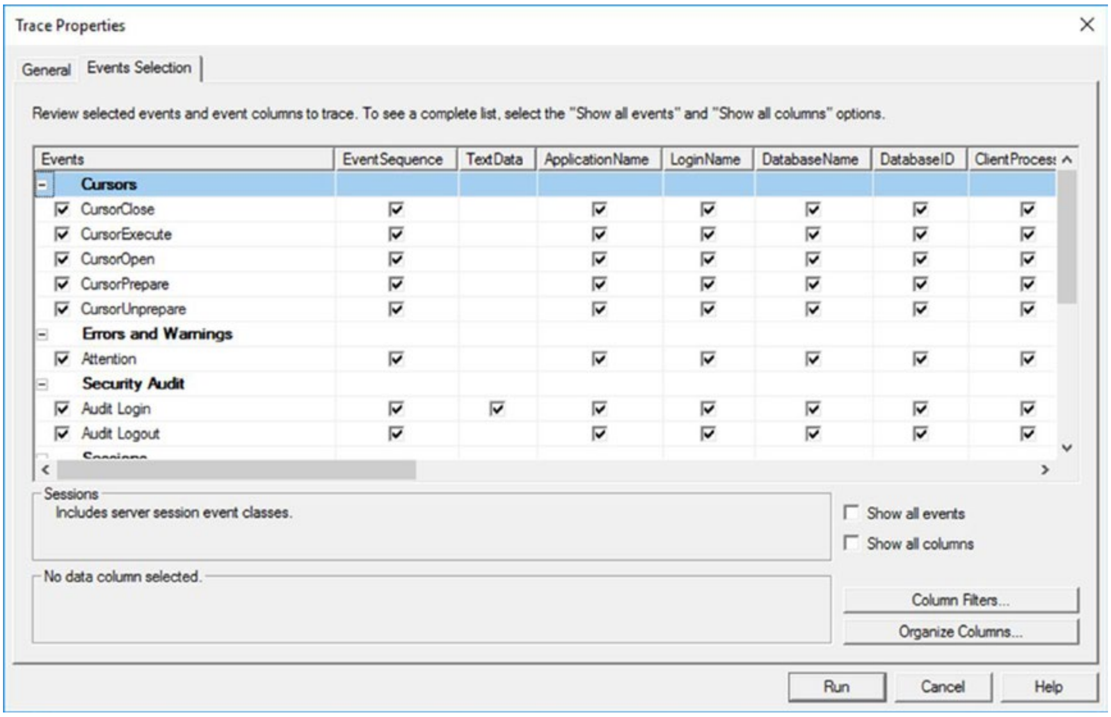


Trace name: Distributed Replay Capture
Trace provider name: WIN-8A2LQANSO51
Trace provider type: Microsoft SQL Server "2017" version: 14.0.3008
Use the template: TSQL_Replay
☑ Save to file: C:\PerfData\Distributed Replay Capture.trc
Set maximum file size (MB): 5
☑ Enable file rollover
☐ Server processes trace data
☐ Save to table:
☐ Set maximum rows (in thousands): 1
☐ Enable trace stop time: 5/ 8/2018 9:20:24 PM

Figure 26-1. *The Distributed Replay trace template*

Since you need a file for Distributed Replay, you'll want to save the output of the trace to file. It's the best way to set up a server-side trace anyway, so this works out. You'll want to output to a location that has sufficient space. Depending on the number of transactions you have to support with your system, trace files can be extremely large. Also, it's a good idea to put a limit on the size of the files and allow them to roll over, creating new files as needed. You'll have more files to deal with, but the operating system can actually deal with a larger number of smaller files for writes better than it can deal with a single large file. I've found this to be true because of two things. First, with a smaller file size, you get a quicker rollover, which means the previous file is available for processing if you need to load it into a table or copy it to another server. Second, in my experience, it generally takes longer for writes to occur with simple log files because the size of such files gets very large. I also suggest defining a stop time for the trace process; again, this helps ensure you don't fill the drive you've designated for storing the trace data.

Since this is a template, the events and columns have already been selected for you. You can validate the events and columns to ensure you are getting exactly what you need by clicking the Events Selection tab. Figure 26-2 shows some of the events and columns, all of which are predefined for you.



