## Forward-Only Cursors

These are the characteristics of forward-only cursors:

- They operate directly on the base tables.
- Rows from the underlying tables are usually not retrieved until the cursor rows are fetched using the cursor `FETCH` operation. However, the database API forward-only cursor type, with the following additional characteristics, retrieves all the rows from the underlying table first:
  - Client-side cursor location
  - Server-side cursor location and read-only cursor concurrency
- They support forward scrolling only (`FETCH NEXT`) through the cursor.
- They allow all changes (`INSERT`, `UPDATE`, and `DELETE`) through the cursor. Also, these cursors reflect all changes made to the underlying tables.

The forward-only characteristic is implemented differently by the database API cursors and the T-SQL cursor. The data access layers implement the forward-only cursor characteristic as one of the four previously listed cursor types. But the T-SQL cursor doesn't implement the forward-only cursor characteristic as a cursor type; rather, it implements it as a property that defines the scrollable behavior of the cursor. Thus, for a T-SQL cursor, the forward-only characteristic can be used to define the scrollable behavior of one of the remaining three cursor types.

The T-SQL syntax provides a specific cursor type option, `FAST_FORWARD`, to create a fast-forward-only cursor. The nickname for the `FAST_FORWARD` cursor is the *fire hose* because it is the fastest way to move data through a cursor and because all the information flows one way. However, don't be surprised when the "firehose" is still not as fast as traditional set-based operations. The following T-SQL statement creates a fast-forward-only T-SQL cursor:

```
DECLARE MyCursor CURSOR FAST_FORWARD FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

The FAST_FORWARD property specifies a forward-only, read-only cursor with performance optimizations enabled.

## Static Cursors

These are the characteristics of static cursors:

- They create a snapshot of cursor results in the tempdb database when the cursor is opened. Thereafter, static cursors operate on the snapshot in the tempdb database.
- Data is retrieved from the underlying tables when the cursor is opened.
- Static cursors support all scrolling options: FETCH FIRST, FETCH NEXT, FETCH PRIOR, FETCH LAST, FETCH ABSOLUTE n, and FETCH RELATIVE n.
- Static cursors are always read-only; data modifications are not allowed through static cursors. Also, changes (INSERT, UPDATE, and DELETE) made to the underlying tables are not reflected in the cursor.

The following T-SQL statement creates a static T-SQL cursor:

```
DECLARE MyCursor CURSOR STATIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

Some tests show that a static cursor can perform as well as—and sometimes faster than—a forward-only cursor. Be sure to test this behavior on your own system in situations where you must use a cursor.

## Keyset-Driven Cursors

These are the characteristics of keyset-driven cursors:

- Keyset cursors are controlled by a set of unique identifiers (or keys) known as a *keyset.* The keyset is built from a set of columns that uniquely identify the rows in the result set.
- These cursors create the keyset of rows in the tempdb database when the cursor is opened.

- Membership of rows in the cursor is limited to the keyset of rows created in the tempdb database when the cursor is opened.
- On fetching a cursor row, the database engine first looks at the keyset of rows in tempdb and then navigates to the corresponding data row in the underlying tables to retrieve the remaining columns.
- They support all scrolling options.
- Keyset cursors allow all changes through the cursor. An INSERT performed outside the cursor is not reflected in the cursor since the membership of rows in the cursor is limited to the keyset of rows created in the tempdb database on opening the cursor. An INSERT through the cursor appears at the end of the cursor. A DELETE performed on the underlying tables raises an error when the cursor navigation reaches the deleted row. An UPDATE on the nonkeyset columns of the underlying tables is reflected in the cursor. An UPDATE on the keyset columns is treated like a DELETE of an old key value and the INSERT of a new key value. If a change disqualifies a row for membership or affects the order of a row, then the row does not disappear or move unless the cursor is closed and reopened.

The following T-SQL statement creates a keyset-driven T-SQL cursor:

```
DECLARE MyCursor CURSOR KEYSET FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

## Dynamic Cursors

These are the characteristics of dynamic cursors:

- Dynamic cursors operate directly on the base tables.
- The membership of rows in the cursor is not fixed since they operate directly on the base tables.
- As with forward-only cursors, rows from the underlying tables are not retrieved until the cursor rows are fetched using a cursor FETCH operation.

- Dynamic cursors support all scrolling options except `FETCH ABSOLUTE n`, since the membership of rows in the cursor is not fixed.
- These cursors allow all changes through the cursor. Also, all changes made to the underlying tables are reflected in the cursor.
- Dynamic cursors don't support all properties and methods implemented by the database API cursors. Properties such as `AbsolutePosition`, `Bookmark`, and `RecordCount`, as well as methods such as `clone` and `Resync`, are not supported by dynamic cursors. Instead, they are supported by keyset-driven cursors.

The following T-SQL statement creates a dynamic T-SQL cursor:

```
DECLARE MyCursor CURSOR DYNAMIC FOR
SELECT adt.Name
FROM Person.AddressType AS adt
WHERE adt.AddressTypeID = 1;
```

The dynamic cursor is absolutely the slowest possible cursor to use in all situations. It takes more locks and holds them longer, which radically increases its poor performance. Take this into account when designing your system.

## Cursor Cost Comparison

Now that you've seen the different cursor flavors, let's look at their costs. If you must use a cursor, you should always use the lightest-weight cursor that meets the requirements of your application. The cost comparisons among the different characteristics of the cursors are detailed next.

## Cost Comparison on Cursor Location

The client-side and server-side cursors have their own cost benefits and overhead, as explained in the sections that follow.

## Client-Side Cursors

Client-side cursors have the following cost benefits compared to server-side cursors:

- *Higher scalability*: Since the cursor metadata is maintained on the individual client machines connected to the server, the overhead of maintaining the cursor metadata is taken up by the client machines. Consequently, the ability to serve a larger number of users is not limited by the server resources.
- *Fewer network round-trips*: Since the result set returned by the `SELECT` statement is passed to the client where the cursor is maintained, extra network round-trips to the server are not required while retrieving rows from the cursor.
- *Faster scrolling*: Since the cursor is maintained locally on the client machine, it's potentially faster to walk through the rows of the cursor.
- *Highly portable*: Since the cursor is implemented using data access layers, it works across a large range of databases: SQL Server, Oracle, Sybase, and so forth.

Client-side cursors have the following cost overhead or drawbacks:

- *Higher pressure on client resources*: Since the cursor is managed at the client side, it increases pressure on the client resources. But it may not be all that bad, considering that most of the time the client applications are web applications and scaling out web applications (or web servers) is quite easy using standard load-balancing solutions. On the other hand, scaling out a transactional SQL Server database is still an art!
- *Support for limited cursor types*: Dynamic and keyset-driven cursors are not supported.

- *Only one active cursor-based statement on one connection*: As many rows of the result set as the client network can buffer are arranged in the form of network packets and sent to the client application. Therefore, until all the cursor's rows are fetched by the application, the database connection remains busy, pushing the rows to the client. During this period, other cursor-based statements cannot use the connection. This is negated by taking advantage of multiple active result sets (MARS), which would allow a connection to have a second active cursor.

## Server-Side Cursors

Server-side cursors have the following cost benefits:

- *Multiple active cursor-based statements on one connection*: While using server-side cursors, no results are left outstanding on the connection between the cursor operations. This frees the connection, allowing the use of multiple cursor-based statements on one connection at the same time. In the case of client-side cursors, as explained previously, the connection remains busy until all the cursor rows are fetched by the application. This means they cannot be used simultaneously by multiple cursor-based statements.
- *Row processing near the data*: If the row processing involves joining with other tables and a considerable amount of set operations, then it is advantageous to perform the row processing near the data using a server-side cursor.
- *Less pressure on client resources*: It reduces pressure on the client resources. But this may not be that desirable because, if the server resources are maxed out (instead of the client resources), then it will require scaling out the database, which is a difficult proposition.
- *Support for all cursor types*: Client-side cursors have limitations on which types of cursors can be supported. There are no limits on the server-side cursors.

Server-side cursors have the following cost overhead or disadvantages:

- *Lower scalability*: They make the server less scalable since server resources are consumed to manage the cursor.
- *More network round-trips*: They increase network round-trips if the cursor row processing is done in the client application. The number of network round-trips can be optimized by processing the cursor rows in the stored procedure or by using the cache size feature of the data access layer.
- *Less portable*: Server-side cursors implemented using T-SQL cursors are not readily portable to other databases because the syntax of the database code managing the cursor is different across databases.

## Cost Comparison on Cursor Concurrency

As expected, cursors with a higher concurrency model create the least amount of blocking in the database and support higher scalability, as explained in the following sections.

### Read-Only

The read-only concurrency model provides the following cost benefits:

- *Lowest locking overhead*: The read-only concurrency model introduces the least locking and synchronization overhead on the database. Since (S) locks are not held on the underlying row after a cursor row is fetched, other users are not blocked from accessing the row. Furthermore, the (S) lock acquired on the underlying row while fetching the cursor row can be avoided by using the `NO_LOCK` locking hint in the `SELECT` statement of the cursor, but only if you don't care about what kind of data you get back because of dirty reads.
- *Highest concurrency*: Since additional locks are not held on the underlying rows, the read-only cursor doesn't block other users from accessing the underlying tables. The shared lock is still acquired.

The main drawback of the read-only cursor is as follows:

- *Nonupdatable*: The content of underlying tables cannot be modified through the cursor.

## Optimistic

The optimistic concurrency model provides the following benefits:

- *Low locking overhead*: Similar to the read-only model, the optimistic concurrency model doesn't hold an (S) lock on the cursor row after the row is fetched. To further improve concurrency, the `NOLOCK` locking hint can also be used, as in the case of the read-only concurrency model. But, please know that `NOLOCK` can absolutely lead to incorrect data or missing or extra rows, so its use requires careful planning. Modification through the cursor to an underlying row requires exclusive rights on the row as required by an action query.
- *High concurrency*: Since only a shared lock is used on the underlying rows, the cursor doesn't block other users from accessing the underlying tables. But the modification through the cursor to an underlying row will block other users from accessing the row during the modification.

The following examples detail the cost overhead of the optimistic concurrency model:

- *Row versioning*: Since the optimistic concurrency model allows the cursor to be updatable, an additional cost is incurred to ensure that the current underlying row is first compared (using either version-based or value-based concurrency control) with the original cursor row fetched before applying a modification through the cursor. This prevents the modification through the cursor from accidentally overwriting the modification made by another user after the cursor row is fetched.

- *Concurrency control without a ROWVERSION column*: As explained previously, a `ROWVERSION` column in the underlying table allows the cursor to perform an efficient version-based concurrency control. In case the underlying table doesn't contain a `ROWVERSION` column, the cursor resorts to value-based concurrency control, which requires matching the current value of the row to the value when the row was read into the cursor. This increases the cost of the concurrency control. Both forms of concurrency control will cause additional overhead in tempdb.

## Scroll Locks

The major benefit of the scroll locks concurrency model is as follows:

- *Simple concurrency control*: By locking the underlying row corresponding to the last fetched row from the cursor, the cursor assures that the underlying row can't be modified by another user. This eliminates the versioning overhead of optimistic locking. Also, since the row cannot be modified by another user, the application is relieved from checking for a row-mismatch error.

The scroll locks concurrency model incurs the following cost overhead:

- *Highest locking overhead*: The scroll locks concurrency model introduces a pessimistic locking characteristic. A (U) lock is held on the last cursor row fetched, until another cursor row is fetched or the cursor is closed.
- *Lowest concurrency*: Since a (U) lock is held on the underlying row, all other users requesting a (U) or an (X) lock on the underlying row will be blocked. This can significantly hurt concurrency. Therefore, please avoid using this cursor concurrency model unless absolutely necessary.

## Cost Comparison on Cursor Type

Each of the basic four cursor types mentioned in the “Cursor Fundamentals” section earlier in the chapter incurs a different cost overhead on the server. Choosing an incorrect cursor type can hurt database performance. Besides the four basic cursor types, a fast-forward-only cursor (a variation of the forward-only cursor) is provided to enhance performance. The cost overhead of these cursor types is explained in the sections that follow.

### Forward-Only Cursors

These are the cost benefits of forward-only cursors:

- *Lower cursor open cost than static and keyset-driven cursors*: Since the cursor rows are not retrieved from the underlying tables and are not copied into the tempdb database during cursor open, the forward-only T-SQL cursor opens quickly. Similarly, the forward-only, server-side API cursors with optimistic/scroll locks concurrency open quickly since they do not retrieve the rows during cursor open.
- *Lower scroll overhead*: Since only `FETCH NEXT` can be performed on this cursor type, it requires less overhead to support different scroll operations.
- *Lower impact on the tempdb database than static and keyset-driven cursors*: Since the forward-only T-SQL cursor doesn’t copy the rows from the underlying tables into the tempdb database, no additional pressure is created on the database.

The forward-only cursor type has the following drawbacks:

- *Lower concurrency*: Every time a cursor row is fetched, the corresponding underlying row is accessed with a lock request depending on the cursor concurrency model (as noted earlier in the discussion about concurrency). It can block other users from accessing the resource.
- *No backward scrolling*: Applications requiring two-way scrolling can’t use this cursor type. But if the applications are designed properly, then it isn’t difficult to live without backward scrolling.

## Fast-Forward-Only Cursor

The fast-forward-only cursor is the fastest and least expensive cursor type. This forward-only and read-only cursor is specially optimized for performance. Because of this, you should always prefer it to the other SQL Server cursor types.

Furthermore, the data access layer provides a fast-forward-only cursor on the client side. That type of cursor uses a so-called *default result set* to make cursor overhead almost disappear.

---

**Note** The default result set is explained later in the chapter in the "Default Result Set" section.

---

## Static Cursors

These are the cost benefits of static cursors:

- *Lower fetch cost than other cursor types*: Since a snapshot is created in the tempdb database from the underlying rows on opening the cursor, the cursor row fetch is targeted to the snapshot instead of the underlying rows. This avoids the lock overhead that would otherwise be required to fetch the cursor rows.
- *No blocking on underlying rows*: Since the snapshot is created in the tempdb database, other users trying to access the underlying rows are not blocked.

On the downside, the static cursor has the following cost overhead:

- *Higher open cost than other cursor types*: The cursor open operation of the static cursor is slower than that of other cursor types since all the rows of the result set have to be retrieved from the underlying tables and the snapshot has to be created in the tempdb database during the cursor open.
- *Higher impact on tempdb than other cursor types*: There can be significant impact on server resources for creating, populating, and cleaning up the snapshot in the tempdb database.

## Keyset-Driven Cursors

These are the cost benefits of keyset-driven cursors:

- *Lower open cost than the static cursor*: Since only the keyset, not the complete snapshot, is created in the tempdb database, the keyset-driven cursor opens faster than the static cursor. SQL Server populates the keyset of a large keyset-driven cursor asynchronously, which shortens the time between when the cursor is opened and when the first cursor row is fetched.
- *Lower impact on tempdb than that with the static cursor*: Because the keyset-driven cursor is smaller, it uses less space in tempdb.

The cost overhead of keyset-driven cursors is as follows:

- *Higher open cost than forward-only and dynamic cursors*: Populating the keyset in the tempdb database makes the cursor open operation of the keyset-driven cursor costlier than that of forward-only (with the exceptions mentioned earlier) and dynamic cursors.
- *Higher fetch cost than other cursor types*: For every cursor row fetch, the key in the keyset has to be accessed first, and then the corresponding underlying row in the user database can be accessed. Accessing both the tempdb and the user database for every cursor row fetch makes the fetch operation costlier than that of other cursor types.
- *Higher impact on tempdb than forward-only and dynamic cursors*: Creating, populating, and cleaning up the keyset in tempdb impacts server resources.
- *Higher lock overhead and blocking than the static cursor*: Since row fetch from the cursor retrieves rows from the underlying table, it acquires an (S) lock on the underlying row (unless the `NOLOCK` locking hint is used) during the row fetch operation.

## Dynamic Cursor

The dynamic cursor has the following cost benefits:

- *Lower open cost than static and keyset-driven cursors*: Since the cursor is opened directly on the underlying rows without copying anything to the tempdb database, the dynamic cursor opens faster than the static and keyset-driven cursors.
- *Lower impact on tempdb than static and keyset-driven cursors*: Since nothing is copied into tempdb, the dynamic cursor places far less strain on tempdb than the other cursor types.

The dynamic cursor has the following cost overhead:

- *Higher lock overhead and blocking than the static cursor*: Every cursor row fetch in a dynamic cursor requeries the underlying tables involved in the `SELECT` statement of the cursor. The dynamic fetches are generally expensive because the original select condition might have to be reexecuted.

For a summary of the different cursors, their positives and negatives, please refer to Table 23-1.

***Table 23-1.*** *Comparing Cursors*

| Cursor Type | Positives | Negatives |
|---|---|---|
| *Forward-only* | Lower cost, lower scroll overhead, lower impact on tempdb | Lower concurrency, no backward scrolling |
| *Fast-forward-only* | Fastest cursor, lowest cost, lowest impact | No backward scrolling, no concurrency |
| *Static* | Lower fetch cost, no blocking, forward and backward scrolling | Higher open cost, higher impact on tempdb, no concurrency |
| *Keyset-driven* | Lower open cost, lower impact on tempdb, forward and backward scrolling, concurrency | Higher open cost, highest fetch cost, highest impact on tempdb, higher locking costs |
| *Dynamic* | Lower open cost, lower impact on tempdb, forward and backward scrolling, concurrency | Highest locking costs |

## Default Result Set

The default cursor type for the data access layers (ADO, OLEDB, and ODBC) is forward-only and read-only. The default cursor type created by the data access layers isn't a true cursor but a stream of data from the server to the client, generally referred to as the *default result set* or *fast-forward-only cursor* (created by the data access layer). In ADO.NET, the `DataReader` control has the forward-only and read-only properties, and it can be considered as the default result set in the ADO.NET environment. SQL Server uses this type of result set processing under the following conditions:

- The application, using the data access layers (ADO, OLEDB, ODBC), leaves all the cursor characteristics at the default settings, which requests a forward-only and read-only cursor.
- The application executes a `SELECT` statement instead of executing a `DECLARE CURSOR` statement.

**Note** Because SQL Server is designed to work with sets of data, not to walk through records one by one, the default result set is always faster than any other type of cursor.

The only request sent from the client to SQL Server is the SQL statement associated with the default cursor. SQL Server executes the query, organizes the rows of the result set in network packets (filling the packets as best it can), and then sends the packets to the client. These network packets are cached in the network buffers of the client. SQL Server sends as many rows of the result set to the client as the client-network buffers can cache. As the client application requests one row at a time, the data access layer on the client machine pulls the row from the client-network buffers and transfers it to the client application.

The following sections outline the benefits and drawbacks of the default result set.

## Benefits

The default result set is generally the best and most efficient way of returning rows from SQL Server for the following reasons:

- *Minimum network round-trips between the client and SQL Server*: Since the result set returned by SQL Server is cached in the client-network buffers, the client doesn't have to make a request across the network to get the individual rows. SQL Server puts most of the rows that it can in the network buffer and sends to the client as much as the client-network buffer can cache.
- *Minimum server overhead*: Since SQL Server doesn't have to store data on the server, this reduces server resource utilization.

## Multiple Active Result Sets

SQL Server 2005 introduced the concept of multiple active result sets, wherein a single connection can have more than one batch running at any given moment. In prior versions, a single result set had to be processed or closed out prior to submitting the next request. MARS allows multiple requests to be submitted at the same time through the same connection. MARS is enabled on SQL Server all the time. It is not enabled by a connection unless that connection explicitly calls for it. Transactions must be handled at the client level and have to be explicitly declared and committed or rolled back. With MARS in action, if a transaction is not committed on a given statement and the connection is closed, all other transactions that were part of that single connection will be rolled back. MARS is enabled through application connection properties.

## Drawbacks

While there are advantages to the default result set, there are drawbacks as well. Using the default result set requires some special conditions for maximum performance:

- *It doesn't support all properties and methods*: Properties such as `AbsolutePosition`, `Bookmark`, and `RecordCount`, as well as methods such as `Clone`, `MoveLast`, `MovePrevious`, and `Resync`, are not supported.
- *Locks may be held on the underlying resource*: SQL Server sends as many rows of the result set to the client as the client-network buffers can cache. If the size of the result set is large, then the client-network buffers may not be able to receive all the rows. SQL Server then holds a lock on the next page of the underlying tables, which has not been sent to the client.

To demonstrate these concepts, consider the following test table:

```
USE AdventureWorks2017;
GO
DROP TABLE IF EXISTS dbo.Test1;
GO

CREATE TABLE dbo.Test1 (C1 INT,
                        C2 CHAR(996));

CREATE CLUSTERED INDEX Test1Index ON dbo.Test1 (C1);

