

## Cursor Overhead

When implementing cursor-centric functionality in an application, you have two choices. You can use either a T-SQL cursor or a database API cursor. Because of the differences between the internal implementation of a T-SQL cursor and a database API cursor, the load created by these cursors on SQL Server is different. The impact of these cursors on the database also depends on the different characteristics of the cursors, such as location, concurrency, and type. You can use Extended Events to analyze the load generated by the T-SQL and database API cursors. The standard events for monitoring queries are, of course, going to be useful. There are also a number of events under the category of *cursor*. The most useful of these events includes the following:

- `cursor_open`
- `cursor_close`
- `cursor_execute`
- `cursor_prepare`

The other events are useful as well, but you'll need them only when you're attempting to troubleshoot specific issues. Even the optimization options for these cursors are different. Let's analyze the overhead of these cursors one by one.

## Analyzing Overhead with T-SQL Cursors

The T-SQL cursors implemented using T-SQL statements are always executed on SQL Server because they need the SQL Server engine to process their T-SQL statements. You can use a combination of the cursor characteristics explained previously to reduce the overhead of these cursors. As mentioned earlier, the most lightweight T-SQL cursor is the one created, not with the default settings but by manipulating the settings to arrive at the forward-only read-only cursor. That still leaves the T-SQL statements used to implement the cursor operations to be processed by SQL Server. The complete load of the cursor

is supported by SQL Server without any help from the client machine. Suppose an application requirement results in the following list of tasks that must be supported:

- Identify all products (from the `Production.WorkOrder` table) that have been scrapped.
- For each scrapped product, determine the money lost, where the money lost per product equals the units in stock *times the* unit price of the product.
- Calculate the total loss.
- Based on the total loss, determine the business status.

The `FOR EACH` phrase in the second point suggests that these application tasks could be served by a cursor. However, a `FOR, WHILE`, cursor, or any other kind of processing of this type can be dangerous within SQL Server. Despite the attraction that this approach holds, it is not set-based, and it is not how you should be processing these types of requests. However, let's see how it works with a cursor. You can implement this application requirement using a T-SQL cursor as follows:

```
CREATE OR ALTER PROC dbo.TotalLoss_CursorBased
AS --Declare a T-SQL cursor with default settings, i.e., fast
--forward-only to retrieve products that have been discarded
DECLARE ScrappedProducts CURSOR FOR
SELECT p.ProductID,
       wo.ScrappedQty,
       p.ListPrice
FROM Production.WorkOrder AS wo
     JOIN Production.ScrapReason AS sr
       ON wo.ScrapReasonID = sr.ScrapReasonID
     JOIN Production.Product AS p
       ON wo.ProductID = p.ProductID;

--Open the cursor to process one product at a time
OPEN ScrappedProducts;

DECLARE @MoneyLostPerProduct MONEY = 0,
        @TotalLoss MONEY = 0;
```

```

--Calculate money lost per product by processing one product
--at a time
DECLARE @ProductId INT,
        @UnitsScrapped SMALLINT,
        @ListPrice MONEY;

FETCH NEXT FROM ScrappedProducts
INTO @ProductId,
    @UnitsScrapped,
    @ListPrice;

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @MoneyLostPerProduct = @UnitsScrapped * @ListPrice; --Calculate
    total loss
    SET @TotalLoss = @TotalLoss + @MoneyLostPerProduct;

    FETCH NEXT FROM ScrappedProducts
    INTO @ProductId,
        @UnitsScrapped,
        @ListPrice;
END

--Determine status
IF (@TotalLoss > 5000)
    SELECT 'We are bankrupt!' AS Status;
ELSE
    SELECT 'We are safe!' AS Status;
--Close the cursor and release all resources assigned to the cursor
CLOSE ScrappedProducts;
DEALLOCATE ScrappedProducts;
GO

```

The stored procedure can be executed as follows, but you should execute it twice to take advantage of plan caching (Figure 23-3):

```
EXEC dbo.TotalLoss_CursorBased;
```

	name	batch_text	duration	logical_reads	row_count
	sql_batch_completed	EXEC dbo.TotalLoss_CursorBased;	32713	8786	2189

**Figure 23-3.** *Extended Events output showing some of the total cost of the data processing using a T-SQL cursor*

The total number of logical reads performed by the stored procedure is 8,786 (indicated by the `sql_batch_completed` event in Figure 23-3). Well, is it high or low? Considering the fact that the `Production.Products` table has only 6,196 pages and the `Production.WorkOrder` table has only 926, it's surely not low. You can determine the number of pages allocated to these tables by querying the dynamic management view `sys.dm_db_index_physical_stats`.

```
SELECT SUM(page_count)
FROM sys.dm_db_index_physical_stats(DB_ID(N'AdventureWorks2017'),
    OBJECT_ID('Production.WorkOrder'),
    DEFAULT, DEFAULT, DEFAULT);
```

---

**Note** The `sys.dm_db_index_physical_stats` DMV is explained in detail in Chapter 13.

---

In most cases, you can avoid cursor operations by rewriting the functionality using SQL queries, concentrating on set-based methods of accessing the data. For example, you can rewrite the preceding stored procedure using SQL queries (instead of the cursor operations) as follows:

```
CREATE OR ALTER PROC dbo.TotalLoss
AS
SELECT CASE --Determine status based on following computation
    WHEN SUM(MoneyLostPerProduct) > 5000 THEN
        'We are bankrupt!'
    ELSE
        'We are safe!'
END AS Status
```

```
FROM
( --Calculate total money lost for all discarded products
  SELECT SUM(wo.ScrapQty * p.ListPrice) AS MoneyLostPerProduct
  FROM Production.WorkOrder AS wo
    JOIN Production.ScrapReason AS sr
      ON wo.ScrapReasonID = sr.ScrapReasonID
    JOIN Production.Product AS p
      ON wo.ProductID = p.ProductID
  GROUP BY p.ProductID) AS DiscardedProducts;
GO
```

In this stored procedure, the aggregation functions of SQL Server are used to compute the money lost per product and the total loss. The CASE statement is used to determine the business status based on the total loss incurred. The stored procedure can be executed as follows; but again, you should execute it twice, so you can see the results of plan caching:

```
EXEC dbo.TotalLoss;
```

Figure 23-4 shows the corresponding Extended Events output.

	name	batch_text	duration	logical_reads	row_count
	sql_batch_completed	EXEC dbo.TotalLoss;	10397	547	1

**Figure 23-4.** Extended Events output showing the total cost of the data processing using an equivalent SELECT statement

In Figure 23-4, you can see that the second execution of the stored procedure, which reuses the existing plan, uses a total of 547 logical reads. However, you can see a result even more important than the reads: the duration falls from 32.7ms to 10.3ms. Using SQL queries instead of the cursor operations made the execution three times faster.

Therefore, for better performance, it is almost always recommended that you use set-based operations in SQL queries instead of T-SQL cursors.

## Cursor Recommendations

An ineffective use of cursors can degrade the application performance by introducing extra network round-trips and load on server resources. To keep the cursor cost low, try to follow these recommendations:

- Use set-based SQL statements over T-SQL cursors since SQL Server is designed to work with sets of data.
- Use the least expensive cursor.
  - When using SQL Server cursors, use the `FAST FORWARD` cursor type.
  - When using the API cursors implemented by ADO, OLEDB, or ODBC, use the default cursor type, which is generally referred to as the *default result set*.
  - When using [ADO.NET](#), use the `DataReader` object.
- Minimize impact on server resources.
  - Use a client-side cursor for API cursors.
  - Do not perform actions on the underlying tables through the cursor.
  - Always deallocate the cursor as soon as possible. This helps free resources, especially in tempdb.
  - Redesign the cursor's `SELECT` statement (or the application) to return the minimum set of rows and columns.
  - Avoid T-SQL cursors entirely by rewriting the logic of the cursor as set-based statements, which are generally more efficient than cursors.
  - Use a `ROWVERSION` column for dynamic cursors to benefit from the efficient, version-based concurrency control instead of relying upon the value-based technique.

- Minimize impact on tempdb.
  - Minimize resource contention in tempdb by avoiding the static and keyset-driven cursor types.
  - Static and key-set cursors put additional load on tempdb, so take that into account if you must use them, or avoid them if your tempdb is under stress.
- Minimize blocking.
  - Use the default result set, fast-forward-only cursor, or static cursor.
  - Process all cursor rows as quickly as possible.
  - Avoid scroll locks or pessimistic locking.
- Minimize network round-trips while using API cursors.
  - Use the CacheSize property of ADO to fetch multiple rows in one round-trip.
  - Use client-side cursors.
  - Use disconnected record sets.