Trace Properties

General | Events Selection

Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	EventSequence	TextData	ApplicationName	LoginName	DatabaseName	DatabaseID	ClientProces
Cursors							
✓ CursorClose	✓		✓	✓	✓	✓	✓
✓ CursorExecute	✓		✓	✓	✓	✓	✓
✓ CursorOpen	✓		✓	✓	✓	✓	✓
✓ CursorPrepare	✓		✓	✓	✓	✓	✓
✓ CursorUnprepare	✓		✓	✓	✓	✓	✓
Errors and Warnings							
✓ Attention	✓		✓	✓	✓	✓	✓
Security Audit							
✓ Audit Login	✓	✓	✓	✓	✓	✓	✓
✓ Audit Logout	✓		✓	✓	✓	✓	✓

Sessions
Includes server session event classes.

☐ Show all events
☐ Show all columns

No data column selected.

***Figure 26-2.** The TSQL_Replay template events and columns*

This template is generic, so it includes the full list of events, including all the cursor events. You can edit it by clicking boxes to deselect events; however, I do not recommend removing anything other than the cursor events, if you're going to remove any.

I started this template connected to a test server instead of a production machine because once you've set it up appropriately, you have to start the trace by clicking Run. I wouldn't do that on a production system. On a test system, however, you can watch the screen to ensure you're getting the events you think you should. It will display the events, as well as capture them to a file. When you're satisfied that it's correct, you can pause the trace. Next, click the File menu and then select Export ➤ Script Trace Definition. Finally, select For SQL Server 2005 – 2014 (see Figure 26-3).

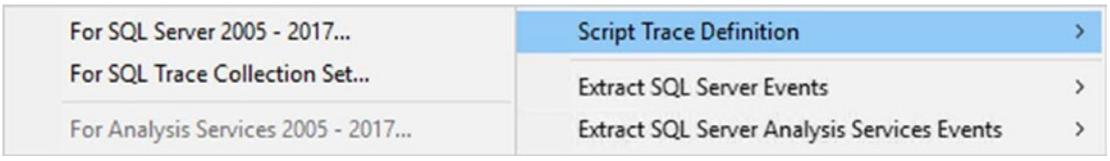

For SQL Server 2005 - 2017...
For SQL Trace Collection Set...
For Analysis Services 2005 - 2017...
Script Trace Definition
Extract SQL Server Events
Extract SQL Server Analysis Services Events

Figure 26-3. *The menu selection to output the trace definition*

This template will allow you to save the trace you just created as a T-SQL file. Once you have the T-SQL, you can configure it to run on any server that you like. The file path will have to be replaced, and you can reset the stop time through parameters within the script. The following script shows the beginning of the T-SQL process used to set up the server-side trace events:

```
/****************************************************/
/* Created by: SQL Server 2017 Profiler          */
/* Date: 05/08/2018  08:27:40 PM         */
/****************************************************/

-- Create a Queue
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

-- Please replace the text InsertFileNameHere, with an appropriate
-- filename prefixed by a path, e.g., c:\MyFolder\MyTrace. The .trc extension
-- will be appended to the filename automatically. If you are writing from
-- remote server to local drive, please use UNC path and make sure server has
-- write access to your network share

exec @rc = sp_trace_create @TraceID output, 0, N'InsertFileNameHere',
@maxfilesize, NULL
if (@rc != 0) goto error
```

You can edit the path where it says `InsertFileNameHere` and provide different values for `@DateTime`. At this point, your script can be run on any SQL Server 2017 server. You can probably run the same script all the way back to SQL Server 2008R2; there have been so few changes made to trace events since then that this is a fixed standard now. However, always test to be on the safe side.

The amount of information you collect really depends on what kind of test you want to run. For a standard performance test, it's probably a good idea to collect at least one hour's worth of information; however, you wouldn't want to capture more than two to three hours of data in most circumstances. Plus, it can't be emphasized enough that trace events are not as lightweight as extended events, so the longer you capture data, the more you're negatively impacting your production server. Capturing more than that would entail managing a lot more data, and it would mean you were planning on running your tests for a long time. It all depends on the business and application behaviors you intend to deal with in your testing.