INSERT INTO dbo.Test1
VALUES (1, '1'),
       (2, '2');
GO
```

Now consider this PowerShell script, which accesses the rows of the test table using ADO with OLEDB and the default cursor type for the database API cursor (`ADODB.Recordset` object) as follows:

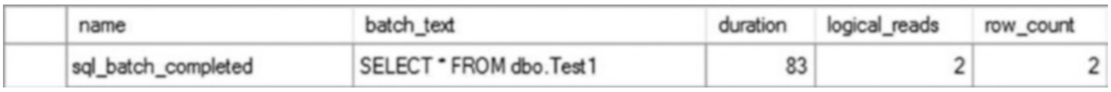
```
$AdoConn = New-Object -comobject ADODB.Connection
$AdoRecordset = New-Object -comobject ADODB.Recordset
```

```
##Change the Data Source to your server
$AdoConn.Open("Provider= SQLOLEDB; Data Source=DOJO\RANDORI; Initial
Catalog=AdventureWorks2017; Integrated Security=SSPI")
$AdoRecordset.Open("SELECT * FROM dbo.Test1", $AdoConn)

do {
    $C1 = $AdoRecordset.Fields.Item("C1").Value
    $C2 = $AdoRecordset.Fields.Item("C2").Value
    Write-Output "C1 = $C1 and C2 = $C2"
    $AdoRecordset.MoveNext()
    } until     ($AdoRecordset.EOF -eq $True)
$AdoRecordset.Close()
$AdoConn.Close()
```

This is not how you normally access databases from PowerShell, but it does show how a client-side cursor operates. Note that the table has two rows with the size of each row equal to 1,000 bytes (= 4 bytes for `INT` + 996 bytes for `CHAR(996)`) without considering the internal overhead. Therefore, the size of the complete result set returned by the `SELECT` statement is approximately 2,000 bytes (= 2 × 1,000 bytes).

On execution of the cursor open statement (`$AdoRecordset.Open()`), a default result set is created on the client machine running the code. The default result set holds as many rows as the client-network buffer can cache.

Since the size of the result set is small enough to be cached by the client-network buffer, all the cursor rows are cached on the client machine during the cursor open statement itself, without retaining any lock on the `dbo.Test1` table. You can verify the lock status for the connection using the `sys.dm_tran_locks` dynamic management view. During the complete cursor operation, the only request from the client to SQL Server is the `SELECT` statement associated to the cursor, as shown in the Extended Events output in Figure 23-1.



| | name | batch_text | duration | logical_reads | row_count |
|---|---|---|---|---|---|
| | sql_batch_completed | SELECT * FROM dbo.Test1 | 83 | 2 | 2 |

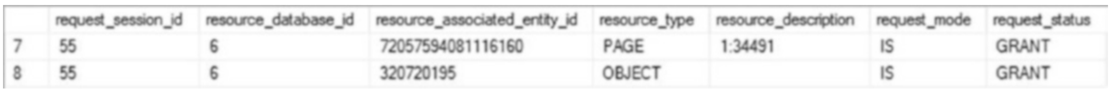
***Figure 23-1.** Profiler trace output showing database requests made by the default result set*

To find out the effect of a large result set on the default result set processing, let's add some more rows to the test table.

```
SELECT TOP 100000
       IDENTITY(INT, 1, 1) AS n
INTO #Tally
FROM master.dbo.syscolumns AS scl,
     master.dbo.syscolumns AS sc2;

INSERT INTO dbo.Test1 (C1,
                       C2)
SELECT n,
       n
FROM #Tally AS t;
GO
```

The additional rows generated by this example increase the size of the result set considerably. Depending on the size of the client-network buffer, only part of the result set can be cached. On execution of the `Ado.Recordset.Open` statement, the default result set on the client machine will get part of the result set, with SQL Server waiting on the other end of the network to send the remaining rows.

On my machine during this period, the locks shown in Figure 23-2 are held on the underlying `Test1` table as obtained from the output of `sys.dm_tran_locks`.



| | request_session_id | resource_database_id | resource_associated_entity_id | resource_type | resource_description | request_mode | request_status |
|---|---|---|---|---|---|---|---|
| 7 | 55 | 6 | 72057594081116160 | PAGE | 1:34491 | IS | GRANT |
| 8 | 55 | 6 | 320720195 | OBJECT | | IS | GRANT |

***Figure 23-2.*** *sys.dm_tran_locks output showing the locks held by the default result set while processing the large result set*

The (IS) lock on the table will block other users trying to acquire an (X) lock. To minimize the blocking issue, follow these recommendations:

- Process all rows of the default result set immediately.
- Keep the result set small. As demonstrated in the example, if the size of the result set is small, then the default result set will be able to read all the rows during the cursor open operation itself.

## 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.

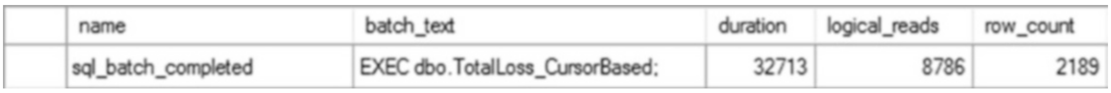
The `FOR EACH` phrase in the second point suggests that these application tasks could be served by a cursor. However, a `FOR`, `WHILE`, cursor, or any other kind of processing of this type can be dangerous within SQL Server. Despite the attraction that this approach holds, it is not set-based, and it 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:

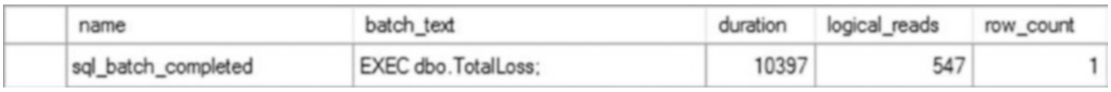
```
CREATE OR ALTER PROC dbo.TotalLoss
AS
SELECT CASE --Determine status based on following computation
           WHEN SUM(MoneyLostPerProduct) > 5000 THEN
               'We are bankrupt!'
           ELSE
               'We are safe!'
       END AS Status
```

```
FROM
( --Calculate total money lost for all discarded products
    SELECT SUM(wo.ScrappedQty * 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;
```

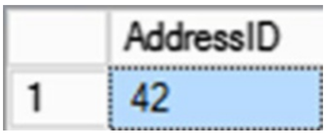
So, to experiment with some actual data in the database, go ahead and load the information stored in `Person.Address` in AdventureWorks2017 into the new table that's stored in-memory in this new database.