## Summary

As you learned in this chapter, a cursor is the natural extension to the result set returned by SQL Server, enabling the calling application to process one row of data at a time. Cursors add a cost overhead to application performance and impact the server resources.

You should always be looking for ways to avoid cursors. Set-based solutions work better in almost all cases. However, if a cursor operation is mandated, then choose the best combination of cursor location, concurrency, type, and cache size characteristics to minimize the cost overhead of the cursor.

In the next chapter, we explore the special functionality introduced with in-memory tables, natively compiled procedures, and the other aspects of memory-optimized objects.

## CHAPTER 24

# Memory-Optimized OLTP Tables and Procedures

One of the principal needs for online transaction processing (OLTP) systems is to get as much speed as possible out of the system. With this in mind, Microsoft introduced the in-memory OLTP enhancements. These were improved on in subsequent releases and added to Azure SQL Database. The memory-optimized technologies consist of in-memory tables and natively compiled stored procedures. This set of features is meant for high-end, transaction-intensive, OLTP-focused systems. In SQL Server 2014, you had access to the in-memory OLTP functionality only in the Enterprise edition of SQL Server. Since SQL Server 2016, all editions support this enhanced functionality. The memory-optimized technologies are another tool in the toolbox of query tuning, but they are a highly specialized tool, applicable only to certain applications. Be cautious in adopting this technology. That said, on the right system with the right amount of memory, in-memory tables and native stored procedures result in blazing-fast speed.

In this chapter, I cover the following topics:

- The basics of how in-memory tables work
- Improving performance by natively compiling stored procedures
- The benefits and drawbacks of natively compiled procedures and in-memory OLTP tables
- Recommendations for when to use in-memory OLTP tables



## In-Memory OLTP Fundamentals

At the core of it all, you can tune your queries to run incredibly fast. But, no matter how fast you make them run, to a degree you're limited by some of the architectural issues within modern computers and the fundamentals of the behavior of SQL Server. Typically, the number-one bottleneck with your hardware is the storage system. Whether you're still looking at spinning platters or you've moved on to some type of SSD or similar technology, the disks are still the slowest aspect of the system. This means for reads or writes, you have to wait. But memory is fast, and with 64-bit operating systems, it can be plentiful. So, if you have tables that you can move completely into memory, you can radically improve the speed. That's part of what in-memory OLTP tables are all about: moving the data access, both reads and writes, into memory and off the disk.

However, Microsoft did more than simply move tables into memory. It recognized that while the disk was slow, another aspect of the system slowing things down was how the data was accessed and managed through the transaction system. So, Microsoft made a series of changes there as well. The primary one was removing the pessimistic approach to transactions. The existing product forces all transactions to get written to the transaction log before allowing the data changes to get flushed to disk. This creates a bottleneck in the processing of transactions. So, instead of pessimism about whether a transaction will successfully complete, Microsoft took an optimistic approach that most of the time, transactions will complete. Further, instead of having a blocking situation where one transaction has to finish updating data before the next can access it or update it, Microsoft versioned the data. It has now eliminated a major point of contention within the system and eliminated locks, and with all this is in memory, so it's even faster.

Microsoft then took all this another step further. Instead of the pessimistic approach to memory latches that prevent more than one process from accessing a page to write to it, Microsoft extended the optimistic approach to memory management. Now, with versioning, in-memory tables work off a model that is "eventually" consistent with a conflict resolution process that will roll back a transaction but never block one transaction by another. This has the potential to lead to some data loss, but it makes everything within the data access layer fast.

Data does get written to disk in order to persist in a reboot or similar situation. However, nothing is read from disk except at the time of starting the server (or bringing the database online). Then all the data for the in-memory tables is loaded into memory and no reads occur against the disk again for any of that data. However, if you are dealing

with temporary data, you can even short circuit this functionality by defining the data as not being persisted to disk at all, reducing even the startup times.

Finally, as you've seen throughout the rest of the book, a major part of query tuning is figuring out how to work with the query optimizer to get a good execution plan and then have that plan reused multiple times. This can also be an intensive and slow process. SQL Server 2014 introduced the concept of natively compiled stored procedures. These are literally T-SQL code compiled down to DLLs and made part of the SQL Server OS. This compile process is costly and shouldn't be used for just any old query. The principal idea is to spend time and effort compiling a procedure to native code and then get to use that procedure millions of times at a radically improved speed.

All this technology comes together to create new functionality that you can use by itself or in combination with existing table structures and standard T-SQL. In fact, you can treat in-memory tables much the same way as you treat normal SQL Server tables and still realize some performance improvements. But, you can't just do this anywhere. There are some fairly specific requirements for taking advantage of in-memory OLTP tables and procedures.

## System Requirements

You must meet a few standard requirements before you can even consider whether memory-optimized tables are a possibility.

- A modern 64-bit processor
- Twice the amount of free disk storage for the data you intend to put into memory
- Lots of memory

Obviously, for most systems, the key is lots of memory. You need to have enough memory for the operating system and SQL Server to function normally. Then you still need to have memory for all the non-memory-optimized requirements of your system including the data cache. Finally, you're going to add, on top of all that, memory for your memory-optimized tables. If you're not looking at a fairly large system, with a minimum of 64GB memory, I don't suggest even considering this as an option. Smaller systems are just not going to provide enough storage in memory to make this worth the time and effort.

In SQL Server 2014 only, you must have the Enterprise edition of SQL Server running. You can also use the Developer edition in SQL Server 2014, of course, but you can't run production loads on that. For versions newer than SQL Server 2014, there are memory limits based on the editions as published by Microsoft.

## Basic Setup

In addition to the hardware requirements, you have to do additional work on your database to enable in-memory tables. I'll start with a new database to illustrate.

```
CREATE DATABASE InMemoryTest
ON PRIMARY (NAME = N'InMemoryTest_Data',
            FILENAME = N'D:\Data\InMemoryTest_Data.mdf',
            SIZE = 5GB)
LOG ON (NAME = N'InMemoryTest_Log',
        FILENAME = N'L:\Log\InMemoryTest_Log.ldf');
```

For the in-memory tables to maintain durability, they must write to disk as well as to memory since memory goes away with the power. Durability (part of the ACID properties of a relational dataset) means that once a transaction commits, it stays committed. You can have a durable in-memory table or a nondurable table. With a nondurable table, you may have committed transactions, but you could still lose that data, which is different from how standard tables work within SQL Server. The most commonly known uses for data that isn't durable are things such as session state or time-sensitive information such as an electronic shopping cart. Anyway, in-memory storage is not the same as the usual storage within your standard relational tables. So, a separate file group and files must be created. To do this, you can just alter the database, as shown here:

```
ALTER DATABASE InMemoryTest
ADD FILEGROUP InMemoryTest_InMemoryData
CONTAINS MEMORY_OPTIMIZED_DATA;
ALTER DATABASE InMemoryTest
ADD FILE (NAME = 'InMemoryTest_InMemoryData',
          FILENAME = 'D:\Data\InMemoryTest_InMemoryData.ndf')
TO FILEGROUP InMemoryTest_InMemoryData;
```

I would have simply altered the AdventureWorks2017 database that you've been experimenting with, but another consideration for in-memory optimized tables is that you can't remove the special filegroup once it's created. You can only ever drop the database. That's why I'll just experiment with a separate database. It's safer. It's also one of the drivers for you being cautious about how and where you implement in-memory technology. You simply can't try it on your production servers without permanently altering them.

There are some limitations to features available to a database using in-memory OLTP.

- `DBCC CHECKDB`: You can run consistency checks, but the memory-optimized tables will be skipped. You'll get an error if you attempt to run `DBCC CHECKTABLE`.
- `AUTO_CLOSE`: This is not supported.
- `DATABASE_SNAPSHOT`: This is not supported.
- `ATTACH_REBUILD_LOG`: This is also not supported.
- *Database mirroring*: You cannot mirror a database with a `MEMORY_OPTIMIZED_DATA` file group. However, availability groups provide a seamless experience, and Failover Clustering supports in-memory tables (but it will affect recovery time).

Once these modifications are complete, you can begin to create in-memory tables in your system.