Before you capture the data, you do need to think about where you're going to run your tests. Let's assume you're not worried about disk space and that you don't need to protect legally audited data (if you have those issues, you'll need to address them separately). If your database is not in Full Recovery mode, then you can't use the log backups to restore it to a point in time. If this is the case, I strongly recommend running a database backup as part of starting the trace data collection. The reason for this is that you need the database to be in the same condition it's in when you start recording transactions. If it's not, you may get a larger number of errors, which could seriously change the way your performance tests run. For example, attempting to select or modify data that doesn't exist will impact the I/O and CPU measured in your tests. If your database remains in the same state that it was at or near the beginning of your trace, then you should few, if any, errors.

With a copy of the database ready to go and a set of trace data, you're ready to run the Distributed Replay tool.

Distributed Replay for Database Testing

Assuming you used the Distributed Replay template to capture your trace information, you should be ready to start processing the files. As noted earlier, the first step is to convert the trace file into a different format, one that can be split up among multiple client machines for playback. But there is more to it than simply running the executable against your file. You also need to make some decisions about how you want the Distributed Replay to run; you make those decisions when you preprocess the trace file.

The decisions are fairly straightforward. First, you need to decide whether you're going to replay system processes along with the user processes. Unless you're dealing with the potential of specific system issues, I suggest setting this value to `No`. This is also the default value. Second, you need to decide how you want to deal with idle time. You can use the actual values for how often calls were made to the database; or, you can put in a value, measured in seconds, to limit the wait time to no more than that value. It really depends on what type of playback you're going to run. Assuming you use Synchronization mode playback, the mode best suited for straight performance measurement, it's a good idea to eliminate idle time by setting the value to something low, such as three to five seconds.

If you choose to use the default values, you don't need to modify the configuration file. But if you've chosen to include the system calls or to change the idle time, then you'll need to change the configuration file, `DReplay.Exe.Preprocess.config`. It's a simple XML configuration file. The one I'm using looks like this:

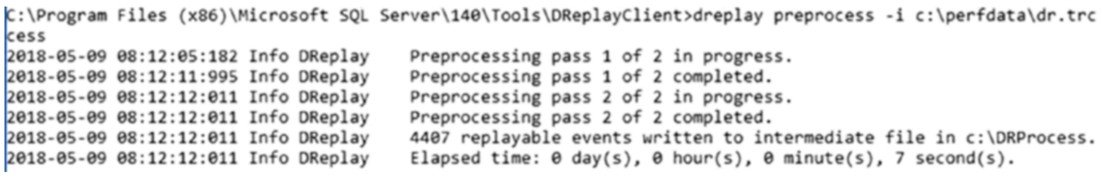
```
<?xml version="1.0" encoding="utf-8"?>
<Options>
<PreprocessModifiers>
<IncSystemSession>No</IncSystemSession>
<MaxIdleTime>2</MaxIdleTime>
</PreprocessModifiers>
</Options>
```

I've made only one change, adjusting `MaxdIdleTime` to limit any down period during the playback.

Before you run the preprocessing, make sure have installed the `DRController` and that the `DReplay` service is running on your system. If so, you'll just need to call `DReplay.exe` to execute the preprocessing.

```
dreplay preprocess -i c:\perfdata\dr.trc -d c:\DRProcess
```

In the preceding code, you can see that `DReplay` runs the preprocess event. The input file was supplied by the `-i` parameter, and a folder to hold the output was supplied through the `-d` parameter. The trace files will be processed, and the output will go to the folder specified. The output will look something like Figure 26-4.



```
C:\Program Files (x86)\Microsoft SQL Server\140\Tools\DReplayClient>dreplay preprocess -i c:\perfdata\dr.trc
cess
2018-05-09 08:12:05:182 Info DReplay    Preprocessing pass 1 of 2 in progress.
2018-05-09 08:12:11:995 Info DReplay    Preprocessing pass 1 of 2 completed.
2018-05-09 08:12:12:011 Info DReplay    Preprocessing pass 2 of 2 in progress.
2018-05-09 08:12:12:011 Info DReplay    Preprocessing pass 2 of 2 completed.
2018-05-09 08:12:12:011 Info DReplay    4407 replayable events written to intermediate file in c:\DRProcess.
2018-05-09 08:12:12:011 Info DReplay    Elapsed time: 0 day(s), 0 hour(s), 0 minute(s), 7 second(s).
```

Figure 26-4. *Output from the preprocessing steps of Distributed Replay*

With the preprocessing complete, you're ready to move ahead with running the Distributed Replay process. Before you do so, however, you need to make sure you have one or more client systems ready to go.

