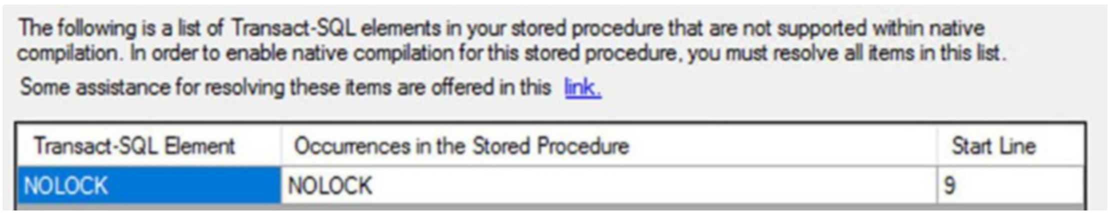
Pay special attention to the note at the top of Figure 24-20. It states that all tables must be memory-optimized tables to natively compile the procedure. But, that check is not part of the Native Compilation Advisor checks.

Clicking Next as prompted, you can then see the problem that was identified by the wizard, as shown in Figure 24-21.



The following is a list of Transact-SQL elements in your stored procedure that are not supported within native compilation. In order to enable native compilation for this stored procedure, you must resolve all items in this list.
Some assistance for resolving these items are offered in this link.

Transact-SQL Element	Occurrences in the Stored Procedure	Start Line
NOLOCK	NOLOCK	9

Figure 24-21. *The problem with the code is identified by the Native Compilation Advisor*

The wizard shows the problematic T-SQL, and it shows the line on which that T-SQL occurs. That's all that's provided by this wizard. If I run the same check against the other procedure, `dbo.WizardPass`, it just reports that there are not any improper T-SQL statements. There is no additional action to compile the procedure for me. To get the procedure to compile, it will be necessary to add the additional functionality as defined earlier in this chapter. Except for this syntax check, there is no other help for natively compiling stored procedures.

Summary

This chapter introduced the concepts of in-memory tables and natively compiled stored procedures. These are high-end methods for achieving potentially massive performance enhancements. There are, however, a lot of limitations on implementing these new objects on a wide scale. You will need to have a machine with enough memory to support the additional load. You're also going to want to carefully test your data and load prior to committing to this approach in a production environment. But, if you do need to make your OLTP systems perform faster than ever before, this is a viable technology. It's also supported within Azure SQL Database.

The next chapter outlines how query and index optimization has been partially automated within SQL Server 2017 and Azure SQL Database.

CHAPTER 25

Automated Tuning in Azure SQL Database and SQL Server

While a lot of query performance tuning involves detailed knowledge of the systems, queries, statistics, indexes, and all the rest of the information put forward in this book, certain aspects of query tuning are fairly mechanical in nature. The process of noticing a missing index suggestion and then testing whether that index helps or hurts a query and whether it hurts other queries could be automated. The same thing goes for certain kinds of bad parameter sniffing where it's clear that one plan is superior to another. Microsoft has started the process of automating these aspects of query tuning. Further, it is putting other forms of automated mechanisms into both Azure SQL Database and SQL Server that will help you by tuning aspects of your queries on the fly. Don't worry, the rest of the book will still be extremely useful because these approaches are only going to fix a fairly narrow range of problems. You'll still have plenty of challenging work to do.

In this chapter, I'll cover the following:

- Automatic plan correction
- Azure SQL Database automatic index management
- Adaptive query processing

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_25

Automatic Plan Correction

The mechanisms behind SQL Server 2017 and Azure SQL Database being able to automatically correct execution plans are best summed up in this way. Microsoft has taken the data now available to it, thanks to the Query Store (for more details on the Query Store, see Chapter 11), and it has weaponized that data to do some amazing things. As your data changes, your statistics can change. You may have a well-written query and appropriate indexes, but over time, as the statistics shift, you might see a new execution plan introduced that hurts performance, basically a bad parameter sniffing issue as outlined in Chapter 17. Other factors could also lead to good query performance turning bad, such as a Cumulative Update changing engine behavior. Whatever the cause, your code and structures still support good performance, if only you can get the right plan in place. Obviously, you can use the tools provided through the Query Store yourself to identify queries that have degraded in performance and which plans supplied better performance and then force those plans. However, notice how the entire process just outlined is very straightforward.

1. Monitor query performance, and note when a query that has not changed in the past suddenly experiences a change in performance.
2. Determine whether the execution plan for that query has changed.
3. If the plan has changed and performance has degraded, force the previous plan.
4. Measure performance to see whether it degrades, improves, or stays the same.
5. If it degrades, undo the change.