## Create Tables

Once the database setup is complete, you have the capability to create tables that will be memory optimized, as described earlier. The actual syntax is quite straightforward. I'm going to replicate, as much as I can, the `Person.Address` table from AdventureWorks2017.

```
USE InMemoryTest;
GO
CREATE TABLE dbo.Address
    (AddressID INT IDENTITY(1, 1) NOT NULL PRIMARY KEY NONCLUSTERED HASH
        WITH (BUCKET_COUNT = 50000),
    AddressLine1 NVARCHAR(60) NOT NULL,
```

```

    AddressLine2 NVARCHAR(60) NULL,
    City NVARCHAR(30) NOT NULL,
    StateProvinceID INT NOT NULL,
    PostalCode NVARCHAR(15) NOT NULL,
    --[SpatialLocation geography NULL,
    --rowguid uniqueidentifier ROWGUIDCOL NOT NULL CONSTRAINT DF_
Address_rowguid DEFAULT (newid()),
    ModifiedDate DATETIME NOT NULL
        CONSTRAINT DF_Address_ModifiedDate
        DEFAULT (GETDATE()))
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);

```

This creates a durable table in the memory of the system using the disk space you defined to retain a durable copy of the data, ensuring that you won't lose data in the event of a power loss. It has a primary key that is an IDENTITY value just like with a regular SQL Server table (however, to use IDENTITY instead of SEQUENCE, you will be surrendering the capability to set the definition to anything except (1,1) in this version of SQL Server). You'll note that the index definition is not clustered. Instead, it's NON-CLUSTERED HASH. I'll talk about indexing and things like BUCKET\_COUNT in the next section. You'll also note that I had to comment out two columns, SpatialLocation and rowguid. These are using data types not available with in-memory tables. Finally, the WITH statement lets SQL Server know where to place this table by defining MEMORY\_OPTIMIZED=ON. You can make an even faster table by modifying the WITH clause to use DURABILITY=SCHEMA\_ONLY. This allows data loss but makes the table even faster since nothing gets written to disk.

There are a number of unsupported data types that could prevent you from taking advantage of in-memory tables.

- XML
- ROWVERSION
- SQL\_VARIANT
- HIERARCHYID
- DATETIMEOFFSET
- GEOGRAPHY/GEOMETRY
- User-defined data types

In addition to data types, you will run into other limitations. I'll talk about the index requirements in the "In-Memory Indexes" section. Starting with SQL Server 2016, support for foreign keys and check constraints and unique constraints was added.

Once a table is created in-memory, you can access it just like a regular table. If I were to run a query against it now, it wouldn't return any rows, but it would function.

```
SELECT a.AddressID
FROM dbo.Address AS a
WHERE a.AddressID = 42;
```

So, to experiment with some actual data in the database, go ahead and load the information stored in `Person.Address` in `AdventureWorks2017` into the new table that's stored in-memory in this new database.

```
CREATE TABLE dbo.AddressStaging (AddressLine1 NVARCHAR(60) NOT NULL,
                                   AddressLine2 NVARCHAR(60) NULL,
                                   City NVARCHAR(30) NOT NULL,
                                   StateProvinceID INT NOT NULL,
                                   PostalCode NVARCHAR(15) NOT NULL);
```

```
INSERT dbo.AddressStaging (AddressLine1,
                           AddressLine2,
                           City,
                           StateProvinceID,
                           PostalCode)
```

```
SELECT a.AddressLine1,
       a.AddressLine2,
       a.City,
       a.StateProvinceID,
       a.PostalCode
FROM AdventureWorks2017.Person.Address AS a;
```

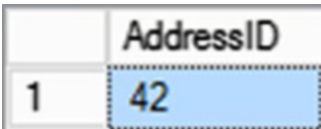
```
INSERT dbo.Address (AddressLine1,
                    AddressLine2,
                    City,
                    StateProvinceID,
                    PostalCode)
```

```
SELECT a.AddressLine1,
       a.AddressLine2,
       a.City,
       a.StateProvinceID,
       a.PostalCode
FROM dbo.AddressStaging AS a;

DROP TABLE dbo.AddressStaging;
```

You can't combine an in-memory table in a cross-database query, so I had to load the approximate 19,000 rows into a staging table and then load them into the in-memory table. This is not meant to be part of the examples for performance, but it's worth nothing that it took nearly 850ms to insert the data into the standard table and only 2ms to load the same data into the in-memory table on my system.

But, with the data in place, I can rerun the query and actually see results, as shown in Figure 24-1.



	AddressID
1	42

**Figure 24-1.** The first query results from an in-memory table

Granted, this is not terribly exciting. So, to have something meaningful to work with, I'm going to create a couple of other tables so that you can see some more query behavior on display.

```
CREATE TABLE dbo.StateProvince (StateProvinceID INT IDENTITY(1, 1) NOT NULL
PRIMARY KEY NONCLUSTERED HASH
    WITH (BUCKET_COUNT = 10000),
    StateProvinceCode NCHAR(3) COLLATE Latin1_General_100_BIN2 NOT NULL,
    CountryRegionCode NVARCHAR(3) NOT NULL,
    Name VARCHAR(50) NOT NULL,
    TerritoryID INT NOT NULL,
    ModifiedDate DATETIME NOT NULL
    CONSTRAINT DF_StateProvince_ModifiedDate
        DEFAULT (GETDATE()))
WITH (MEMORY_OPTIMIZED = ON);
```

```

CREATE TABLE dbo.CountryRegion (CountryRegionCode NVARCHAR(3) NOT NULL,
                                Name VARCHAR(50) NOT NULL,
                                ModifiedDate DATETIME NOT NULL
                                CONSTRAINT DF_CountryRegion_ModifiedDate
                                    DEFAULT (GETDATE()),
                                CONSTRAINT PK_CountryRegion_CountryRegionCode
                                    PRIMARY KEY CLUSTERED
                                (
                                    CountryRegionCode ASC
                                ));

```

That's an additional memory-optimized table and a standard table. I'll also load data into these so you can make more interesting queries.

```

SELECT sp.StateProvinceCode,
       sp.CountryRegionCode,
       sp.Name,
       sp.TerritoryID
INTO dbo.StateProvinceStaging
FROM AdventureWorks2017.Person.StateProvince AS sp;

INSERT dbo.StateProvince (StateProvinceCode,
                          CountryRegionCode,
                          Name,
                          TerritoryID)

SELECT StateProvinceCode,
       CountryRegionCode,
       Name,
       TerritoryID
FROM dbo.StateProvinceStaging;

DROP TABLE dbo.StateProvinceStaging;

INSERT dbo.CountryRegion (CountryRegionCode,
                          Name)

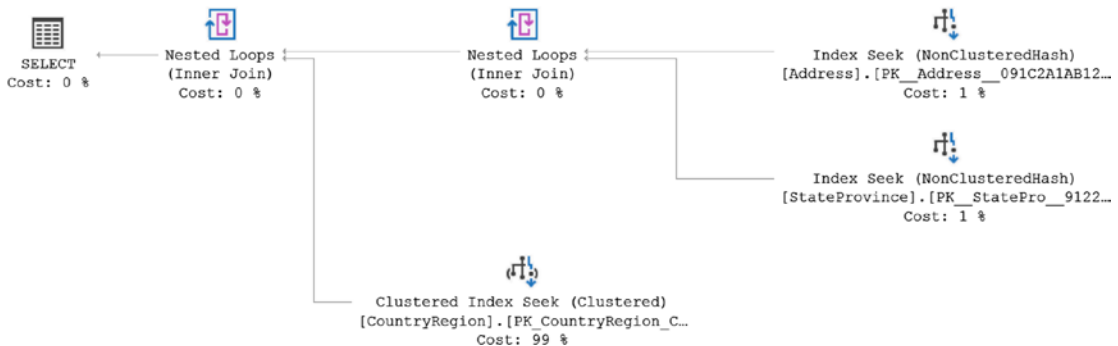
SELECT cr.CountryRegionCode,
       cr.Name
FROM AdventureWorks2017.Person.CountryRegion AS cr;

```



With the data loaded, the following query returns a single row and has an execution plan that looks like Figure 24-2:

```
SELECT a.AddressLine1,
       a.City,
       a.PostalCode,
       sp.Name AS StateProvinceName,
       cr.Name AS CountryName
FROM dbo.Address AS a
     JOIN dbo.StateProvince AS sp
       ON sp.StateProvinceID = a.StateProvinceID
     JOIN dbo.CountryRegion AS cr
       ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.AddressID = 42;
```



**Figure 24-2.** An execution plan showing both in-memory and standard tables

As you can see, it's entirely possible to get a normal execution plan even when using in-memory tables. The operators are even the same. In this case, you have three different index seek operations. Two of them are against the nonclustered hash indexes you created with the in-memory tables, and the other is a standard clustered index seek against the standard table. You might also note that the estimated cost on this plan adds up to 101 percent. You may occasionally see such anomalies dealing with in-memory tables since the cost for them through the optimizer is so radically different than regular tables.