Configuring the Client

The client machines will have to be configured to work with the Distributed Replay controller. Begin by installing your clients to the different machines. For illustration purposes only, I'm running everything on a single machine; however, the setup is no different if you use multiple machines. You need to configure the client to work with a controller, and a client can work with only one controller at a time. You also need to have space on the system for two items. First, you need a location for working files that are overwritten at each replay. Second, you need room for trace file output from the client if you want to collect execution data from that client. You also get to decide on the logging level of the client process. All of this is set in another XML configuration file, `DReplayClient.config`. Here is my configuration:

```
<Options>
<Controller>PerfTune</Controller>
<WorkingDirectory>C:\DRClientWork\</WorkingDirectory>
<ResultDirectory>C:\DRClientOutput\</ResultDirectory>
<LoggingLevel>CRITICAL</LoggingLevel>
</Options>
```

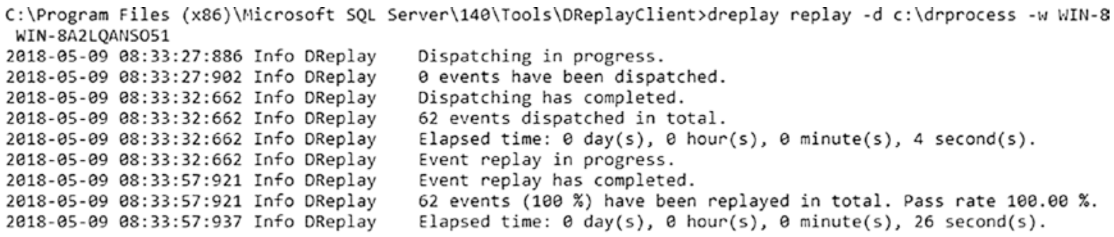
The directories and logging level are clear. I also had to point the client to the server where the Distributed Replay service is running. No other settings are required for multiple clients to work; you just need to be sure they're going to the right controller system.

Running the Distributed Tests

So far you have configured everything and captured the data. Next, you need to go back to the command line to run things from the `Dreplay.exe` executable. Most of the control is accomplished through the configuration files, so there is little input required in the executable. You invoke the tests using the following command:

```
Dreplay replay -d c:\data -w DOJO
```

You need to feed in the location of the output from the preprocessing, which means you need to list the client machines that are taking part in a comma-delimited list. The output from the execution would look something like Figure 26-5.



```
C:\Program Files (x86)\Microsoft SQL Server\140\Tools\DReplayClient>dreplay replay -d c:\drprocess -w WIN-8
 WIN-8A2LQANSO51
2018-05-09 08:33:27:886 Info DReplay    Dispatching in progress.
2018-05-09 08:33:27:902 Info DReplay    0 events have been dispatched.
2018-05-09 08:33:32:662 Info DReplay    Dispatching has completed.
2018-05-09 08:33:32:662 Info DReplay    62 events dispatched in total.
2018-05-09 08:33:32:662 Info DReplay    Elapsed time: 0 day(s), 0 hour(s), 0 minute(s), 4 second(s).
2018-05-09 08:33:32:662 Info DReplay    Event replay in progress.
2018-05-09 08:33:57:921 Info DReplay    Event replay has completed.
2018-05-09 08:33:57:921 Info DReplay    62 events (100 %) have been replayed in total. Pass rate 100.00 %.
2018-05-09 08:33:57:937 Info DReplay    Elapsed time: 0 day(s), 0 hour(s), 0 minute(s), 26 second(s).
```

Figure 26-5. *The output from running DReplay.exe*

As you can see, 62 events were captured, and all 62 events were successfully replayed. If, on the other hand, you had errors or events that failed, you might need to establish what information might exist about why some of the events failed. This information is available in the logs. Then, simply reconfigure the tests and run them again. The whole idea behind having a repeatable testing process is that you can run it over and over. The preceding example represents a light load run against my local copy of AdventureWorks2017, captured over about five minutes. However, I configured the limits on idle time, so the replay completes in only 26 seconds.