In a nutshell, this is what happens within SQL Server. A process within SQL Server observes the query performance within the Query Store. If it sees that the query has remained the same but the performance degraded when the execution plan changed, it will document this as a suggested plan regression. If you enable the automatic tuning of queries, when a regression is identified, it will automatically force the last good plan. The automatic process will then observe behavior to see whether forcing the plan was a bad choice. If it was, it corrects the issue and records that fact for you to look at later. In short, Microsoft automated fixing things such as bad parameter sniffing through automated plan forcing thanks to the data available in the Query Store.

Tuning Recommendations

To start with, let's see how SQL Server identifies tuning recommendations. Since this process is completely dependent on the Query Store, you can enable it only on databases that also have the Query Store enabled (see Chapter 11). With the Query Store enabled, SQL Server, whether 2017 or Azure SQL Database, will automatically begin monitoring for regressed plans. There's nothing else you have to enable once you've enabled the Query Store.

Microsoft is not defining precisely what makes a plan become regressed sufficiently that it is marked as such. So, we're not going to take any chances. We're going to use Adam Machanic's script (`make_big_adventure.sql`) to create some very large tables within AdventureWorks. The script can be downloaded from `http://bit.ly/2mNBIhg`. We also used this in Chapter 9 when working with columnstore indexes. If you are still using the same database, drop those tables and re-create them. This will give us a very large data set from which we can create a query that behaves two different ways depending on the data values passed to it. To see a regressed plan, let's take a look at the following script:

```
CREATE INDEX ix_ActualCost ON dbo.bigTransactionHistory (ActualCost);
GO

--a simple query for the experiment
CREATE OR ALTER PROCEDURE dbo.ProductByCost (@ActualCost MONEY)
AS
SELECT bth.ActualCost
FROM dbo.bigTransactionHistory AS bth
JOIN dbo.bigProduct AS p
ON p.ProductID = bth.ProductID
WHERE bth.ActualCost = @ActualCost;
GO

--ensuring that Query Store is on and has a clean data set
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE = ON;
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE CLEAR;
GO
```

This code creates an index that we're going to use on the dbo.bigTransactionHistory table. It also creates a simple stored procedure that we're going to test. Finally, the script ensures that the Query Store is set to ON and it's clear of all data. With all that in place, we can run our test script as follows:

```
--establish a history of query performance
EXEC dbo.ProductByCost @ActualCost = 8.2205;
GO 30

--remove the plan from cache
DECLARE @PlanHandle VARBINARY(64);
SELECT  @PlanHandle = deps.plan_handle
FROM    sys.dm_exec_procedure_stats AS deps
WHERE   deps.object_id = OBJECT_ID('dbo.ProductByCost');
IF @PlanHandle IS NOT NULL
    BEGIN
        DBCC FREEPROCCACHE(@PlanHandle);
    END
GO

--execute a query that will result in a different plan
EXEC dbo.ProductByCost @ActualCost = 0.0;
GO

--establish a new history of poor performance
EXEC dbo.ProductByCost @ActualCost = 8.2205;
GO 15
```

This will take a while to execute. Once it's complete, we should have a tuning recommendation in our database. Referring to the previous listing, we established a particular behavior in our query by executing it at least 30 times. The query itself returns just a single row of data when the value 8.2205 is used. The plan used looks like Figure 25-1.

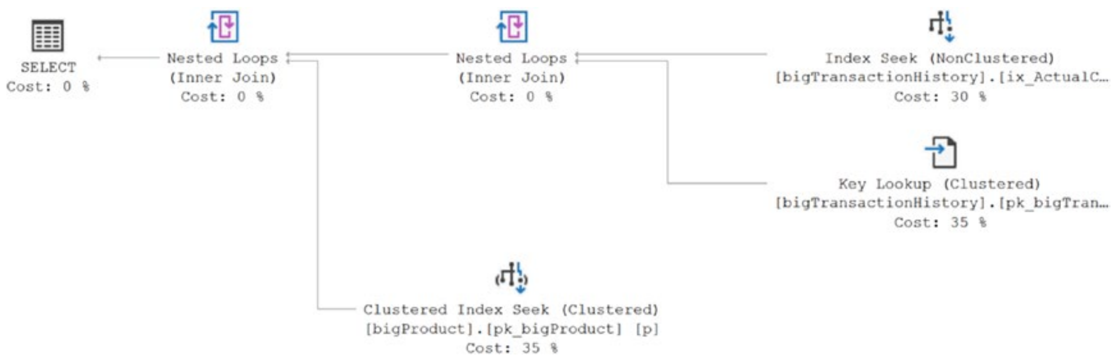

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[bigTransactionHistory].[ix_ActualC…
Cost: 30 %
Key Lookup (Clustered)
[bigTransactionHistory].[pk_bigTran…
Cost: 35 %
Clustered Index Seek (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 35 %

Figure 25-1. *Initial execution plan for the query when returning a small data set*

While the plan shown in Figure 25-1 may have some tuning opportunities, especially with the inclusion of the `Key Lookup` operation, it works well for small data sets. Running the query multiple times builds up a history within the Query Store. Next, we remove the plan from cache, and a new plan is generated when we use the value 0.0, visible in Figure 25-2.