The principal performance enhancements come from the lack of locking and latching, allowing massive inserts and updates while simultaneously allowing for querying. But, the queries do run faster as well. The previous query resulted in the execution time and reads shown in Figure 24-3.

	name	batch_text	duration	logical_reads	row_count
	sql_batch_completed	SELECT a.AddressLine1, a.City, ...	107	2	1

**Figure 24-3.** Query metrics for an in-memory table

Running a similar query against the AdventureWorks2017 database results in the behavior shown in Figure 24-4.

	name	batch_text	duration	logical_reads	row_count
	sql_batch_completed	SELECT a.AddressLine1, a.City, ...	152	6	1

**Figure 24-4.** Query metrics for a regular table

While it's clear that the execution times are much better with the in-memory table, what's not clear is how the reads are dealt with. But, since I'm talking about reading from the in-memory storage and not either pages in memory or pages on the disk but the hash index instead, things are completely different in terms of measuring performance. You won't be using all the same measures as before but will instead rely on execution time. The reads in this case are a measure of the activity of the system, so you can anticipate that higher values mean more access to the data and lower values mean less.

With the tables in place and proof of improved performance both for inserts and for selects, let's talk about the indexes that you can use with in-memory tables and how they're different from standard indexes.

## In-Memory Indexes

An in-memory table can have up to eight indexes created on it at one time. But, every memory-optimized table must have at least one index. The index defined by the primary key counts. A durable table must have a primary key. You can create three index types: the nonclustered hash index that you used previously, the nonclustered index, and the columnstore indexes. These indexes are not the type of indexes that are created with

standard tables. First, they're maintained in memory in the same way the in-memory tables are. Second, the same rules apply about durability of the indexes as the in-memory tables. In-memory indexes do not have a fixed page size either, so they won't suffer from fragmentation. Let's discuss each of the index types in a little more detail.

## Hash Index

A hash index is not a balanced-tree index that's just stored in memory. Instead, the hash index uses a predefined hash bucket, or table, and hash values of the key to provide a mechanism for retrieving the data of a table. SQL Server has a hash function that will always result in a constant hash value for the inputs provided. This means for a given key value, you'll always have the same hash value. You can store multiple copies of the hash value in the hash bucket. Having a hash value to retrieve a point lookup, a single row, makes for an extremely efficient operation, that is, as long as you don't run into lots of hash collisions. A hash collision is when you have multiple values stored at the same location.

This means the key to getting the hash index right is getting the right distribution of values across buckets. You do this by defining the bucket count for the index. For the first table I created, `dbo.Address`, I set a bucket count of 50,000. There are 19,000 rows currently in the table. So, with a bucket count of 50,000, I ensure that I have plenty of storage for the existing set of values, and I provide a healthy growth overhead. You need to set the bucket count so that it's big enough without being too big. If the bucket count is too small, you'll be storing lots of data within a bucket and seriously impact the ability of the system to efficiently retrieve the data. In short, it's best to have your bucket be too big. If you look at Figure 24-5, you can see this laid out in a different way.



**Figure 24-5.** Hash values in lots of buckets and few buckets

The first set of buckets has what is called a *shallow distribution*, which is few hash values distributed across a lot of buckets. This is a more optimal storage plan. Some buckets may be empty as you can see, but the lookup speed is fast because each bucket contains a single value. The second set of buckets shows a small bucket count, or a *deep distribution*. This is more hash values in a given bucket, requiring a scan within the bucket to identify individual hash values.

Microsoft's recommendation on bucket count is go between one to two times the quantity of the number of rows in the table. But, since you can't alter in-memory tables, you also need to consider projected growth. If you think your in-memory table is likely to grow three times as large over the next three to six months, you may want to expand the size of your bucket count. The only problem you'll encounter with an oversized bucket count is that scans will take longer, so you'll be allocating more memory. But, if your queries are likely to lead to scans, you really shouldn't be using the nonclustered hash index. Instead, just go to the nonclustered index. The current recommendation is to go to no more than ten times the number of unique values you're likely to be dealing with when setting the bucket count.

You also need to worry about how many values can be returned by the hash value. Unique indexes and primary keys are prime candidates for using the hash index because they're always unique. Microsoft's recommendation is that if, on average, you're going

to see more than five values for any one hash value, you should move away from the nonclustered hash index and use the nonclustered index instead. This is because the hash bucket simply acts as a pointer to the first row that is stored in that bucket. Then, if duplicate or additional values are stored in the bucket, the first row points to the next row, and each subsequent row points to the row following it. This can turn point lookups into scanning operations, again radically hurting performance. This is why going with a small number of duplicates, less than five, or unique values work best with hash indexes.

To see the distribution of your index within the hash table, you can use `sys.dm_db_xtp_hash_index_stats`.

```
SELECT i.name AS [index name],
       hs.total_bucket_count,
       hs.empty_bucket_count,
       hs.avg_chain_length,
       hs.max_chain_length
FROM sys.dm_db_xtp_hash_index_stats AS hs
     JOIN sys.indexes AS i
       ON hs.object_id = i.object_id
          AND hs.index_id = i.index_id
WHERE OBJECT_NAME(hs.object_id) = 'Address';
```

Figure 24-6 shows the results of this query.

	index name	total_bucket_count	empty_bucket_count	avg_chain_length	max_chain_length
1	PK_Address__091C2A1AB12B1E34	65536	48652	1	5

**Figure 24-6.** Results of querying `sys.dm_db_xtp_hash_index_stats`

With this you can see a few interesting facts about how hash indexes are created and maintained. You’ll note that the total bucket count is not the value I set, 50,000. The bucket count is rounded up to the next closest power of two, in this case, 65,536. There are 48,652 empty buckets. The average chain length, since this is a unique index, is a value of 1 because the values are unique. There are some chain values because as rows get modified or updated there will be versions of the data stored until everything is resolved.

## Nonclustered Indexes

The nonclustered indexes are basically just like regular indexes except that they're stored in memory along with the data to assist in data retrieval. They also have pointers to the storage location of the data similar to how a nonclustered index behaves with a heap table. One interesting difference between an in-memory nonclustered index and a standard nonclustered index is that SQL Server can't retrieve the data in reverse order from the in-memory index. Other behavior seems close to the same as standard indexes.

To see the nonclustered index in action, let's take this query:

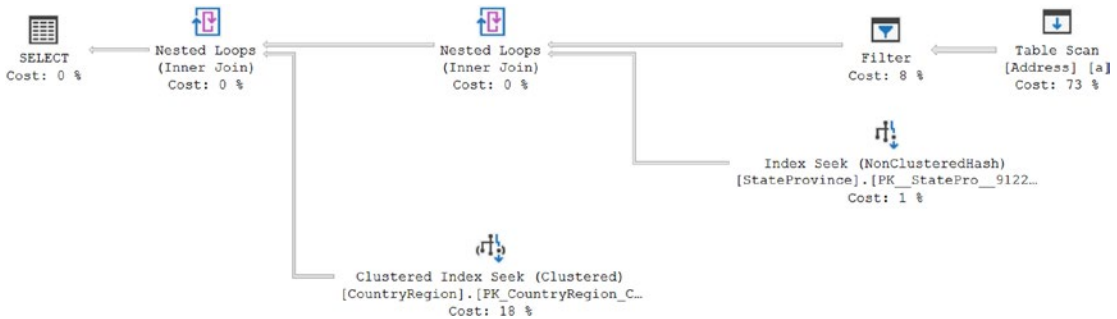
```
SELECT  a.AddressLine1,
        a.City,
        a.PostalCode,
        sp.Name AS StateProvinceName,
        cr.Name AS CountryName
FROM    dbo.Address AS a
        JOIN dbo.StateProvince AS sp
        ON sp.StateProvinceID = a.StateProvinceID
        JOIN dbo.CountryRegion AS cr
        ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE   a.City = 'Walla Walla';
```

Currently the performance looks like Figure 24-7.

	name	batch_text	duration	logical_reads	row_count
	sql_batch_completed	SELECT a.AddressLine1, a.City...	3561	200	100

**Figure 24-7.** Metrics of query without an index

Figure 24-8 shows the execution plan.



**Figure 24-8.** Query results in an execution plan that has table scans

While an in-memory table scan is certainly going to be faster than the same scan on a table stored on disk, it's still not a good situation. Plus, considering the extra work resulting from the Filter operation and the Sort operation to satisfy the Merge Join that the optimizer felt it needed, this is a problematic query. So, you should add an index to the table to speed it up.