```
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;
```

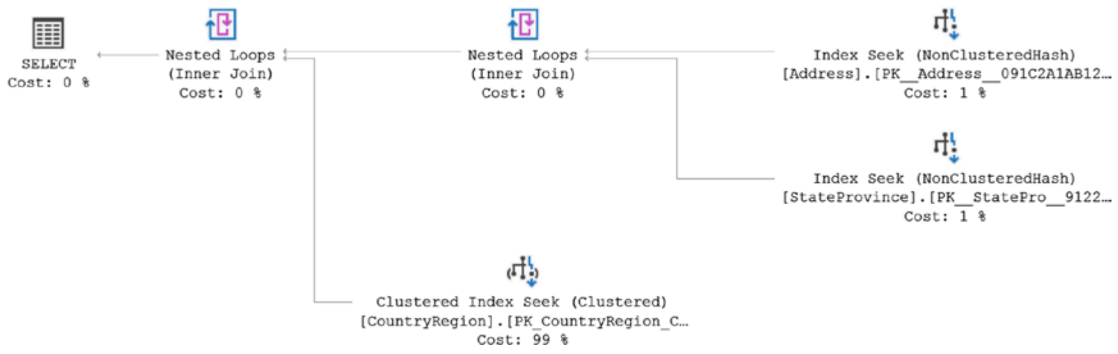
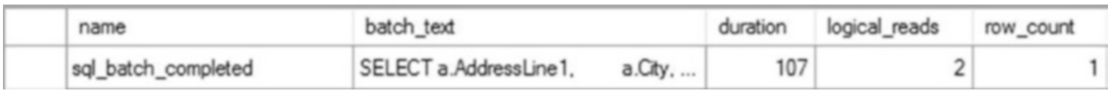

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClusteredHash)
[Address].[PK__Address__091C2A1AB12…
Cost: 1 %
Index Seek (NonClusteredHash)
[StateProvince].[PK__StatePro__9122…
Cost: 1 %
Clustered Index Seek (Clustered)
[CountryRegion].[PK_CountryRegion_C…
Cost: 99 %

***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.

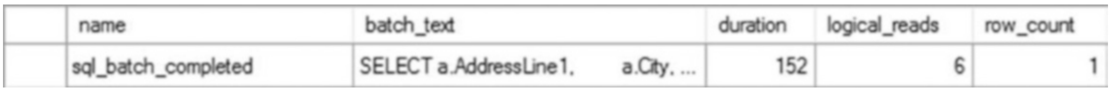
The principal performance enhancements come from the lack of locking and latching, allowing massive inserts and updates while simultaneously allowing for querying. But, the queries do run faster as well. The previous query resulted in the execution time and reads shown in Figure 24-3.



| | 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.

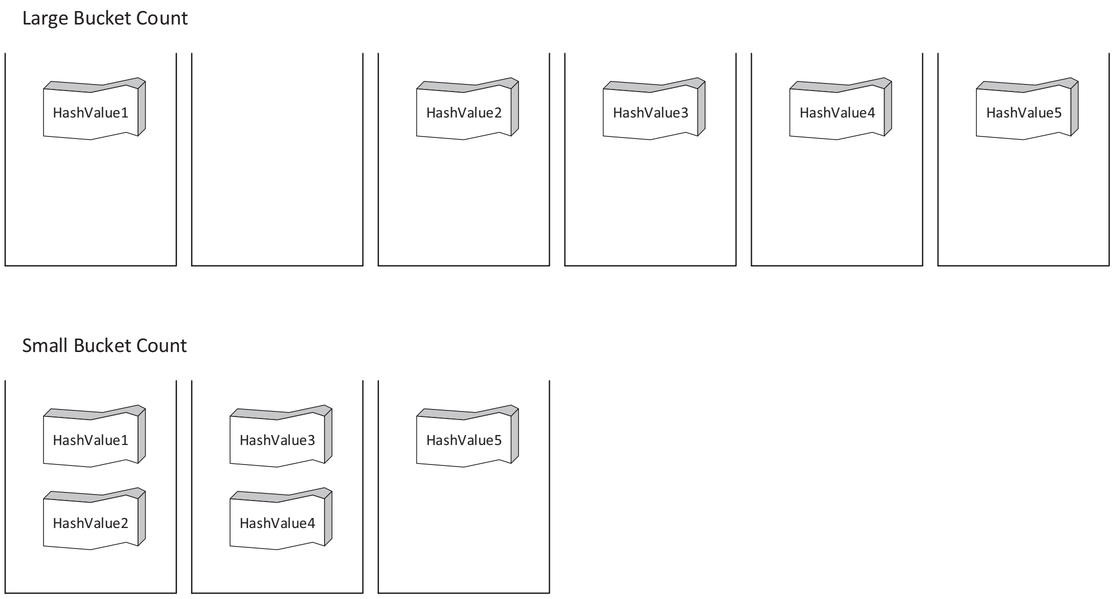


***Figure 24-5.*** *Hash values in lots of buckets and few buckets*

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.

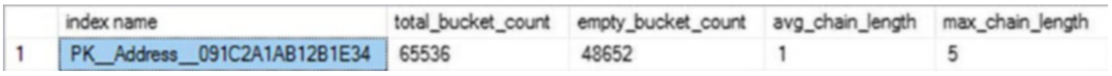
Microsoft's recommendation on bucket count is go between one to two times the quantity of the number of rows in the table. But, since you can't alter in-memory tables, you also need to consider projected growth. If you think your in-memory table is likely to grow three times as large over the next three 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.

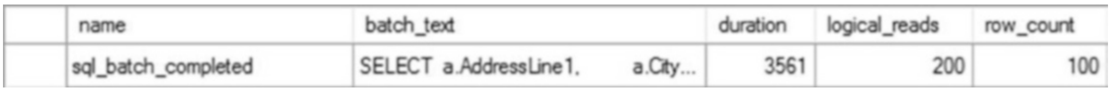
## Nonclustered Indexes

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.

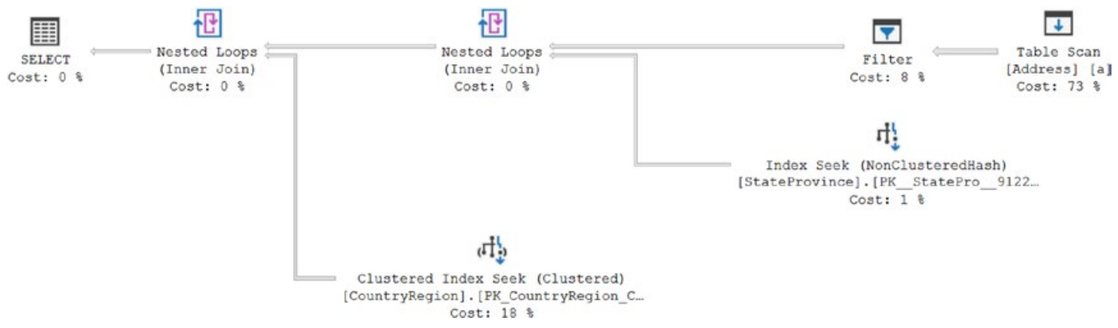


***Figure 24-8.*** *Query results in an execution plan that has table scans*

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.

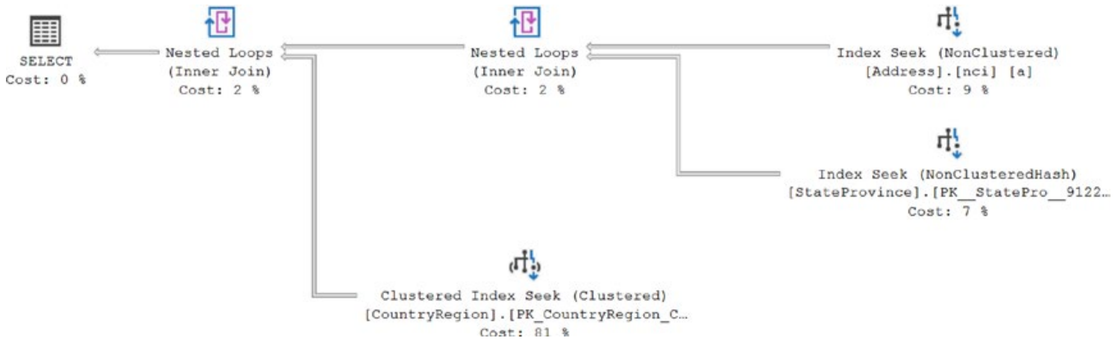

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 2 %
Nested Loops
(Inner Join)
Cost: 2 %
Index Seek (NonClustered)
[Address].[nci] [a]
Cost: 9 %
Index Seek (NonClusteredHash)
[StateProvince].[PK__StatePro__9122…
Cost: 7 %
Clustered Index Seek (Clustered)
[CountryRegion].[PK_CountryRegion_C…
Cost: 81 %

***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

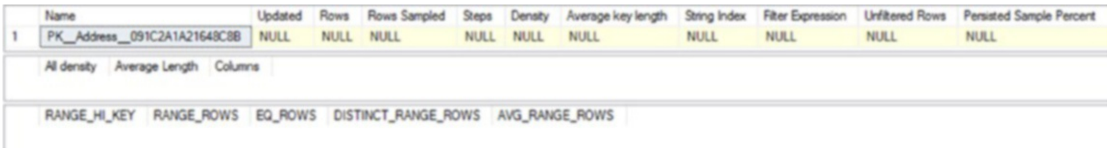
There actually isn't much to say about adding a columnstore index to an in-memory table. Since columnstore indexes work best on tables with 100,000 rows or more, you will need quite a lot of memory to support their implementation on your in-memory tables. You are limited to clustered columnstore indexes. 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

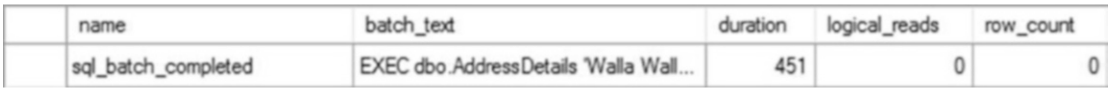
Just getting the table into memory and radically reducing the locking contention with the optimistic approaches results in impressive performance improvements. To really make things move quickly, you can also implement the new feature of compiling stored procedures into a DLL that runs within the SQL 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.

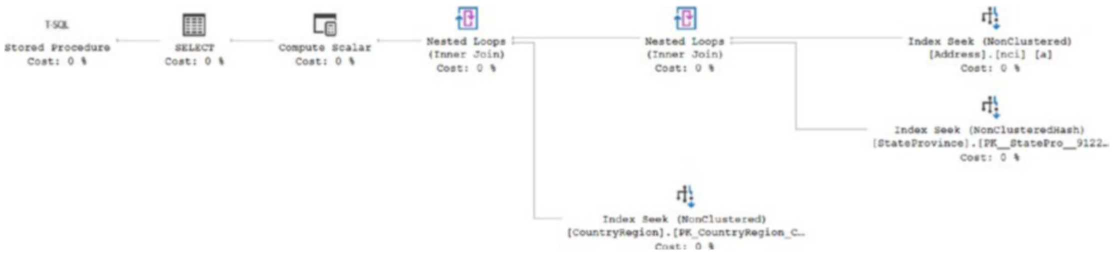
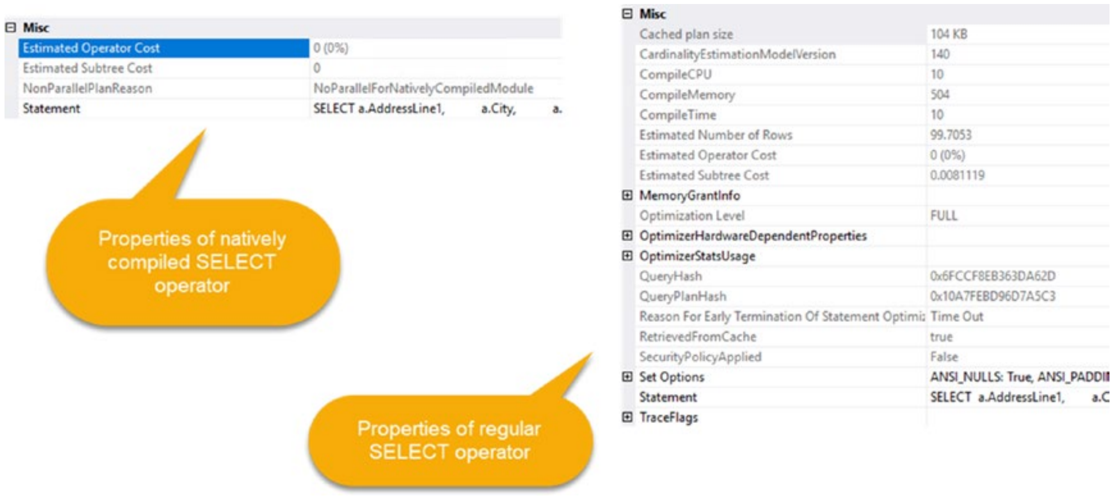

T-SQL
Stored Procedure Cost: 0 %
SELECT Cost: 0 %
Compute Scalar Cost: 0 %
Nested Loops (Inner Join) Cost: 0 %
Nested Loops (Inner Join) Cost: 0 %
Index Seek (NonClustered) [Address].[nci] [a] Cost: 0 %
Index Seek (NonClusteredHash) [StateProvince].[PK__StatePro__9122… Cost: 0 %
Index Seek (NonClustered) [CountryRegion].[PK_CountryRegion_C… Cost: 0 %

*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 | |
|---|---|
| Estimated Operator Cost | 0 (0%) |
| Estimated Subtree Cost | 0 |
| NonParallelPlanReason | NoParallelForNativelyCompiledModule |
| Statement | SELECT a.AddressLine1, a.City, a. |

| Misc | |
|---|---|
| Cached plan size | 104 KB |
| CardinalityEstimationModelVersion | 140 |
| CompileCPU | 10 |
| 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 | 0x6FCCF8EB363DA62D |
| QueryPlanHash | 0x10A7FEBD96D7A5C3 |
| Reason For Early Termination Of Statement Optimi: | Time Out |
| RetrievedFromCache | true |
| SecurityPolicyApplied | False |
| ⊞ Set Options | ANSI_NULLS: True, ANSI_PADDI |
| Statement | SELECT a.AddressLine1, a.C |
| ⊞ TraceFlags | |

Properties of natively compiled SELECT operator
Properties of regular SELECT operator

*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.

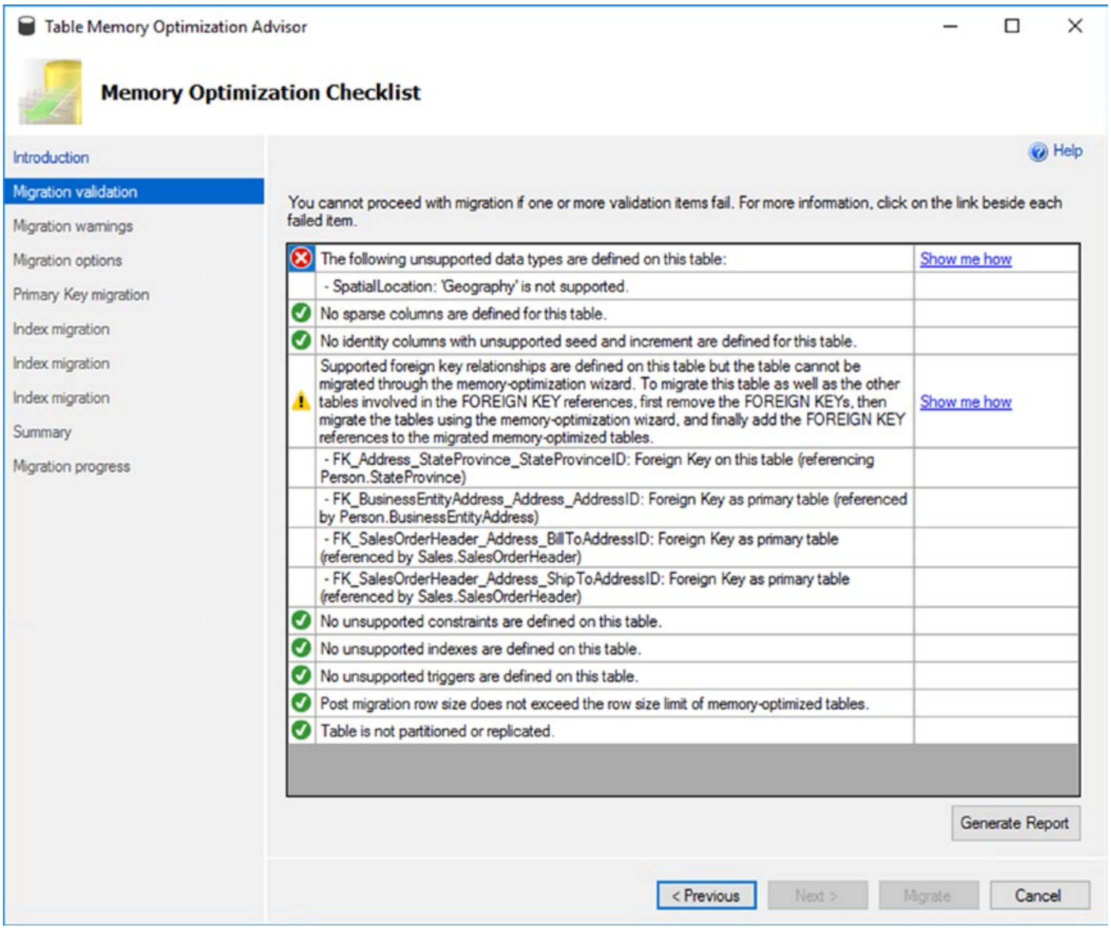


Table Memory Optimization Advisor

Memory Optimization Checklist

You cannot proceed with migration if one or more validation items fail. For more information, click on the link beside each failed item.

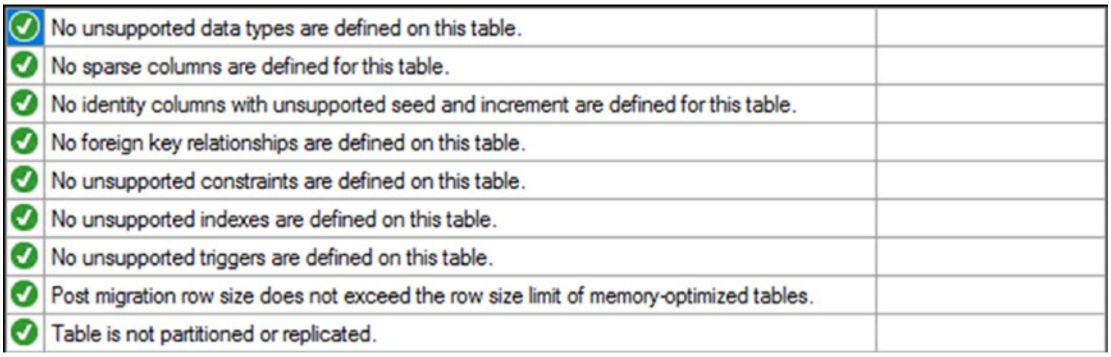
| | | |
|---|---|---|
| ✗ | The following unsupported data types are defined on this table: | Show me how |
| | - SpatialLocation: 'Geography' is not supported. | |
| ✓ | No sparse columns are defined for this table. | |
| ✓ | No identity columns with unsupported seed and increment are defined for this table. | |
| ⚠ | Supported foreign key relationships are defined on this table but the table cannot be migrated through the memory-optimization wizard. To migrate this table as well as the other tables involved in the FOREIGN KEY references, first remove the FOREIGN KEYs, then migrate the tables using the memory-optimization wizard, and finally add the FOREIGN KEY references to the migrated memory-optimized tables. | Show me how |
| | - FK_Address_StateProvince_StateProvinceID: Foreign Key on this table (referencing Person.StateProvince) | |
| | - FK_BusinessEntityAddress_Address_AddressID: Foreign Key as primary table (referenced by Person.BusinessEntityAddress) | |
| | - FK_SalesOrderHeader_Address_BillToAddressID: Foreign Key as primary table (referenced by Sales.SalesOrderHeader) | |
| | - FK_SalesOrderHeader_Address_ShipToAddressID: Foreign Key as primary table (referenced by Sales.SalesOrderHeader) | |
| ✓ | 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 24-14.*** *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
    );
```