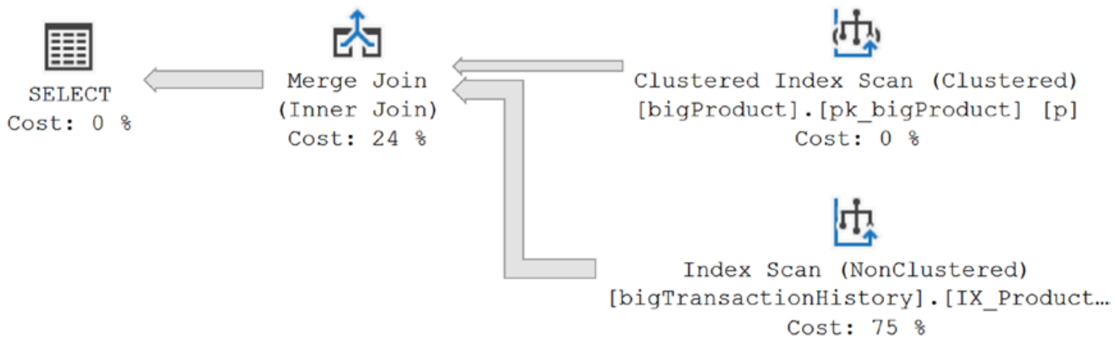

SELECT
Cost: 0 %
Merge Join
(Inner Join)
Cost: 24 %
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 0 %
Index Scan (NonClustered)
[bigTransactionHistory].[IX_Product…
Cost: 75 %

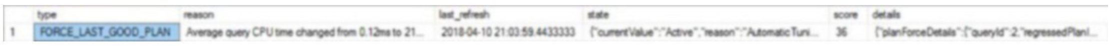
Figure 25-2. *Execution plan for a much larger data set*

After that plan is generated, we execute the procedure a number of additional times (15 seems to work) so that it's clear that we're not looking at a simple anomaly but a true plan regression in progress. That will suggest a tuning recommendation.

We can validate that this set of queries resulted in a tuning recommendation by looking at the new DMV called `sys.dm_db_tuning_recommendations`. Here is an example query returning a limited result set:

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

There is even more than this available from `sys.dm_db_tuning_recommendations`, but let's step through what we have currently, based on the plan regression from earlier. You can see the results of this query in Figure 25-3.



	type	reason	last_refresh	state	score	details
1	FORCE_LAST_GOOD_PLAN	Average query CPU time changed from 0.12ms to 21...	2018-04-10 21:03:59.4433333	{"currentValue":"Active","reason":"AutomaticTuni...	36	{"planForceDetails":{"queryId":2,"regressedPlanI...

Figure 25-3. *First tuning recommendation from sys.dm_db_tuning_ recommendations DMV*

The information presented here is both straightforward and a little confusing. To start with, the `TYPE` value is easy enough to understand. The recommendation here is that we need `FORCE_LAST_GOOD_PLAN` for this query. Currently, this is the only available option at the time of publication, but this will change as additional automatic tuning mechanisms are implemented. The `reason` value is where things get interesting. In this case, the explanation for the need to revert to a previous plan is as follows:

```
Average query CPU time changed from 0.12ms to 2180.37ms
```

Our CPU changed from less than 1ms to just over 2.2 seconds for each execution of the query. That is an easily identifiable issue. The `last_refresh` value tells us the last time any of the data changed within the recommendation. We get the state value, which is a small JSON document consisting of two fields, `currentValue` and `reason`. Here is the document from the previous result set:

```
{"currentValue":"Active","reason":"AutomaticTuningOptionNotEnabled"}
```

It's showing that this recommendation is `Active` but that it was not implemented because we have not yet implemented automatic tuning. There are a number of possible values for the `Status` field. We'll go over them and the values for the `reason` field in the next section, "Enabling Automatic Tuning." The score is an estimated impact value ranging between 0 and 100. The higher the value, the greater the impact of the suggested process. Finally, you get `details`, another JSON document containing quite a bit more information, as you can see here:

```
{"planForceDetails":{"queryId":2,"regressedPlanId":4,
"regressedPlanExecutionCount":15,"regressedPlanErrorCount":0,
"regressedPlanCpuTimeAverage":2.180373266666667e+006,
"regressedPlanCpuTimeStddev":1.680328201712986e+006,
"recommendedPlanId":2,"recommendedPlanExecutionCount":30,
"recommendedPlanErrorCount":0,"recommendedPlanCpuTimeAverage":
1.176333333333333e+002,"recommendedPlanCpuTimeStddev":
6.079253426385694e+001},"implementationDetails":{"method":"TSql",
"script":"exec sp_query_store_force_plan @query_id = 2, @plan_id = 2"}}
```