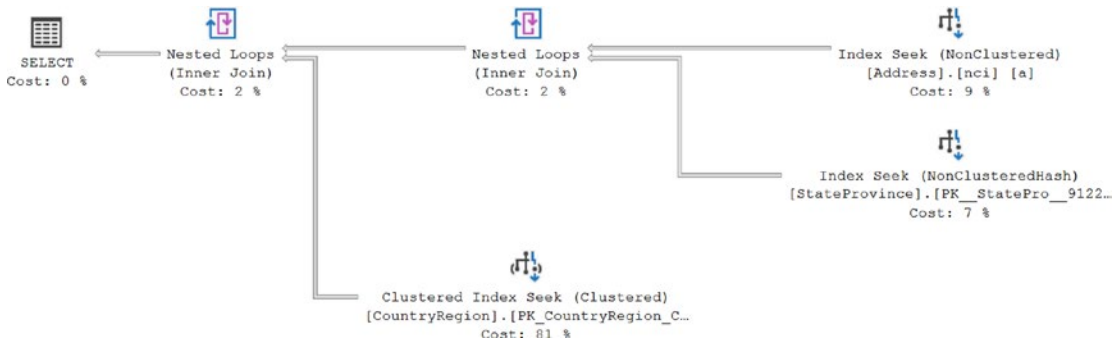
But, you can't just run `CREATE INDEX` on the `dbo.Address` table. Instead, you have two choices, re-creating the table or altering the table. You'll need to test your system as to which works better. The `ALTER TABLE` command for adding an index to an in-memory table can be costly. If you wanted to drop the table and re-create it, the table creation script now looks like this:

```
CREATE TABLE dbo.Address (
    AddressID INT IDENTITY(1, 1) NOT NULL PRIMARY KEY NONCLUSTERED HASH
        WITH (BUCKET_COUNT = 50000),
    AddressLine1 NVARCHAR(60) NOT NULL,
    AddressLine2 NVARCHAR(60) NULL,
    City NVARCHAR(30) NOT NULL,
    StateProvinceID INT NOT NULL,
    PostalCode NVARCHAR(15) NOT NULL,
    ModifiedDate DATETIME NOT NULL
    CONSTRAINT DF_Address_ModifiedDate
        DEFAULT (GETDATE()),
    INDEX nci NONCLUSTERED (City))
WITH (MEMORY_OPTIMIZED = ON);
```

Creating the same index using the ALTER TABLE command looks like this:

```
ALTER TABLE dbo.Address ADD INDEX nci (City);
```

After reloading the data into the newly created table, you can try the query again. This time it ran in 800 microseconds on my system, much faster than the 3.7ms it ran in previously. The reads stayed the same. Figure 24-9 shows the execution plan.



**Figure 24-9.** An improved execution plan taking advantage of nonclustered indexes

As you can see, the nonclustered index was used instead of a table scan to improve performance much as you would expect from an index on a standard table. However, unlike the standard table, while this query did pull columns that were not part of nonclustered index, no key lookup was required to retrieve the data from the in-memory table because each index points directly to the storage location, in memory, of the data necessary. This is yet another small but important improvement over how standard tables behave.

## Columnstore Index

There actually isn't much to say about adding a columnstore index to an in-memory table. Since columnstore indexes work best on tables with 100,000 rows or more, you will need quite a lot of memory to support their implementation on your in-memory tables. You are limited to clustered columnstore indexes. You also cannot apply a filtered columnstore index to an in-memory table. Except for those limitations, the creation of an in-memory columnstore index is the same as the indexes we've already seen:

```
ALTER TABLE dbo.Address ADD INDEX ccs CLUSTERED COLUMNSTORE;
```



## Statistics Maintenance

There are many fundamental differences between how indexes get created with in-memory tables when compared to standard tables. Index maintenance, defragmenting indexes, is not something you have to take into account. However, you do need to worry about statistics of in-memory tables. In-memory indexes maintain statistics that will need to be updated. You'll also want information about the in-memory indexes such as whether they're being accessed using scans or seeks. While the desire to track all this is the same, the mechanisms for doing so are different.

You can't actually see the statistics on hash indexes. You can run `DBCC SHOW_STATISTICS` against the index, but the output looks like Figure 24-10.

Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows	Persisted Sample Percent
1 PK_Address_091C2A1A21648C8B	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
All density   Average Length   Columns										
RANGE_HI_KEY   RANGE_ROWS   EQ_ROWS   DISTINCT_RANGE_ROWS   AVG_RANGE_ROWS										

**Figure 24-10.** The empty output of statistics on an in-memory index

This means there is no way to look at the statistics of the in-memory indexes. You can check the statistics of any nonclustered index. Whether you can see the statistics or not, those statistics will still get out-of-date as your data changes. Statistics are automatically maintained in SQL Server 2016 and newer for in-memory tables and indexes. The rules are the same as for disk-based statistics. SQL Server 2014 does not have automatic statistics maintenance, so you will have to use manual methods.

You can use `sp_updatestats`. The current version of the procedure is completely aware of in-memory indexes and their differences. You can also use `UPDATE STATISTICS`, but in SQL Server 2014, you must use `FULLSCAN` or `RESAMPLE` along with `NORECOMPUTE` as follows:

```
UPDATE STATISTICS dbo.Address WITH FULLSCAN, NORECOMPUTE;
```

If you don't use this syntax, it appears that you're attempting to alter the statistics on the in-memory table, and you can't do that. You'll be presented with a pretty clear error.

Msg 41346, Level 16, State 2, Line 1  
CREATE and UPDATE STATISTICS for memory optimized tables requires the WITH FULLSCAN or RESAMPLE and the NORECOMPUTE options. The WHERE clause is not supported.

Defining the sampling as either FULLSCAN or RESAMPLE and then letting it know that you're not attempting to turn on automatic update by using NORECOMPUTE, the statistics will get updated.

In SQL Server 2016 and greater, you can control the sampling methods as you would with other statistics.

## Natively Compiled Stored Procedures

Just getting the table into memory and radically reducing the locking contention with the optimistic approaches results in impressive performance improvements. To really make things move quickly, you can also implement the new feature of compiling stored procedures into a DLL that runs within the SQL Server executable. This really makes the performance scream. The syntax is straightforward. This is how you could take the query from before and compile it:

```
CREATE PROC dbo.AddressDetails @City NVARCHAR(30)
    WITH NATIVE_COMPILATION,
        SCHEMABINDING,
        EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'us_english')
    SELECT a.AddressLine1,
        a.City,
        a.PostalCode,
        sp.Name AS StateProvinceName,
        cr.Name AS CountryName
    FROM dbo.Address AS a
        JOIN dbo.StateProvince AS sp
            ON sp.StateProvinceID = a.StateProvinceID
        JOIN dbo.CountryRegion AS cr
            ON cr.CountryRegionCode = sp.CountryRegionCode
    WHERE a.City = @City;
END
```

Unfortunately, if you attempt to run this query definition as currently defined, you’ll receive the following error:

Msg 10775, Level 16, State 1, Procedure AddressDetails, Line 7 [Batch Start Line 5013]  
Object 'dbo.CountryRegion' is not a memory optimized table or a natively compiled inline table-valued function and cannot be accessed from a natively compiled module.

While you can query a mix of in-memory and standard tables, you can only create natively compiled stored procedures against in-memory tables. I’m going to use the same methods shown previously to load the `dbo.CountryRegion` table into memory and then run the script again. This time it will compile successfully. If you then execute the query using `@City = 'Walla Walla'` as before, the execution time won’t even register inside SSMS. You have to capture the event through Extended Events, as shown in Figure 24-11.

	name	batch_text	duration	logical_reads	row_count
	sql_batch_completed	EXEC dbo.AddressDetails 'Walla Wall...	451	0	0

**Figure 24-11.** Extended Events showing the execution time of a natively compiled procedure

The execution time there is not in milliseconds but microseconds. So, the query execution time has gone from the native run time of 3.7ms down to the in-memory run time of 800 microseconds and then finally 451 microseconds. That’s a pretty hefty performance improvement.

But, there are restrictions. As was already noted, you have to be referencing only in-memory tables. The parameter values assigned to the procedures cannot accept NULL values. If you choose to set a parameter to NOT NULL, you must also supply an initial value. Otherwise, all parameters are required. You must enforce schema binding with the underlying tables. Finally, you need to have the procedures exist with an ATOMIC BLOCK. An atomic blocks require that all statements within the transaction succeed or all statements within the transaction will be rolled back.

Here are another couple of interesting points about the natively compiled procedures. You can retrieve only an estimated execution plan, not an actual plan. If you turn on actual plans in SSMS and then execute the query, nothing appears. But, if you

request an estimated plan, you can retrieve one. Figure 24-12 shows the estimated plan for the procedure created earlier.