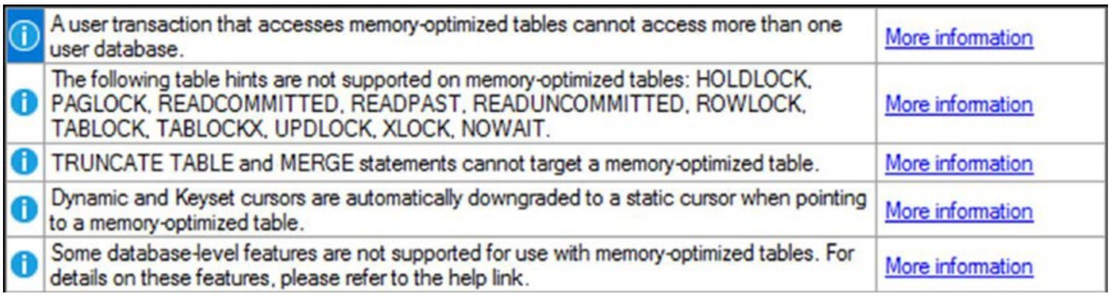
Now, running the Memory Optimization Advisor has completely different results in the first step, as shown in Figure 24-15.



| | | |
|---|---|---|
| ✓ | 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*

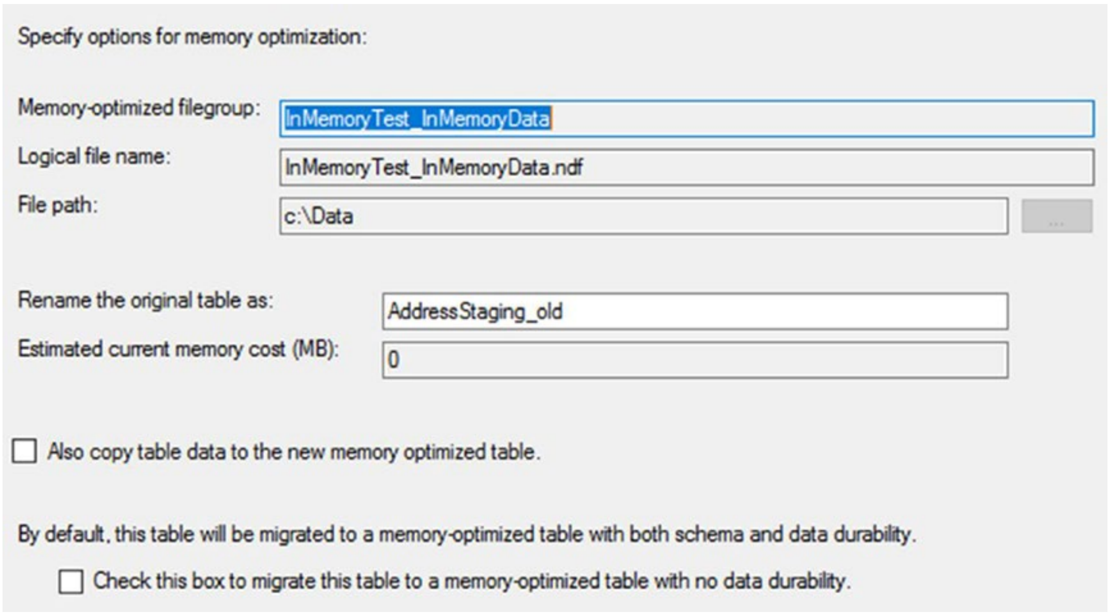
The next step in the wizard shows a fairly standard set of warnings about the differences that using the in-memory tables will cause in your T-SQL as well as links to further reading about these limitations. It's a useful reminder that you may have to address your code should you choose to migrate this 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. | More information |
| ⓘ | The following table hints are not supported on memory-optimized tables: HOLDLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, ROWLOCK, TABLOCK, TABLOCKX, UPDLOCK, XLOCK, NOWAIT. | More information |
| ⓘ | TRUNCATE TABLE and MERGE statements cannot target a memory-optimized table. | More information |
| ⓘ | Dynamic and Keyset cursors are automatically downgraded to a static cursor when pointing to a memory-optimized table. | More information |
| ⓘ | Some database-level features are not supported for use with memory-optimized tables. For details on these features, please refer to the help link. | More information |