That's a lot of information in a bit of a blob, so let's break it down more directly into a grid:

`planForceDetails`		
	`queryID`	2: `query_id` value from the Query Store
	`regressedPlanID`	4: `plan_id` value from the Query Store of the problem plan
	`regressedPlanExecutionCount`	15: Number of times the regressed plan was used
	`regressedPlanErrorCount`	0: When there is a value, errors during execution
	`regressedPlanCpuTimeAverage`	2.18037326666667e+006: Average CPU of the plan
	`regressedPlanCpuTimeStddev`	1.60328201712986e+006: Standard deviation of that value

(*continued*)

planForceDetails		
	recommendedPlanID	2: plan_id that the tuning recommendation is suggesting
	recommendedPlanExecutionCount	30: Number of times the recommended plan was used
	recommendedPlanErrorCount	0: When there is a value, errors during execution
	recommendedPlanCpuTimeAverage	1.176333333333333e+002: Average CPU of the plan
	recommendedPlanCpuTimeStddev	6.079253426385694e+001: Standard deviation of that value
implementationDetails		
	Method	TSql: Value will always be T-SQL
	script	exec sp_query_store_force_plan @query_id = 2, @plan_id = 2

That represents the full details of the tuning recommendations. Without ever enabling automatic tuning, you can see suggestions for plan regressions and the full details behind why these suggestions are being made. You even have the script that will enable you to, if you want, execute the suggested fix without enabling automatic plan correction.

With this information, you can then write a much more sophisticated query to retrieve all the information that would enable you to fully investigate these suggestions, including taking a look at the execution plans. All you have to do is query the JSON data directly and then join that to the other information you have from the Query Store, much as this script does:

```
WITH DbTuneRec
AS (SELECT ddtr.reason,
           ddtr.score,
           pfd.query_id,
           pfd.regressedPlanId,
           pfd.recommendedPlanId,
           JSON_VALUE(ddtr.state,
                      '$.currentValue') AS CurrentState,
           JSON_VALUE(ddtr.state,
                      '$.reason') AS CurrentStateReason,
           JSON_VALUE(ddtr.details,
                      '$.implementationDetails.script') AS
ImplementationScript
    FROM sys.dm_db_tuning_recommendations AS ddtr
        CROSS APPLY
        OPENJSON(ddtr.details,
                 '$.planForceDetails')
        WITH (query_id INT '$.queryId',
              regressedPlanId INT '$.regressedPlanId',
              recommendedPlanId INT '$.recommendedPlanId') AS pfd)
SELECT qsq.query_id,
       dtr.reason,
       dtr.score,
       dtr.CurrentState,
       dtr.CurrentStateReason,
       qsqt.query_sql_text,
       CAST(rp.query_plan AS XML) AS RegressedPlan,
       CAST(sp.query_plan AS XML) AS SuggestedPlan,
       dtr.ImplementationScript
FROM DbTuneRec AS dtr
    JOIN sys.query_store_plan AS rp
        ON rp.query_id = dtr.query_id
           AND rp.plan_id = dtr.regressedPlanId
    JOIN sys.query_store_plan AS sp
        ON sp.query_id = dtr.query_id
           AND sp.plan_id = dtr.recommendedPlanId
    JOIN sys.query_store_query AS qsq
        ON qsq.query_id = rp.query_id
    JOIN sys.query_store_query_text AS qsqt
        ON qsqt.query_text_id = qsq.query_text_id;
```

The next steps after you've observed how the tuning recommendations are arrived at are to investigate them, implement them, and then observe their behavior over time, or you can enable automatic tuning so you don't have to baby-sit the process.

You do need to know that the information in `sys.dm_db_tuning_recommendations` is not persisted. If the database or server goes offline for any reason, this information is lost. If you find yourself using this regularly, you should plan on scheduling an export to a more permanent home.