From here, you can reconfigure the tests, reset the database, and run the tests over and over again, as needed. Note that changing the configuration files will require you to restart the associated services to ensure that the changes are implemented with the next set of tests. One of the best ways to deal with testing here is to have the Query Store enabled. You can capture one set of results, reset the test, make whatever changes to the system you're going to make, and then capture another set of results from a second test. Then, you can easily look at reports for regressed queries, queries that used the most resources, and so on.

Conclusion

With the inclusion of the Distributed Replay utilities, SQL Server now gives you the ability to perform load and function testing against your databases. You accomplish this by capturing your code in a simple manner with a server-side trace. If you plan to take advantage of this feature, however, be sure to validate that the changes you make to queries based on the principles put forward in this book actually work and will help improve the performance of your system. You should also make sure you reset the database to avoid errors as much as possible.

CHAPTER 27

Database Workload Optimization

So far, you have learned about a number of aspects that can affect query performance, the tools that you can use to analyze query performance, and the optimization techniques you can use to improve query performance. Next, you will learn how to apply this information to analyze, troubleshoot, and optimize the performance of a database workload. I'll walk you through a tuning process, including possibly going down a bad path or two, so bear with me as we navigate the process.

In this chapter, I cover the following topics:

- The characteristics of a database workload
- The steps involved in database workload optimization
- How to identify costly queries in the workload
- How to measure the baseline resource use and performance of costly queries
- How to analyze factors that affect the performance of costly queries
- How to apply techniques to optimize costly queries
- How to analyze the effects of query optimization on the overall workload

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_27

Workload Optimization Fundamentals

Optimizing a database workload often fits the 80/20 rule: 80 percent of the workload consumes about 20 percent of server resources. Trying to optimize the performance of the majority of the workload is usually not very productive. So, the first step in workload optimization is to find the 20 percent of the workload that consumes 80 percent of the server resources.

Optimizing the workload requires a set of tools to measure the resource consumption and response time of the different parts of the workload. As you saw in Chapters 4 and 5, SQL Server provides a set of tools and utilities to analyze the performance of a database workload and individual queries.

In addition to using these tools, it is important to know how you can use different techniques to optimize a workload. The most important aspect of workload optimization to remember is that not every optimization technique is guaranteed to work on every performance problem. Many optimization techniques are specific to certain database application designs and database environments. Therefore, for each optimization technique, you need to measure the performance of each part of the workload (that is, each individual query) before and after you apply an optimization technique. You can use the techniques discussed in Chapter 26 to make this happen.

It is not unusual to find that an optimization technique has little effect—or even a negative effect—on the other parts of the workload, thereby hurting the overall performance of the workload. For instance, a nonclustered index added to optimize a `SELECT` statement can hurt the performance of `UPDATE` statements that modify the value of the indexed column. The `UPDATE` statements have to update index rows in addition to the data rows. However, as demonstrated in Chapter 6, sometimes indexes can improve the performance of action queries, too. Therefore, improving the performance of a particular query could benefit or hurt the performance of the overall workload. As usual, your best course of action is to validate any assumptions through testing.

Workload Optimization Steps

The process of optimizing a database workload follows a specific series of steps. As part of this process, you will use the set of optimization techniques presented in previous chapters. Since every performance problem is a new challenge, you can use a different set of optimization techniques for troubleshooting different performance problems. Just remember that the first step is always to ensure that the server is well configured and operating within acceptable limits, as defined in Chapters 2 and 3.

To understand the query optimization process, you will simulate a sample workload using a set of queries.

The core of query tuning comes down to just a few steps.

1. Identify the query to tune.
2. Look at the execution plan to understand resource usage and behavior.
3. Modify the query or modify the structure to improve performance.

Most of the time, the answer is, modify the query. In a nutshell, that's all that's necessary to do query tuning. However, this assumes a lot of knowledge of the system, and you've looked at things like statistics in the past. When you're approaching query tuning for the first time or you're on a new system, the process is quite a bit more detailed. For a thorough and complete definition of the steps necessary to tune a query, here's what you're going to do. These are the optimization steps you will follow as you optimize the sample workload:

1. Capture the workload.
2. Analyze the workload.
3. Identify the costliest/most frequently called/longest-running query.
4. Quantify the baseline resource use of the costliest query.
5. Determine the overall resource use.
6. Compile detailed information on resource use.
7. Analyze and optimize external factors.
8. Analyze the use of indexes.
9. Analyze the batch-level options used by the application.
10. Analyze the effectiveness of statistics.
11. Assess the need for defragmentation.
12. Analyze the internal behavior of the costliest query.