**Figure 24-12.** Estimated execution plan for a natively compiled procedure

You'll note that it looks largely like a regular execution plan, but there are quite a few differences behind the scenes. If you click the SELECT operator, you don't have nearly as many properties. Compare the two sets of data from the compiled stored procedure shown earlier and the properties of the regular query run earlier in Figure 24-13.

Misc		Misc	
Estimated Operator Cost	0 (0%)	Cached plan size	104 KB
Estimated Subtree Cost	0	CardinalityEstimationModelVersion	140
NonParallelPlanReason	NoParallelForNativelyCompiledModule	CompileCPU	10
Statement	SELECT a.AddressLine1, a.City, a.	CompileMemory	504
		CompileTime	10
		Estimated Number of Rows	99,7053
		Estimated Operator Cost	0 (0%)
		Estimated Subtree Cost	0.0081119
		MemoryGrantInfo	
		Optimization Level	FULL
		OptimizerHardwareDependentProperties	
		OptimizerStatsUsage	
		QueryHash	0x6FCCF8E8363DA62D
		QueryPlanHash	0x10A7FEBD96D7A5C3
		Reason For Early Termination Of Statement Optimiz	Time Out
		RetrievedFromCache	true
		SecurityPolicyApplied	False
		Set Options	ANSI_NULLS: True, ANSI_PADDING
		Statement	SELECT a.AddressLine1, a.C
		TraceFlags	

**Figure 24-13.** SELECT operator properties from two different execution plans

Much of the information you would expect to see is gone because the natively compiled procedures just don't operate in the same way as the other queries. The use of execution plans to determine the behavior of these queries is absolutely as valuable here as it was with standard queries, but the internals are going to be different.

## Recommendations

While the in-memory tables and natively compiled procedures can result in radical improvements in performance within SQL Server, you're still going to want to evaluate whether their use is warranted in your situation. The limits imposed on the behavior of these objects means they are not going to be useful in all circumstances. Further, because of the requirements on both hardware and on the need for an enterprise-level installation of SQL Server, many just won't be able to implement these new objects and their behavior. To determine whether your workload is a good candidate for the use of these new objects, you can do a number of things.

## Baselines

You should already be planning on establishing a performance baseline of your system by gathering various metrics using Performance Monitor, the dynamic management objects, Extended Events, and all the other tools at your disposal. Once you have the baseline, you can make determinations if your workload is likely to benefit from the reduced locking and increased speed of the in-memory tables.

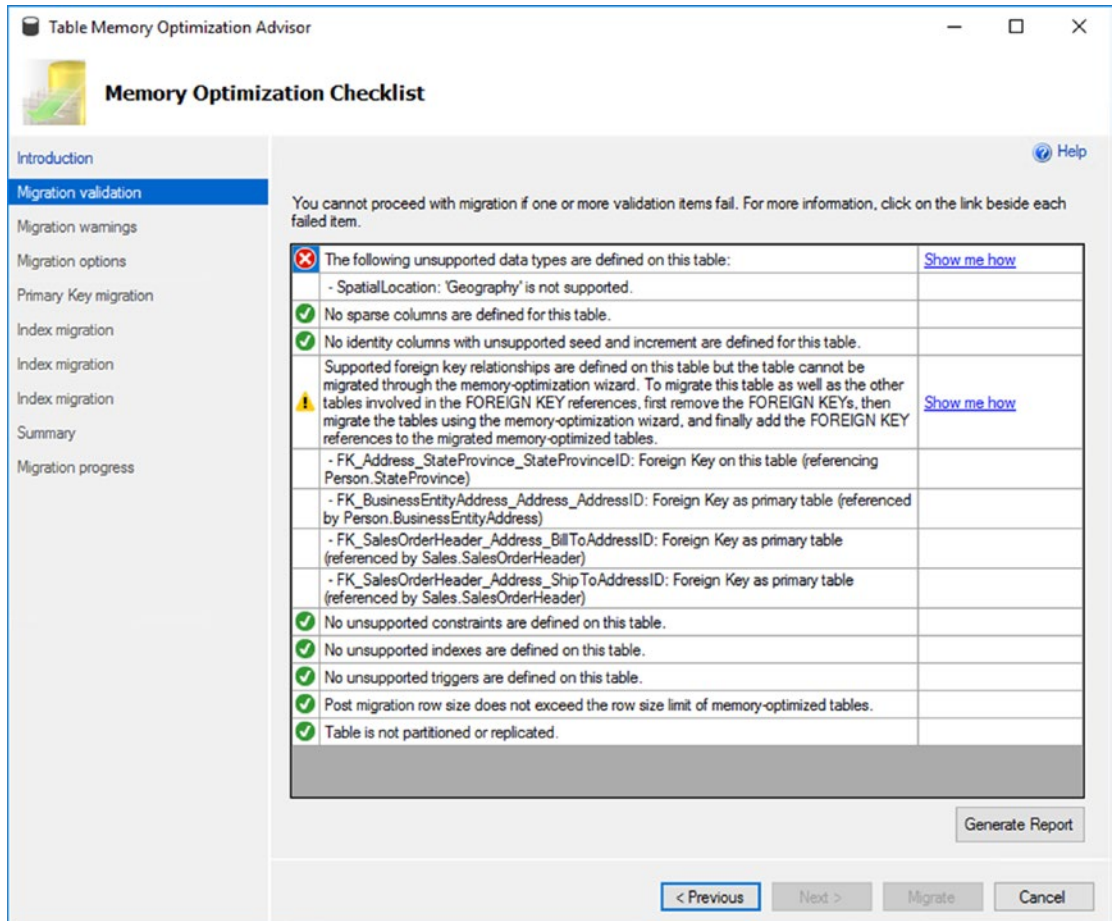
## Correct Workload

This technology is called in-memory OLTP tables for a reason. If you are dealing with a system that is primarily read focused, has only nightly or intermittent loads, or has a very low level of online transaction processing as its workload, the in-memory tables and natively compiled procedures are unlikely to be a major benefit for you. If you're dealing with a lot of latency in your system, the in-memory tables could be a good solution. Microsoft has outlined several other potentially beneficial workloads that you could consider using in-memory tables and natively compiled procedures; see Books Online (<http://bit.ly/1r6dmKY>).

## Memory Optimization Advisor

To quickly and easily determine whether a table is a good candidate for moving to in-memory storage, Microsoft has supplied a tool within SSMS. If you use the Object Explorer to navigate to a particular table, you can right-click that table and select Memory Optimization Advisor from the context menu. That will open a wizard. If I select

the Person.Address table that I manually migrated earlier, the initial check will find all the columns that are not supported within the in-memory table. That will stop the wizard, and no other options are available. The output looks like Figure 24-14.












**Figure 24-14.** Table Memory Optimization Advisor showing all the unsupported data types

That means this table, as currently structured, would not be a candidate for moving to in-memory storage. So that you can see a clean run-through of the tool, I'll create a clean copy of the table in the InMemoryTest database created earlier, shown here:

```
USE InMemoryTest;
GO

CREATE TABLE dbo.AddressStaging
(
    AddressID INT NOT NULL
                IDENTITY(1, 1)
                PRIMARY KEY,
    AddressLine1 NVARCHAR(60) NOT NULL,
    AddressLine2 NVARCHAR(60) NULL,
    City NVARCHAR(30) NOT NULL,
    StateProvinceID INT NOT NULL,
    PostalCode NVARCHAR(15) NOT NULL
);
```

Now, running the Memory Optimization Advisor has completely different results in the first step, as shown in Figure 24-15.

	No unsupported data types are defined on this table.	
	No sparse columns are defined for this table.	
	No identity columns with unsupported seed and increment are defined for this table.	
	No foreign key relationships are defined on this table.	
	No unsupported constraints are defined on this table.	
	No unsupported indexes are defined on this table.	
	No unsupported triggers are defined on this table.	
	Post migration row size does not exceed the row size limit of memory-optimized tables.	
	Table is not partitioned or replicated.	

**Figure 24-15.** Successful first check of the Memory Optimization Advisor

The next step in the wizard shows a fairly standard set of warnings about the differences that using the in-memory tables will cause in your T-SQL as well as links to further reading about these limitations. It's a useful reminder that you may have to address your code should you choose to migrate this table to in-memory storage. You can see that in Figure 24-16.