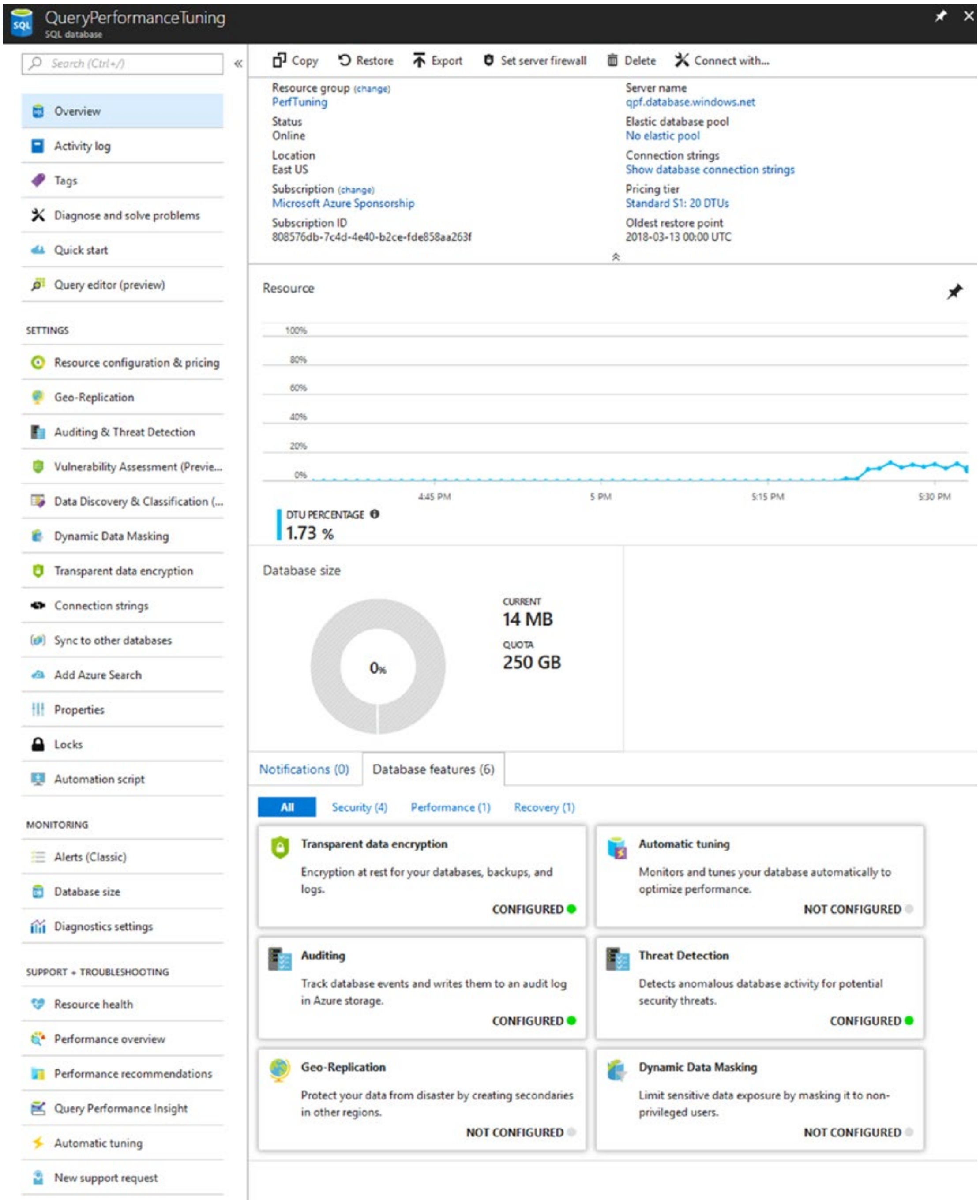
Enabling Automatic Tuning

The process to enable automatic tuning completely depends on if you're working within Azure SQL Database or if you're in SQL Server 2017. Since automatic tuning is dependent on the Query Store, turning it on is also a database-by-database undertaking. Azure offers two methods: using the Azure portal or using T-SQL commands. SQL Server 2017 only supports T-SQL. We'll start with the Azure portal.

Note The Azure portal is updated frequently. Screen captures in this book may be out-of-date, and you may see different graphics when you walk through on your own.

Azure Portal

I'm going to assume you already have an Azure account and that you know how to create an Azure SQL Database and can navigate to it. We'll start from the main blade of a database. You can see all the various standard settings on the left. The top of the page will show the general settings of the database. The center of the page will show the performance metrics. Finally, at the bottom right of the page are the database features. You can see all this in Figure 25-4.



QueryPerformanceTuning
SQL database

Search (Ctrl+/)

- Overview
- Activity log
- Tags
- Diagnose and solve problems
- Quick start
- Query editor (preview)

SETTINGS

- Resource configuration & pricing
- Geo-Replication
- Auditing & Threat Detection
- Vulnerability Assessment (Previe...
- Data Discovery & Classification (...
- Dynamic Data Masking
- Transparent data encryption
- Connection strings
- Sync to other databases
- Add Azure Search
- Properties
- Locks
- Automation script

MONITORING

- Alerts (Classic)
- Database size
- Diagnostics settings

SUPPORT + TROUBLESHOOTING

- Resource health
- Performance overview
- Performance recommendations
- Query Performance Insight
- Automatic tuning
- New support request

Copy | Restore | Export | Set server firewall | Delete | Connect with...