13. Analyze the query execution plan.
14. Identify the costly operators in the execution plan.
15. Analyze the effectiveness of the processing strategy.
16. Optimize the costliest query.
17. Analyze the effects of the changes on database workload.
18. Iterate through multiple optimization phases.

As explained in Chapter 1, performance tuning is an iterative process. Therefore, you should iterate through the performance optimization steps multiple times until you achieve the desired application performance targets. After a certain period of time, you will need to repeat the process to address the impact on the workload caused by data and database changes. Further, as you find yourself working on a server over time, you may be skipping lots of the previous steps since you've already validated transaction methods or statistics maintenance or other steps. You don't have to follow this slavishly. It's meant to be a guide. I'll refer you to Chapter 1 for the graphical representation of the steps needed to tune a query.

Sample Workload

To troubleshoot SQL Server performance, you need to know the SQL workload that is executed on the server. You can then analyze the workload to identify causes of poor performance and applicable optimization steps. Ideally, you should capture the workload on the SQL Server facing the performance problems. In this chapter, you will use a set of queries to simulate a sample workload so that you can follow the optimization steps listed in the previous section. The sample workload you'll use consists of a combination of good and bad queries.

Note I recommend you restore a clean copy of the AdventureWorks2017 database so that any artifacts left over from previous chapters are completely removed.

The simple test workload is simulated by the following set of sample stored procedures; you execute them using the second script on the AdventureWorks2017 database:

```
USE AdventureWorks2017;
GO

CREATE OR ALTER PROCEDURE dbo.ShoppingCart @ShoppingCartId VARCHAR(50)
AS
--provides the output from the shopping cart including the line total
SELECT sci.Quantity,
       p.ListPrice,
       p.ListPrice * sci.Quantity AS LineTotal,
       p.Name
FROM Sales.ShoppingCartItem AS sci
    JOIN Production.Product AS p
        ON sci.ProductID = p.ProductID
WHERE sci.ShoppingCartID = @ShoppingCartId;
GO

CREATE OR ALTER PROCEDURE dbo.ProductBySalesOrder @SalesOrderID INT
AS
/*provides a list of products from a particular sales order,
and provides line ordering by modified date but ordered
by product name*/

SELECT ROW_NUMBER() OVER (ORDER BY sod.ModifiedDate) AS LineNumber,
       p.Name,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
WHERE soh.SalesOrderID = @SalesOrderID
ORDER BY p.Name ASC;
GO
```

```
CREATE OR ALTER PROCEDURE dbo.PersonByFirstName @FirstName NVARCHAR(50)
AS
--gets anyone by first name from the Person table
SELECT p.BusinessEntityID,
       p.Title,
       p.LastName,
       p.FirstName,
       p.PersonType
FROM Person.Person AS p
WHERE p.FirstName = @FirstName;
GO

CREATE OR ALTER PROCEDURE dbo.ProductTransactionsSinceDate
    @LatestDate DATETIME,
    @ProductName NVARCHAR(50)
AS
--Gets the latest transaction against
--all products that have a transaction
SELECT p.Name,
       th.ReferenceOrderID,
       th.ReferenceOrderLineID,
       th.TransactionType,
       th.Quantity
FROM Production.Product AS p
    JOIN Production.TransactionHistory AS th
        ON p.ProductID = th.ProductID
           AND th.TransactionID = (   SELECT TOP (1)
                                             th2.TransactionID
                                      FROM Production.TransactionHistory AS
th2
                                      WHERE th2.ProductID = p.ProductID
                                      ORDER BY th2.TransactionID DESC)
WHERE th.TransactionDate > @LatestDate
      AND p.Name LIKE @ProductName;
GO
```