***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

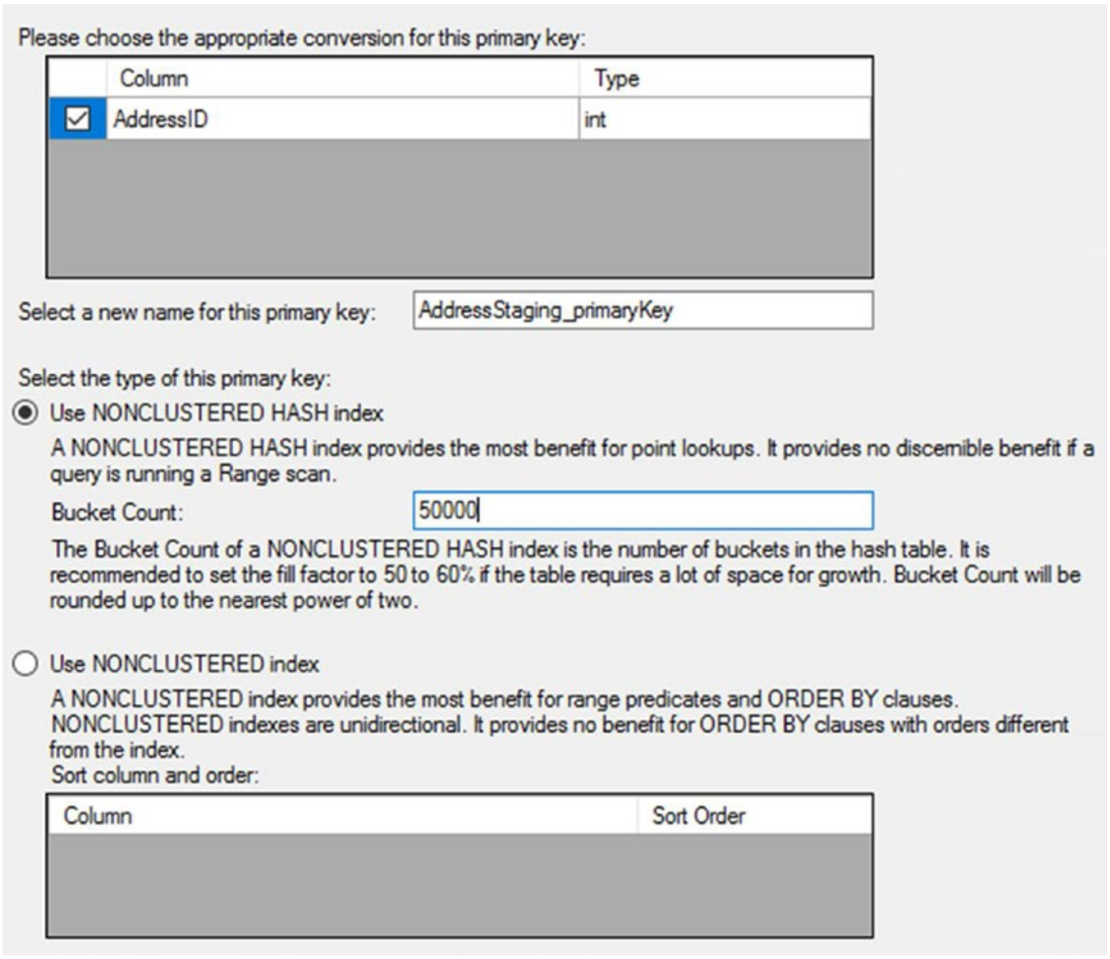
☐ Also copy table data to the new memory optimized table.

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 |
|---|---|---|
| ✓ | AddressID | int |

Select a new name for this primary key: AddressStaging_primaryKey

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: 50000

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.

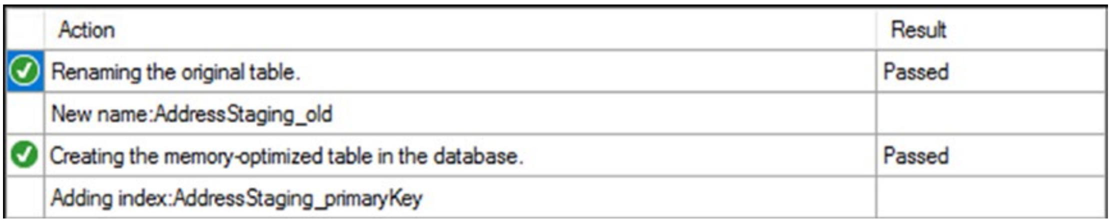
○ Use NONCLUSTERED index

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.

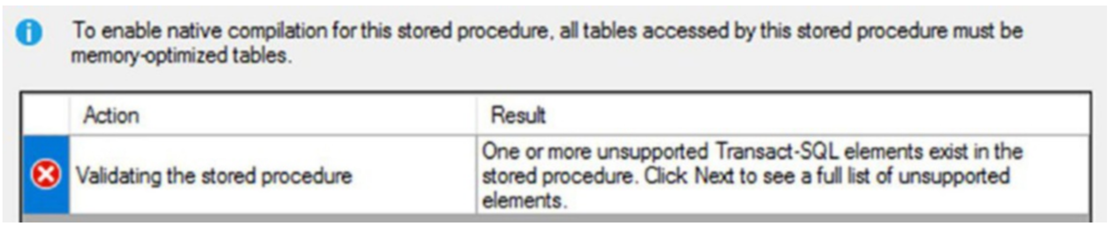
## Native Compilation Advisor

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 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.



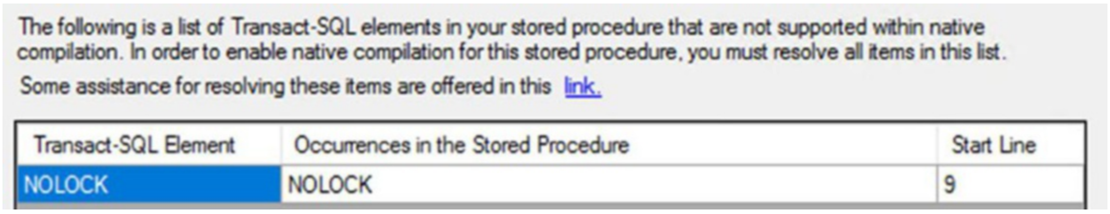
To enable native compilation for this stored procedure, all tables accessed by this stored procedure must be memory-optimized tables.

| | Action | Result |
|---|---|---|
| ⊗ | Validating the stored procedure | One or more unsupported Transact-SQL elements exist in the stored procedure. Click Next to see a full list of unsupported elements. |

***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

© 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.

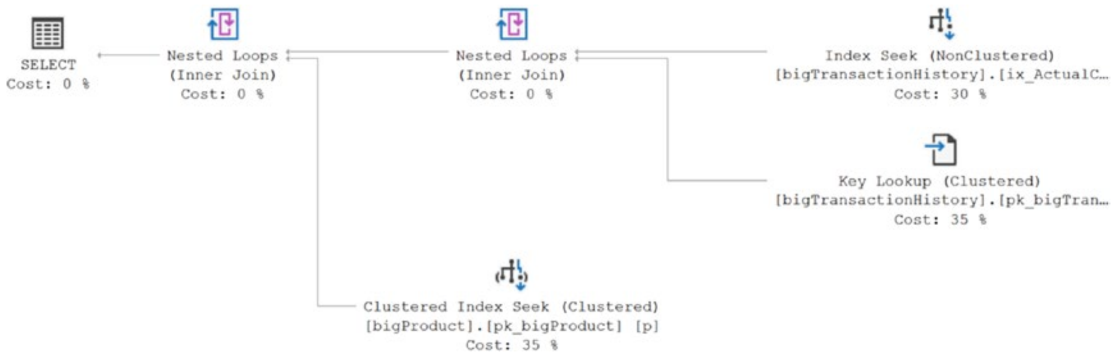

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[bigTransactionHistory].[ix_ActualC…
Cost: 30 %
Key Lookup (Clustered)
[bigTransactionHistory].[pk_bigTran…
Cost: 35 %
Clustered Index Seek (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: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

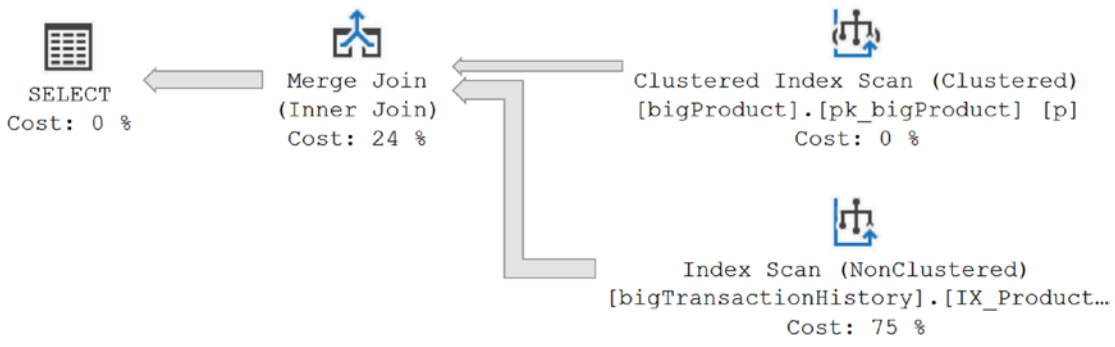

SELECT
Cost: 0 %
Merge Join
(Inner Join)
Cost: 24 %
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct] [p]
Cost: 0 %
Index Scan (NonClustered)
[bigTransactionHistory].[IX_Product…
Cost: 75 %

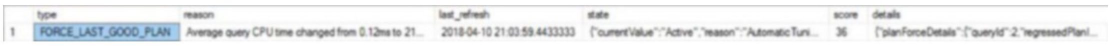
***Figure 25-2.*** *Execution plan for a much larger data set*

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.