Resource group (change) PerfTuning	Server name qpf.database.windows.net
Status Online	Elastic database pool No elastic pool
Location East US	Connection strings Show database connection strings
Subscription (change) Microsoft Azure Sponsorship	Pricing tier Standard S1: 20 DTUs
Subscription ID 808576db-7c4d-4e40-b2ce-fde858aa263f	Oldest restore point 2018-03-13 00:00 UTC

Resource

DTU PERCENTAGE
1.73 %

Database size

CURRENT
14 MB
QUOTA
250 GB

Notifications (0) | Database features (6)

All | Security (4) | Performance (1) | Recovery (1)

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

Automatic tuning
Monitors and tunes your database automatically to optimize performance.
NOT CONFIGURED ●

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

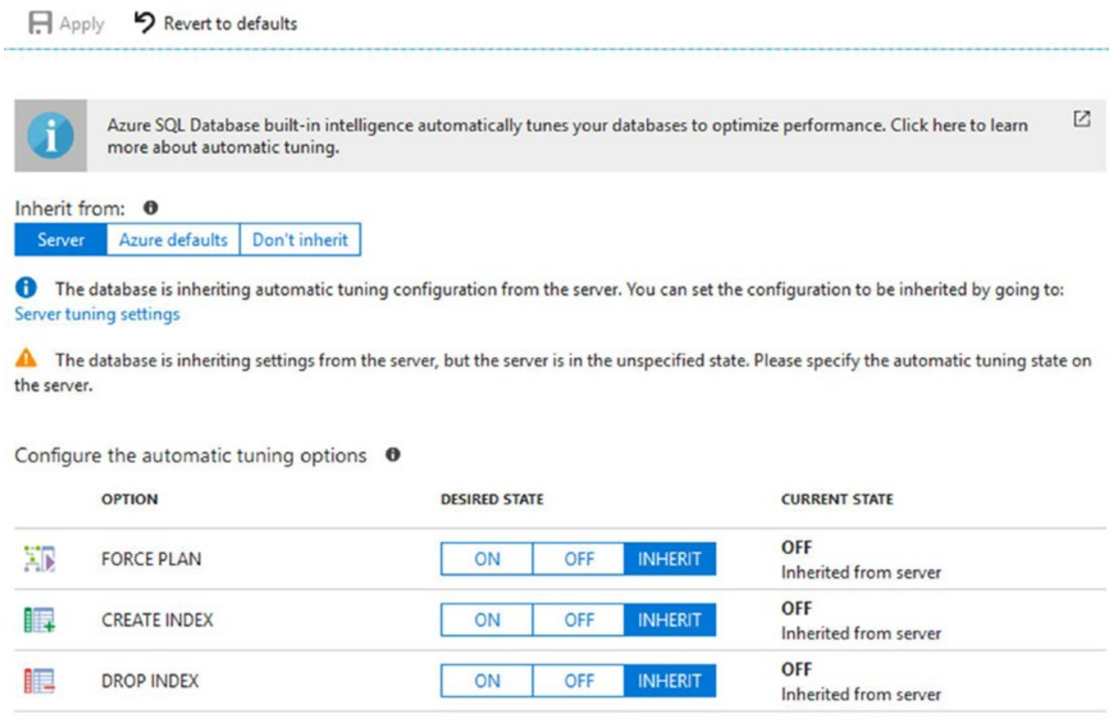
Threat Detection
Detects anomalous database activity for potential security threats.
CONFIGURED ●

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 25-4.** Database blade on Azure portal*

We'll focus down on the details at the lower right of the screen and click the automatic tuning feature. That will open a new blade with the settings for automatic tuning within Azure, as shown in Figure 25-5.



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

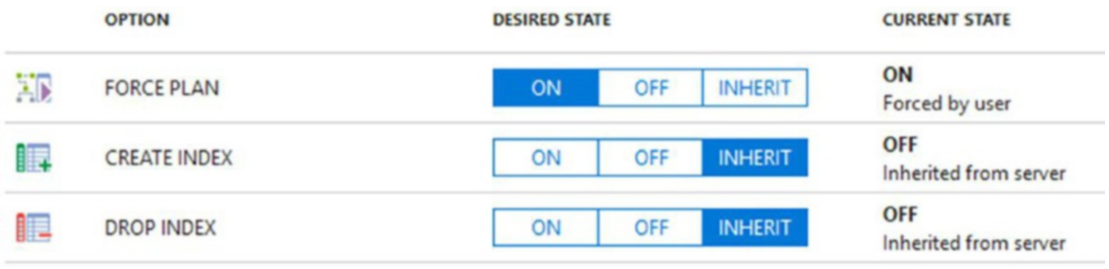
The database is inheriting settings from the server, but the server is in the unspecified state. Please specify the automatic tuning state on the server.