	A user transaction that accesses memory-optimized tables cannot access more than one user database.	<a href="#">More information</a>
	The following table hints are not supported on memory-optimized tables: HOLDLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, ROWLOCK, TABLOCK, TABLOCKX, UPDLOCK, XLOCK, NOWAIT.	<a href="#">More information</a>
	TRUNCATE TABLE and MERGE statements cannot target a memory-optimized table.	<a href="#">More information</a>
	Dynamic and Keyset cursors are automatically downgraded to a static cursor when pointing to a memory-optimized table.	<a href="#">More information</a>
	Some database-level features are not supported for use with memory-optimized tables. For details on these features, please refer to the help link.	<a href="#">More information</a>

Figure 24-16. Data migration warnings

You can stop there and click the Report button to generate a report of the check that was run against your table. Or, you can use the wizard to actually move the table into memory. Clicking Next from the Warnings page will open an Options page where you can determine how the table will be migrated into memory. You get to choose what the old table will be named. It assumes you'll be keeping the table name the same for the in-memory table. Several other options are available, as shown in Figure 24-17.

Specify options for memory optimization:

Memory-optimized filegroup:

InMemoryTest\_InMemoryData

Logical file name:

InMemoryTest\_InMemoryData.ndf

File path:

c:\Data

...

Rename the original table as:

AddressStaging\_old

Estimated current memory cost (MB):

0

☐ Also copy table data to the new memory optimized table.

By default, this table will be migrated to a memory-optimized table with both schema and data durability.

☐ Check this box to migrate this table to a memory-optimized table with no data durability.

Figure 24-17. Setting the options for migrating the standard table to in-memory



Clicking Next you get to determine how you’re going to create the primary key for the table. You get to supply it with a name. Then you have to choose if you’re going with a nonclustered hash or a nonclustered index. If you choose the nonclustered hash, you will have to provide a bucket count. Figure 24-18 shows how I configured the key in much the same way as I did it earlier using T-SQL.

Please choose the appropriate conversion for this primary key:

Column	Type
<input checked="" type="checkbox"/> AddressID	int

Select a new name for this primary key:

Select the type of this primary key:

☒ Use NONCLUSTERED HASH index

A NONCLUSTERED HASH index provides the most benefit for point lookups. It provides no discernible benefit if a query is running a Range scan.

Bucket Count:

The Bucket Count of a NONCLUSTERED HASH index is the number of buckets in the hash table. It is recommended to set the fill factor to 50 to 60% if the table requires a lot of space for growth. Bucket Count will be rounded up to the nearest power of two.

☐ Use NONCLUSTERED index



A NONCLUSTERED index provides the most benefit for range predicates and ORDER BY clauses. NONCLUSTERED indexes are unidirectional. It provides no benefit for ORDER BY clauses with orders different from the index.

Sort column and order:

Column	Sort Order
--------	------------

**Figure 24-18.** Choosing the configuration of the primary key of the new in-memory table

Clicking Next will show you a summary of the choices you have made and enable a button at the bottom of the screen to immediately migrate the table. It will migrate the table, renaming the old table however it was told to, and it will migrate the data if you chose that option. The output of a successful migration looks like Figure 24-19.

	Action	Result
	Renaming the original table.	Passed
	New name:AddressStaging_old	
	Creating the memory-optimized table in the database.	Passed
	Adding index:AddressStaging_primaryKey	

**Figure 24-19.** A successful in-memory table migration using the wizard

The Memory Optimization Advisor can then identify which tables can physically be moved into memory and can do that work for you. But, it doesn't have the judgment to know which tables should be moved into memory. You're still going to have to think that through on your own.

## Native Compilation Advisor

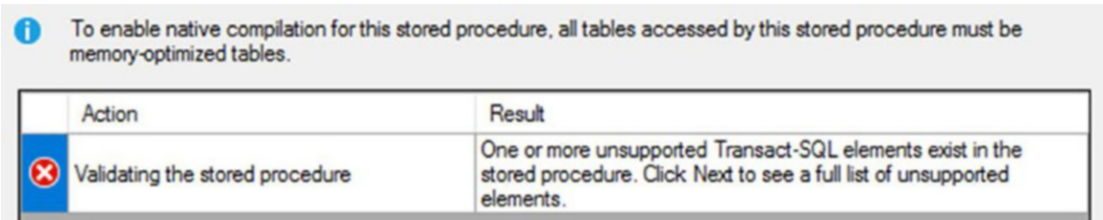
Similar in function to the Memory Optimization Advisor, the Native Compilation Advisor can be run against an existing stored procedure to determine whether it can be compiled natively. However, it's much simpler in function than the prior wizard. To show it in action, I'm going to create two different procedures, shown here:

```
CREATE OR ALTER PROCEDURE dbo.FailWizard (@City NVARCHAR(30))
AS
SELECT a.AddressLine1,
       a.City,
       a.PostalCode,
       sp.Name AS StateProvinceName,
       cr.Name AS CountryName
FROM dbo.Address AS a
     JOIN dbo.StateProvince AS sp
       ON sp.StateProvinceID = a.StateProvinceID
     JOIN dbo.CountryRegion AS cr WITH (NOLOCK)
```

```
        ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.City = @City;
GO

CREATE OR ALTER PROCEDURE dbo.PassWizard (@City NVARCHAR(30))
AS
SELECT a.AddressLine1,
       a.City,
       a.PostalCode,
       sp.Name AS StateProvinceName,
       cr.Name AS CountryName
FROM dbo.Address AS a
      JOIN dbo.StateProvince AS sp
        ON sp.StateProvinceID = a.StateProvinceID
      JOIN dbo.CountryRegion AS cr
        ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.City = @City;
GO
```

The first procedure includes a NOLOCK hint that can't be run against in-memory tables. The second procedure is just a repeat of the procedure you've been working with throughout this chapter. After executing the script to create both procedures, I can access the Native Compilation Advisor by right-clicking the stored procedure `dbo.FailWizard` and selecting Native Compilation Advisor from the context menu. After getting past the wizard start screen, the first step identifies a problem with the procedure, as shown in Figure 24-20.



**Figure 24-20.** The Native Compilation Advisor has identified inappropriate T-SQL syntax

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

## 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/2mNB1hg>. 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

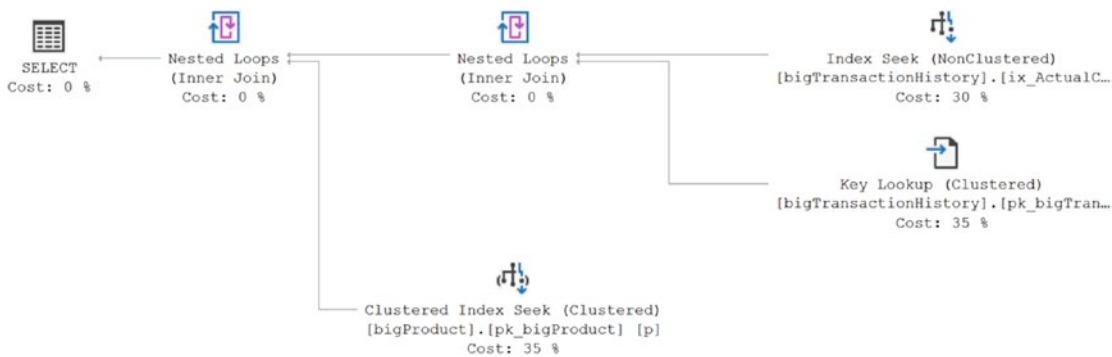
--remove the plan from cache
DECLARE @PlanHandle VARBINARY(64);
SELECT @PlanHandle = deps.plan_handle
FROM sys.dm_exec_procedure_stats AS deps
WHERE deps.object_id = OBJECT_ID('dbo.ProductByCost');
IF @PlanHandle IS NOT NULL
    BEGIN
        DBCC FREEPROCCACHE(@PlanHandle);
    END
GO