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderBySalesPersonName
    @LastName NVARCHAR(50),
    @VendorID INT = NULL
AS
SELECT poh.PurchaseOrderID,
       poh.OrderDate,
       pod.LineTotal,
       p.Name AS ProductName,
       e.JobTitle,
       per.LastName + ', ' + per.FirstName AS SalesPerson,
       poh.VendorID
FROM Purchasing.PurchaseOrderHeader AS poh
    JOIN Purchasing.PurchaseOrderDetail AS pod
        ON poh.PurchaseOrderID = pod.PurchaseOrderID
    JOIN Production.Product AS p
        ON pod.ProductID = p.ProductID
    JOIN HumanResources.Employee AS e
        ON poh.EmployeeID = e.BusinessEntityID
    JOIN Person.Person AS per
        ON e.BusinessEntityID = per.BusinessEntityID
WHERE per.LastName LIKE @LastName
      AND poh.VendorID = COALESCE(@VendorID,
                                  poh.VendorID)
ORDER BY per.LastName,
         per.FirstName;
GO

CREATE OR ALTER PROCEDURE dbo.TotalSalesByProduct @ProductID INT
AS
--retrieve aggregation of sales based on a productid
SELECT SUM((isnull((sod.UnitPrice*((1.0)-sod.UnitPriceDiscount))*sod.
OrderQty,(0.0)))) AS TotalSales,
    AVG(sod.OrderQty) AS AverageQty,
    AVG(sod.UnitPrice) AS AverageUnitPrice,
    SUM(sod.LineTotal)
FROM Sales.SalesOrderDetail AS sod
```

```
WHERE sod.ProductID = @ProductID
GROUP BY sod.ProductID;
GO
```

Please remember that this is just meant to be an illustrative example, not a literal and real load placed on a server. Real procedures are generally much more complex, but there's only so much space we can devote to setting up a simulated production load. With these procedures in place, you can execute them using the following script:

```
EXEC dbo.PurchaseOrderBySalesPersonName @LastName = 'Hill%';
GO
EXEC dbo.ShoppingCart @ShoppingCartId = '20621';
GO
EXEC dbo.ProductBySalesOrder @SalesOrderID = 43867;
GO
EXEC dbo.PersonByFirstName @FirstName = 'Gretchen';
GO
EXEC dbo.ProductTransactionsSinceDate @LatestDate = '9/1/2004',
                                      @ProductName = 'Hex Nut%';
GO
EXEC dbo.PurchaseOrderBySalesPersonName @LastName = 'Hill%',
                                        @VendorID = 1496;
GO
EXEC dbo.TotalSalesByProduct @ProductID = 707;
GO
```

I know I'm repeating myself, but I want to be clear. This is an extremely simplistic workload that just illustrates the process. You're going to see hundreds and thousands of additional calls across a much wider set of procedures and ad hoc queries in a typical system. As simple as it is, however, this sample workload consists of the different types of queries you usually execute on SQL Server.

- Queries using aggregate functions
- Point queries that retrieve only one row or a small number of rows
- Queries joining multiple tables

- Queries retrieving a narrow range of rows
- Queries performing additional result set processing, such as providing a sorted output

The first optimization step is to capture the workload, meaning see how these queries are performing, as explained in the next section.

Capturing the Workload

As part of the diagnostic-data collection step, you must define an Extended Events session to capture the workload on the database server. You can use the tools and methods recommended in Chapter 6 to do this. Table 27-1 lists the specific events you can use to measure how many resources your queries use.

Table 27-1. *Events to Capture Information About Costly Queries*

Category	Event
Execution	`rpc_completed` `sql_batch_completed`

As explained in Chapter 6, for production databases it is recommended that you capture the output of the Extended Events session to a file. Here are a couple significant advantages to capturing output to a file:

- Since you intend to analyze the SQL queries once the workload is captured, you do not need to display the SQL queries while capturing them.
- Running the session through SSMS doesn't provide a flexible timing control over the tracing process.

Let's look at the timing control more closely. Assume you want to start capturing events at 11 p.m. and record the SQL workload for 24 hours. You can define an Extended Events session using the GUI or T-SQL. However, you don't have to start the process until you're ready. This means you can create commands in SQL Agent or with some other scheduling tool to start and stop the process with the `ALTER EVENT SESSION` command.