Configure the automatic tuning options

OPTION	DESIRED STATE	CURRENT STATE
FORCE PLAN	ON OFF [✓ INHERIT]	OFF Inherited from server
CREATE INDEX	ON OFF [✓ INHERIT]	OFF Inherited from server
DROP INDEX	ON OFF [✓ INHERIT]	OFF Inherited from server

Figure 25-5. *Automatic tuning features of the Azure SQL Database*

To enable automatic tuning within this database, we change the settings for FORCE PLAN from INHERIT, which is OFF by default, to ON. You will then have to click the Apply button at the top of the page. Once this process is complete, your options should look like mine in Figure 25-6.

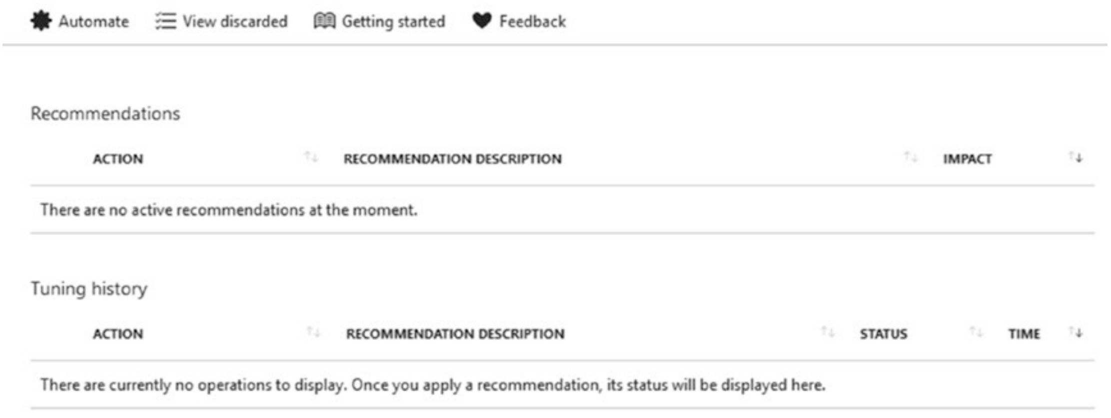


OPTION	DESIRED STATE	CURRENT STATE
FORCE PLAN	[✓ ON] OFF INHERIT	ON Forced by user
CREATE INDEX	ON OFF [✓ INHERIT]	OFF Inherited from server
DROP INDEX	ON OFF [✓ INHERIT]	OFF Inherited from server

Figure 25-6. *Automatic tuning options change to FORCE PLAN as ON*

You can change these settings for the server, and then each database can automatically inherit them. Turning them on or off does not reset connections or in any way take the database offline. The other options will be discussed in the section later in this chapter titled "Azure SQL Database Automatic Index Management."

With this completed, Azure SQL Database will begin to force the last good plan in the event of a regressed query, as you saw earlier in the section "Tuning Recommendations." As before, you can query the DMVs to retrieve the information. You can also use the portal to look at this information. On the left side of the SQL Database blade are the list of functions. Under the heading "Support + Troubleshooting" you'll see "Performance recommendations." Clicking that will bring up a screen similar to Figure 25-7.



Recommendations

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

Tuning history

ACTION	RECOMMENDATION DESCRIPTION	STATUS	TIME
There are currently no operations to display. Once you apply a recommendation, its status will be displayed here.			

Figure 25-7. *Performance recommendations page on the portal*

The information on display in Figure 25-7 should look partly familiar. You've already seen the action, recommendation, and impact from the DMVs we queried in the "Tuning Recommendations" section earlier. From here you can manually apply recommendations, or you can view discarded recommendations. You can also get back to the settings screen by clicking the Automate button. All of this is taking advantage of the Query Store, which is enabled by default in all new databases.

That's all that's needed to enable automatic tuning within Azure. Let's see how to do it within SQL Server 2017.

SQL Server 2017

There is no graphical interface for enabling automatic query tuning within SQL Server 2017 at this point. Instead, you have to use a T-SQL command. You can also use this same command within Azure SQL Database. The command is as follows:

```
ALTER DATABASE current SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = ON);
```

You can of course substitute the appropriate database name for the default value of `current` that I use here. This command can be run on only one database at a time. If you want to enable automatic tuning for all databases on your instance, you have to enable it in the model database before those other databases are created, or you need to turn it on for each database on the server.

The only option currently for `automatic_tuning` is to do as we have done and enable the forcing of the last good plan. You can disable this by using the following command:

```
ALTER DATABASE current SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = OFF);
```

If you run this script, remember to run it again using `ON` to keep plan automated tuning in place.

Automatic Tuning in Action

With the automatic tuning enabled, we can rerun our script that generates a regressed plan. However, just to verify that automated tuning is running, let's use a new system view, `sys.database_automatic_tuning_options`, to verify.