--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
```

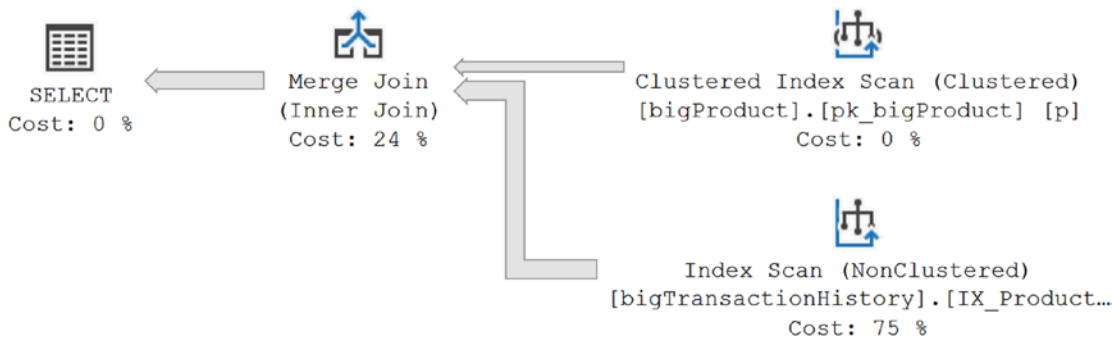
This will take a while to execute. Once it's complete, we should have a tuning recommendation in our database. Referring to the previous listing, we established a particular behavior in our query by executing it at least 30 times. The query itself returns just a single row of data when the value 8.2205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





**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.



**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":"Automatic Tun...	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	2.180373266666667e+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](#).

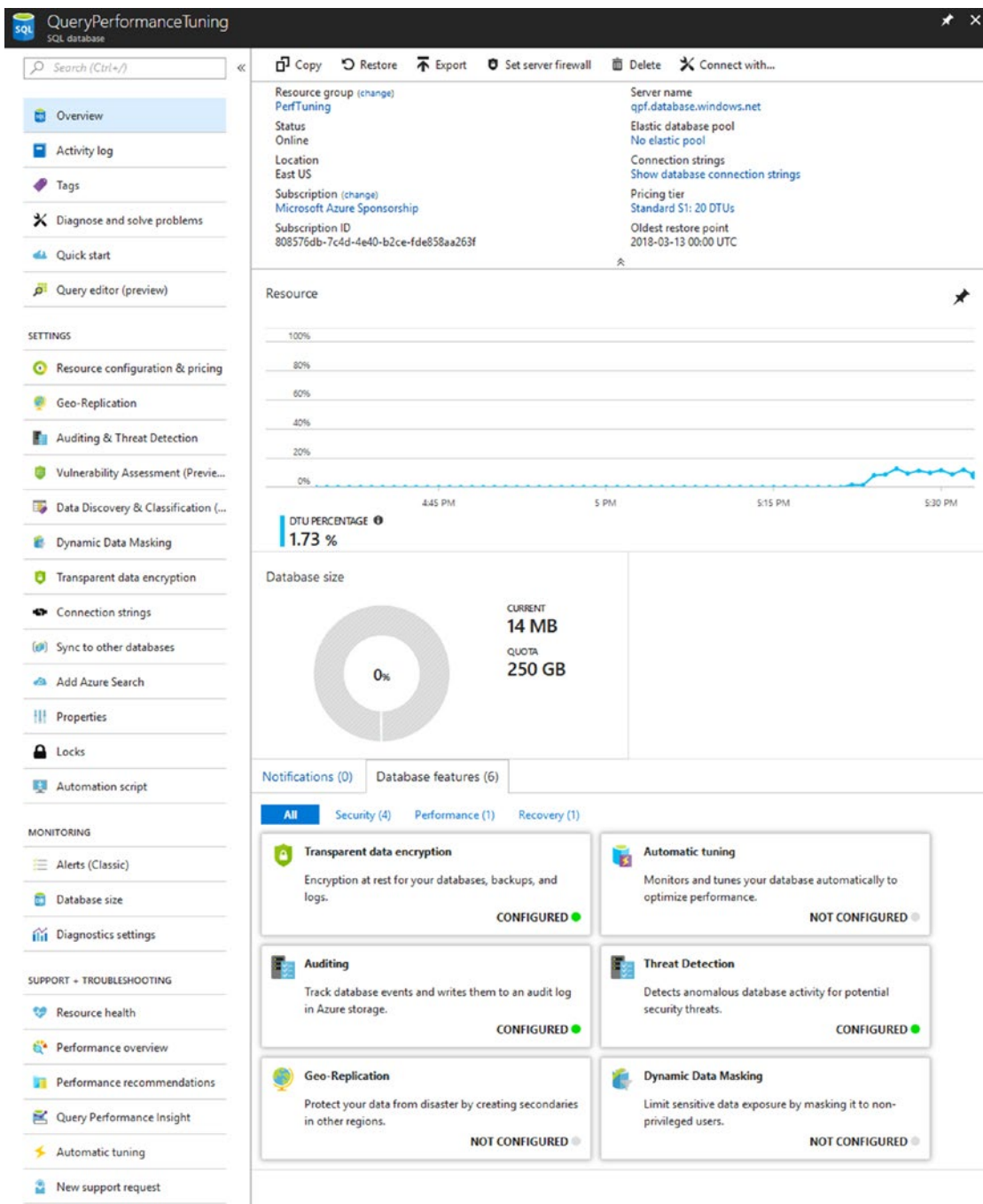
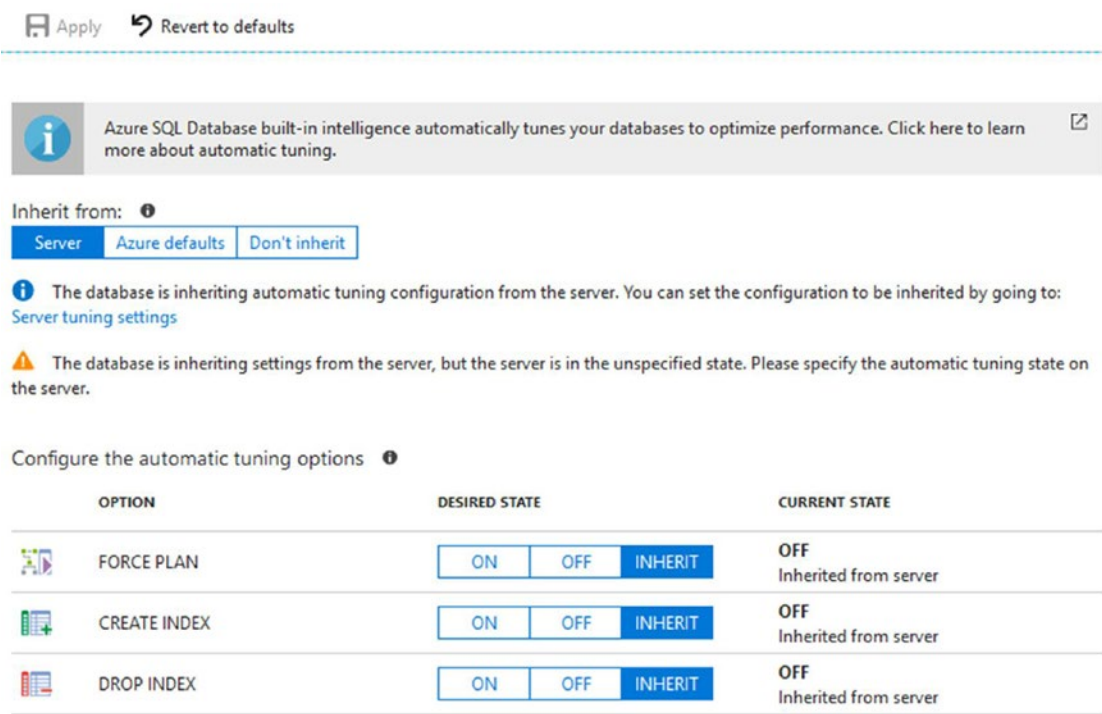


Figure 25-4. Database blade on Azure portal

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.



**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.

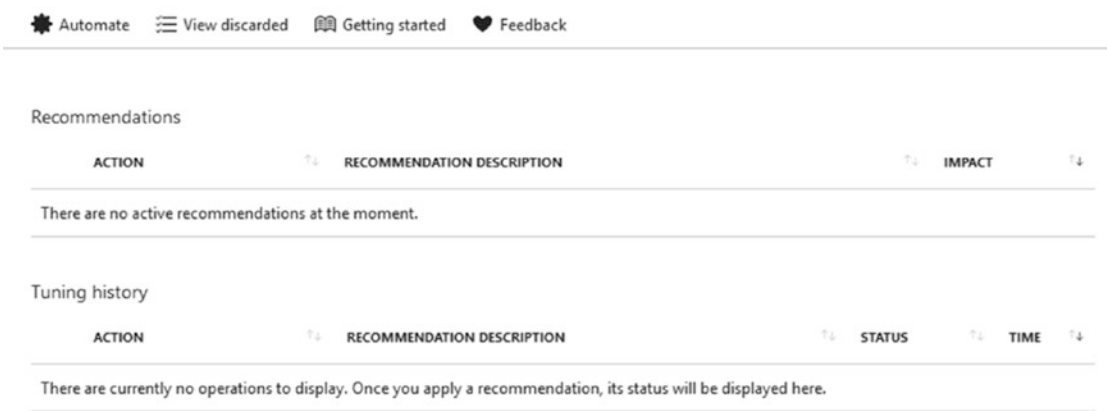


**Figure 25-6.** Automatic tuning options change to FORCE PLAN as ON



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.



**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.