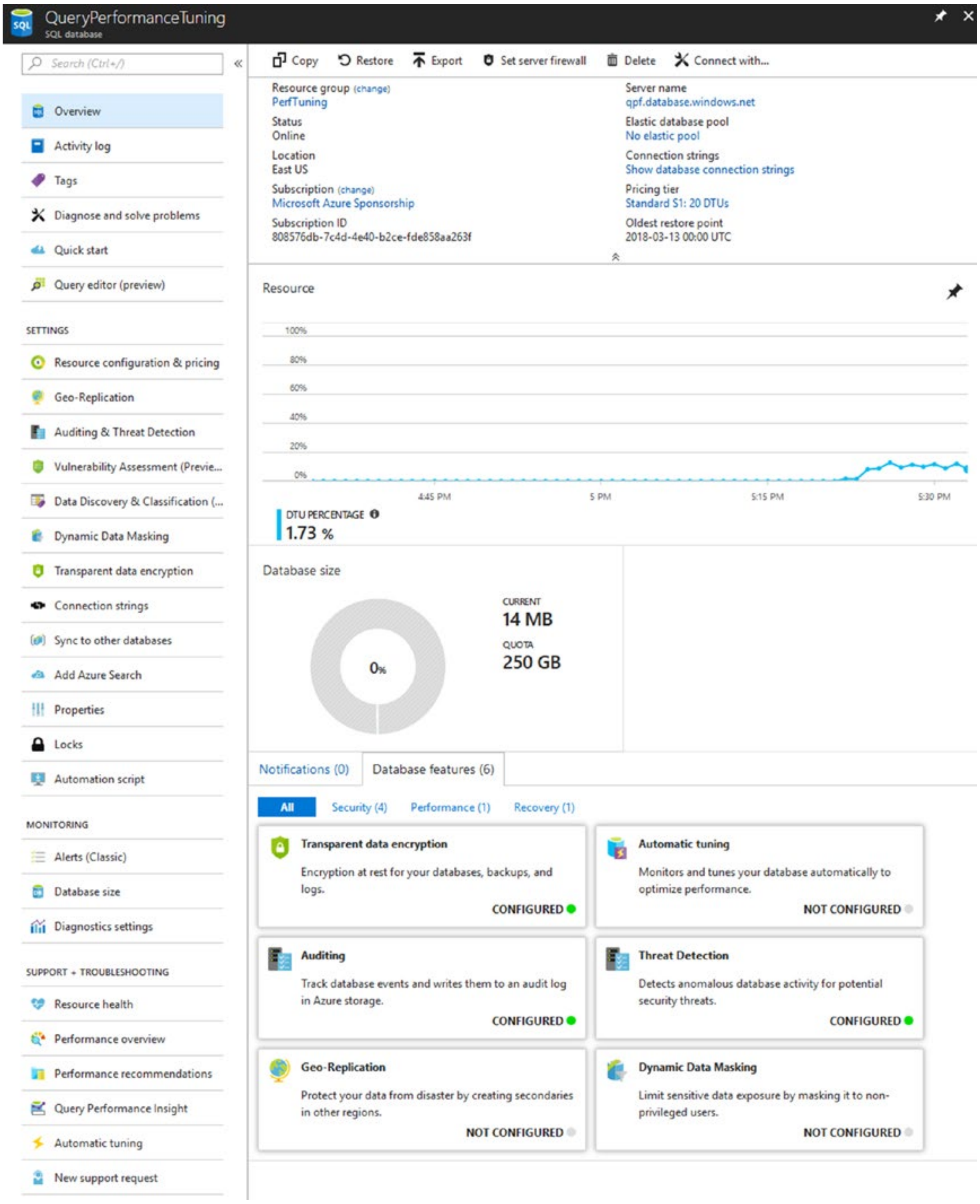
---

**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)<br>PerfTuning | Server name<br>qpf.database.windows.net |
| Status<br>Online | Elastic database pool<br>No elastic pool |
| Location<br>East US | Connection strings<br>Show database connection strings |
| Subscription (change)<br>Microsoft Azure Sponsorship | Pricing tier<br>Standard S1: 20 DTUs |
| Subscription ID<br>808576db-7c4d-4e40-b2ce-fde858aa263f | Oldest restore point<br>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 ●

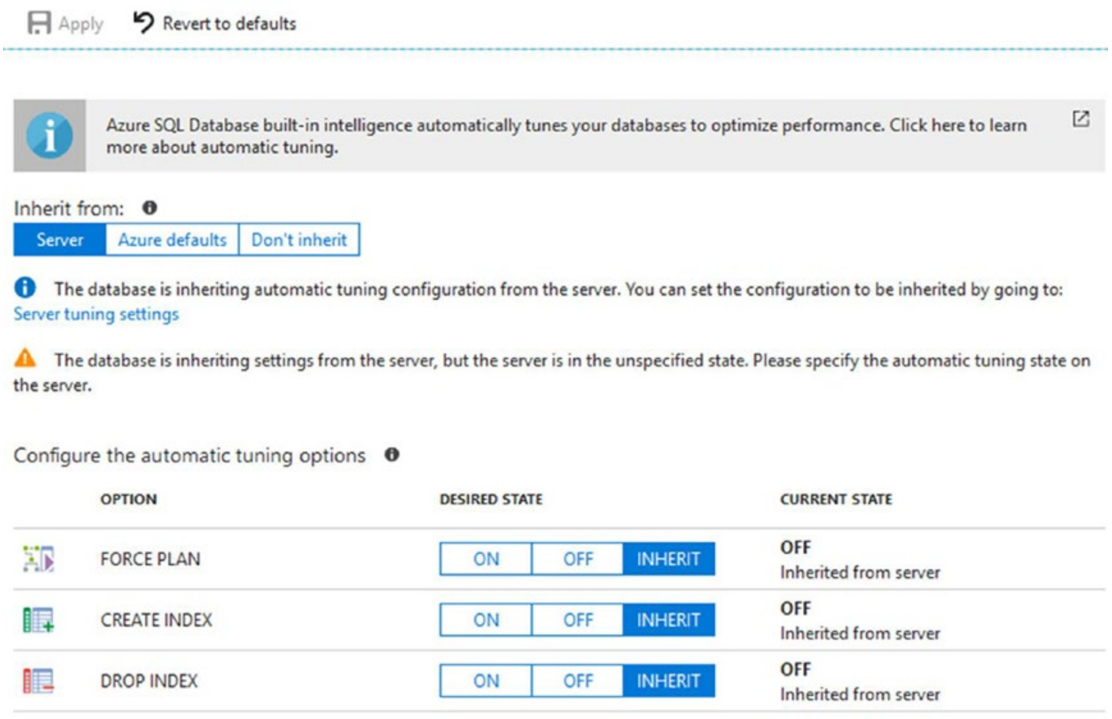
**Threat Detection**
Detects anomalous database activity for potential security threats.
CONFIGURED ●

**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

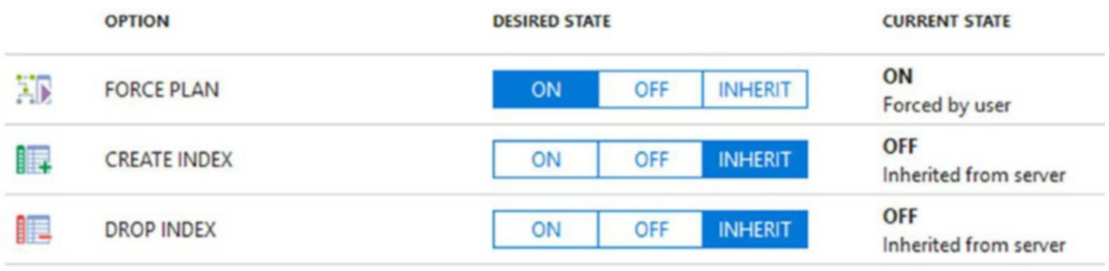
The database is inheriting settings from the server, but the server is in the unspecified state. Please specify the automatic tuning state on the server.

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.

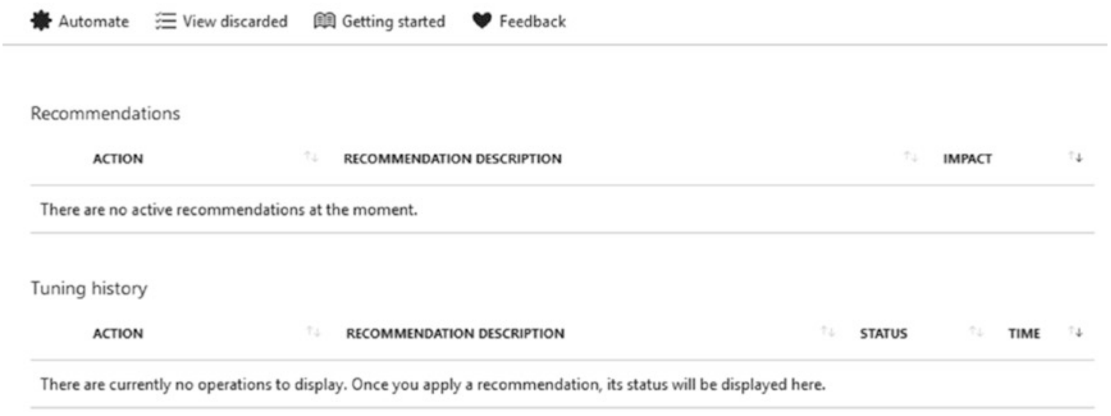


| OPTION | DESIRED STATE | CURRENT STATE |
|---|---|---|
| FORCE PLAN | [✓ ON] OFF INHERIT | ON Forced by user |
| CREATE INDEX | ON OFF [✓ INHERIT] | OFF Inherited from server |
| DROP INDEX | ON OFF [✓ INHERIT] | OFF Inherited from server |

***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.

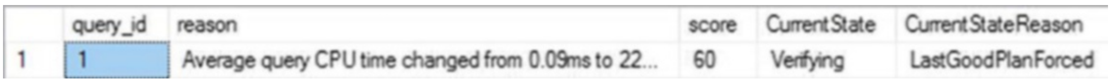
```
SELECT name,
       desired_state,
       desired_state_desc,
       actual_state,
       actual_state_desc,
       reason,
       reason_desc
FROM sys.database_automatic_tuning_options;
```

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 |

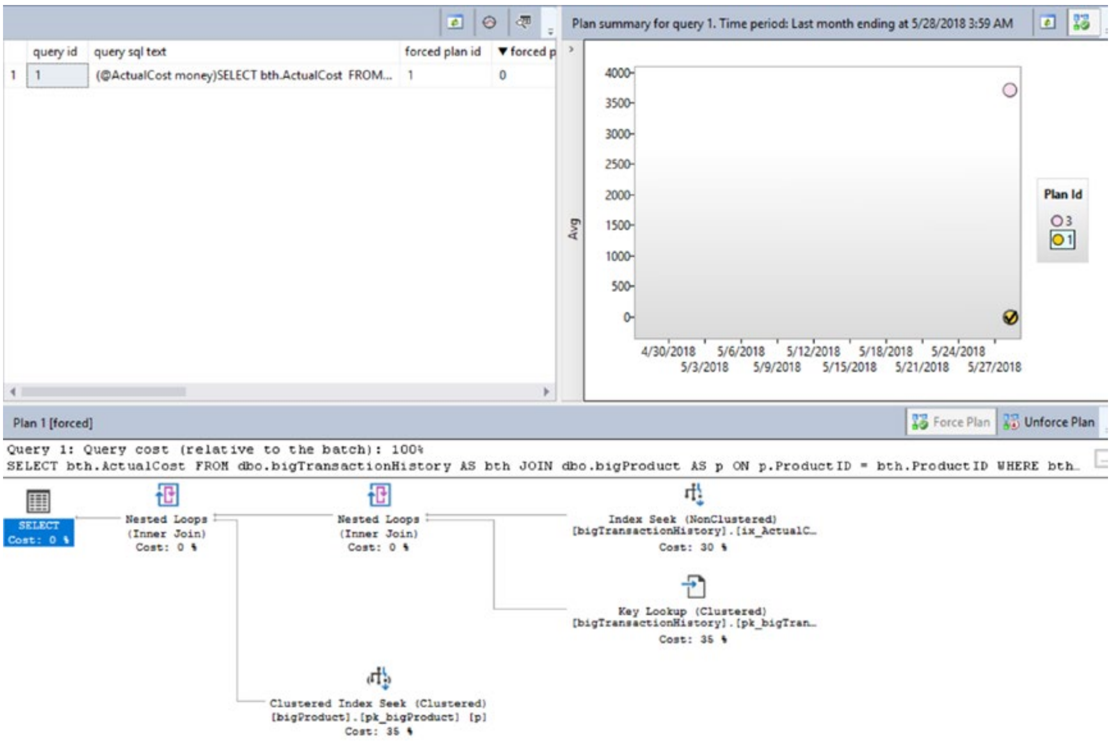
***Figure 25-8.** The regressed query has been forced*