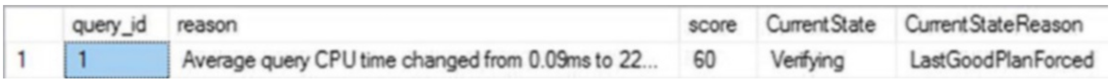
```
SELECT name,
       desired_state,
       desired_state_desc,
       actual_state,
       actual_state_desc,
       reason,
       reason_desc
FROM sys.database_automatic_tuning_options;
```

The results show a `desired_state` value of `1` and a `desired_state_desc` value of `On`.

I clear the cache first when I do it for testing as follows:

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO

EXEC dbo.ProductByCost @ActualCost = 8.2205;
GO 30

--remove the plan from cache
DECLARE @PlanHandle VARBINARY(64);
SELECT  @PlanHandle = deps.plan_handle
FROM    sys.dm_exec_procedure_stats AS deps
WHERE   deps.object_id = OBJECT_ID('dbo.ProductByCost');
IF @PlanHandle IS NOT NULL
    BEGIN
        DBCC FREEPROCCACHE(@PlanHandle);
    END
GO

--execute a query that will result in a different plan
EXEC dbo.ProductByCost @ActualCost = 0.0;
GO

--establish a new history of poor performance
EXEC dbo.ProductByCost @ActualCost = 8.2205;
GO 15
```

Now, when we query the DMV using my sample script from earlier, the results are different, as shown in Figure 25-8.



	query_id	reason	score	CurrentState	CurrentStateReason
1	1	Average query CPU time changed from 0.09ms to 22...	60	Verifying	LastGoodPlanForced

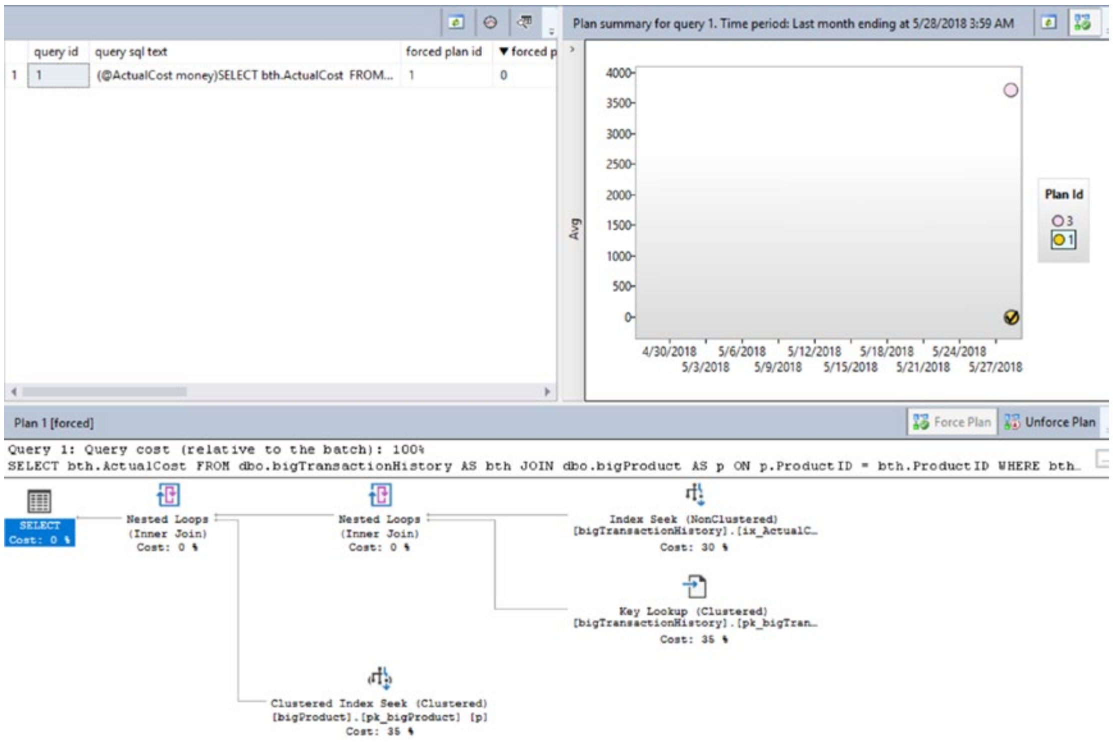
Figure 25-8. *The regressed query has been forced*

The `CurrentState` value has been changed to `Verifying`. It will measure performance over a number of executions, much as it did before. If the performance degrades, it will unforce the plan. Further, if there are errors such as timeouts or aborted executions, the plan will also be unforced. You'll also see the `error_prone` column in `sys.dm_db_tuning_recommendations` changed to a value of Yes in this event.

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…
```

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
```