```
ALTER EVENT SESSION <sessionname>
ON SERVER
STATE = <start/stop>;
```

For this example, I've put a filter on the session to capture events only from the AdventureWorks2017 database. The file will capture queries against only that database, reducing the amount of information I need to deal with. This may be a good choice for your systems, too. While Extended Events sessions can be very low cost, especially when compared to the older trace events, they are not free. Good filtering should always be applied to ensure minimum impact.

Analyzing the Workload

Once the workload is captured in a file, you can analyze the workload either by browsing through the data using SSMS or by importing the content of the output file into a database table.

SSMS provides the following two methods for analyzing the content of the file, both of which are relatively straightforward:

- *Sort the output on a data column by right-clicking to select a sort order or to group by a particular column*: You may want to select columns from the Details tab and use the "Show column in table" command to move them up. Once there, you can issue grouping and sorting commands on that column.
- *Rearrange the output to a selective list of columns and events*: You can change the output displayed through SSMS by right-clicking the table and selecting Pick Columns from the context menu. This lets you do more than simply pick and choose columns; it also lets you combine them into new columns.

As I've shown throughout the book, the Live Data Explorer within SSMS when used with Extended Events can be used to put together basic aggregations. For example, if you wanted to group by the text of queries or the object ID and then get the average duration or a count of the number of executions, you can. In fact, SSMS is an way to do this type of simpler aggregation.

If, on the other hand, you want to do an in-depth analysis of the workload, you must import the content of the trace file into a database table. Then you can create much more complex queries. The output from the session puts most of the important data into an XML field, so you'll want to query it as you load the data as follows:

```
DROP TABLE IF EXISTS dbo.ExEvents;
GO
WITH xEvents
AS (SELECT object_name AS xEventName,
           CAST(event_data AS XML) AS xEventData
    FROM sys.fn_xe_file_target_read_file('C:\PerfData\QueryPerfTuning2017*.xel',
                                         NULL,
                                         NULL,
                                         NULL) )
SELECT xEventName,
       xEventData.value('(/event/data[@name="duration"]/value)[1]',
                        'bigint') AS Duration,
       xEventData.value('(/event/data[@name="physical_reads"]/value)[1]',
                        'bigint') AS PhysicalReads,
       xEventData.value('(/event/data[@name="logical_reads"]/value)[1]',
                        'bigint') AS LogicalReads,
       xEventData.value('(/event/data[@name="cpu_time"]/value)[1]',
                        'bigint') AS 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_plan_hash"]/value)[1]',
                        'binary(8)') AS QueryPlanHash
INTO dbo.ExEvents
FROM xEvents;
```

You need to substitute your own path and file name for `<ExEventsFileName>`. Once you have the content in a table, you can use SQL queries to analyze the workload. For example, to find the slowest queries, you can execute this SQL query:

```
SELECT  *
FROM    dbo.ExEvents AS ee
ORDER BY ee.Duration DESC;
```

The preceding query will show the single costliest query, and it is adequate for the tests you're running in this chapter. You may also want to run a query like this on a production system; however, it's more likely you'll want to work from aggregations of data, as in this example:

```
SELECT  ee.SQLText,
        SUM(Duration) AS SumDuration,
        AVG(Duration) AS AvgDuration,
        COUNT(Duration) AS CountDuration
FROM    dbo.ExEvents AS ee
GROUP BY ee.SQLText;
```

Executing this query lets you order things by the fields you're most interested in—say, `CountDuration` to get the most frequently called procedure or `SumDuration` to get the procedure that runs for the longest cumulative amount of time. You need a method to remove or replace parameters and parameter values. This is necessary to aggregate based on just the procedure name or just the text of the query without the parameters or parameter values (since these will be constantly changing).

Another mechanism is to simply query the cache to see the costliest queries through there. It is easier than setting up Extended Events. Further, you'll probably capture most of the bad queries most of the time. Because of this, if you're just getting started with query tuning your system for the first time, you may want to skip setting up Extended Events to identify the costliest queries. However, I've found that as time goes on and you begin to quantify your systems behaviors, you're going to want the kind of detailed data that using Extended Events provides.

One more method we have already explored in the book is using the Query Store to gather metrics on the behavior of the queries in your system. It has the benefits of being extremely easy to set up and easy to query, with no XML involved. The only detriment is if you need granular and detailed performance metrics on individual calls to queries and