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…
```

Avg
4000
3500
3000
2500
2000
1500
1000
500
0
4/30/2018
5/3/2018
5/6/2018
5/9/2018
5/12/2018
5/15/2018
5/18/2018
5/21/2018
5/24/2018
5/27/2018
Plan Id
3
1
Force Plan
Unforce Plan
SELECT Cost: 0 %
Nested Loops (Inner Join) Cost: 0 %
Nested Loops (Inner Join) Cost: 0 %
Index Seek (NonClustered) [bigTransactionHistory].[ix_ActualC… Cost: 30 %
Key Lookup (Clustered) [bigTransactionHistory].[pk_bigTran… Cost: 35 %
Clustered Index Seek (Clustered) [bigProduct].[pk_bigProduct] [p] Cost: 35 %

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

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

## Azure SQL Database Automatic Index Management

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

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

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

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

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

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

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

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

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

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

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

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

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

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

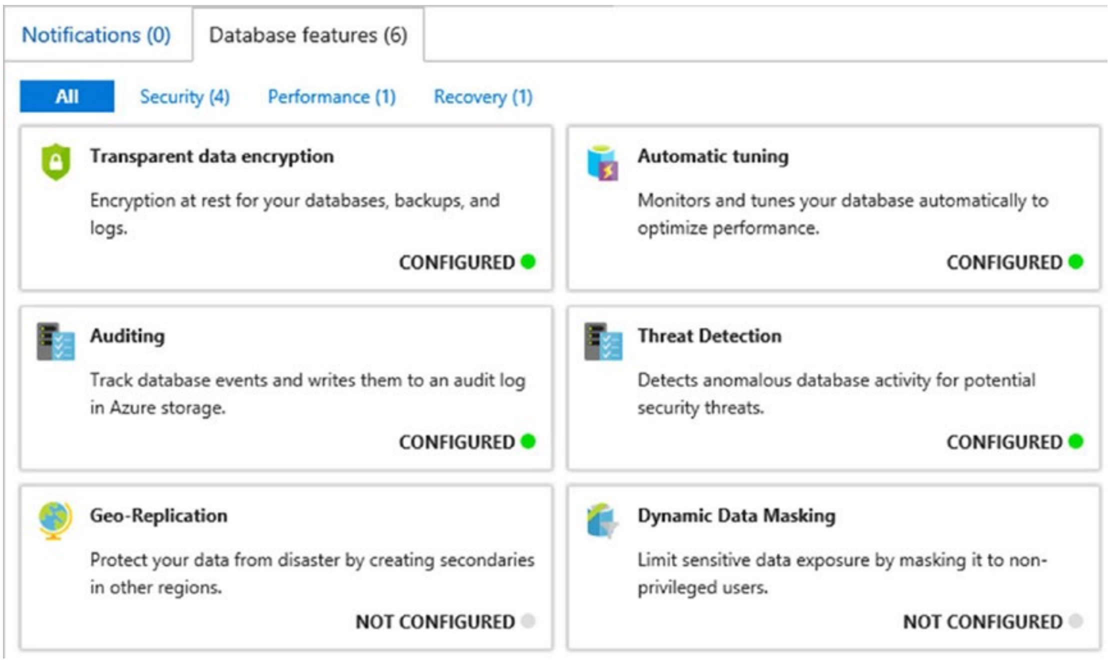
```
while(1 -ne 0)
{
    foreach($row in $DatDataSet.Tables[0])
        {

        $name = $row[0]
        $email = $row[1]
        $SqlConnection.Open()
        $Proccmd.Parameters["@FirstName"].Value = $name
        $Proccmd.ExecuteNonQuery() | Out-Null
        $EmailCmd.Parameters["@EmailAddress"].Value = $email
        $EmailCmd.ExecuteNonQuery() | Out-Null
        $SalesCmd.Parameters["@FirstName"].Value = $name
        $SalesCmd.ExecuteNonQuery() | Out-Null
        $OddCmd.Parameters["@FirstName"].Value = $name
        $OddCmd.ExecuteNonQuery() | Out-Null
        $SqlConnection.Close()

 }
 }
```

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

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

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

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


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.
CONFIGURED ●

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

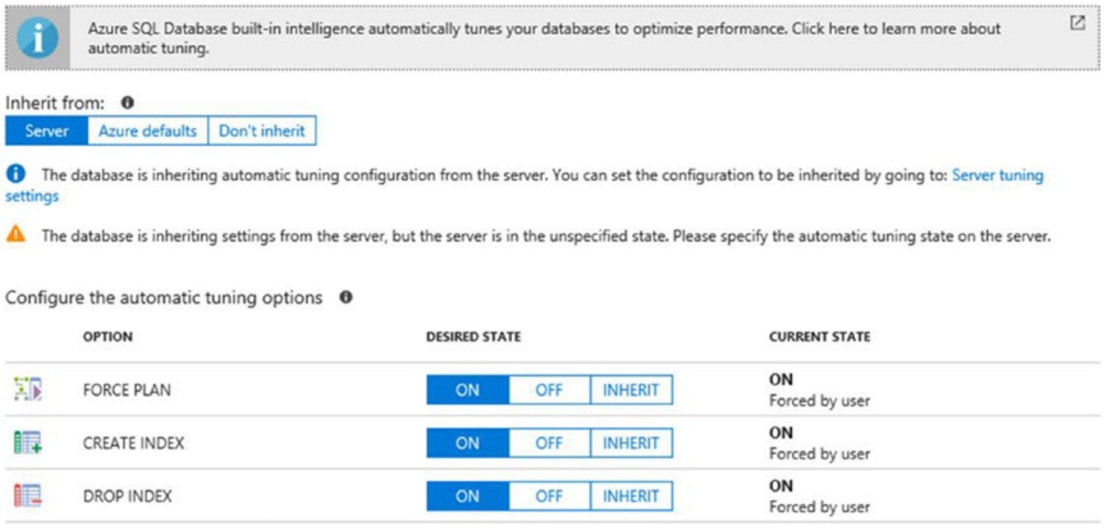
**Threat Detection**
Detects anomalous database activity for potential security threats.
CONFIGURED ●

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

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

***Figure 25-10.*** *Database features in Azure SQL Database including "Automatic tuning"*

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


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

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

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

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

Configure the automatic tuning options

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

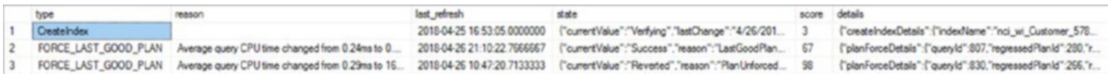
***Figure 25-11.*** *Enabling "Automatic tuning" within Azure SQL Database*

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

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

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

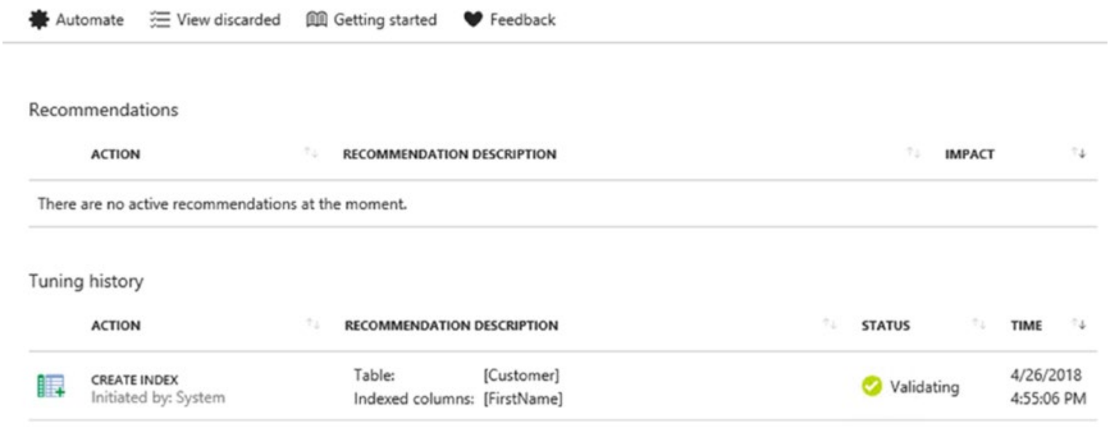
The results on my Azure SQL Database look something like Figure 25-12.



|  | type | reason | last_refresh | state | score | details |
|---|---|---|---|---|---|---|
| 1 | CreateIndex |  | 2018-04-25 16:53:05.0000000 | {"currentValue":"Verifying","lastChange":"4/26/201... | 3 | {"createIndexDetails":{"indexName":"nci_wi_Customer_578... |
| 2 | FORCE_LAST_GOOD_PLAN | Average query CPU time changed from 0.24ms to 0... | 2018-04-26 21:10:22.7666667 | {"currentValue":"Success","reason":"LastGoodPlan... | 67 | {"planForceDetails":{"queryId":807,"regressedPlanId":280,"r... |
| 3 | FORCE_LAST_GOOD_PLAN | Average query CPU time changed from 0.29ms to 16... | 2018-04-26 10:47:20.7133333 | {"currentValue":"Reverted","reason":"PlanUnforced... | 98 | {"planForceDetails":{"queryId":830,"regressedPlanId":296,"r... |

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

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



Recommendations

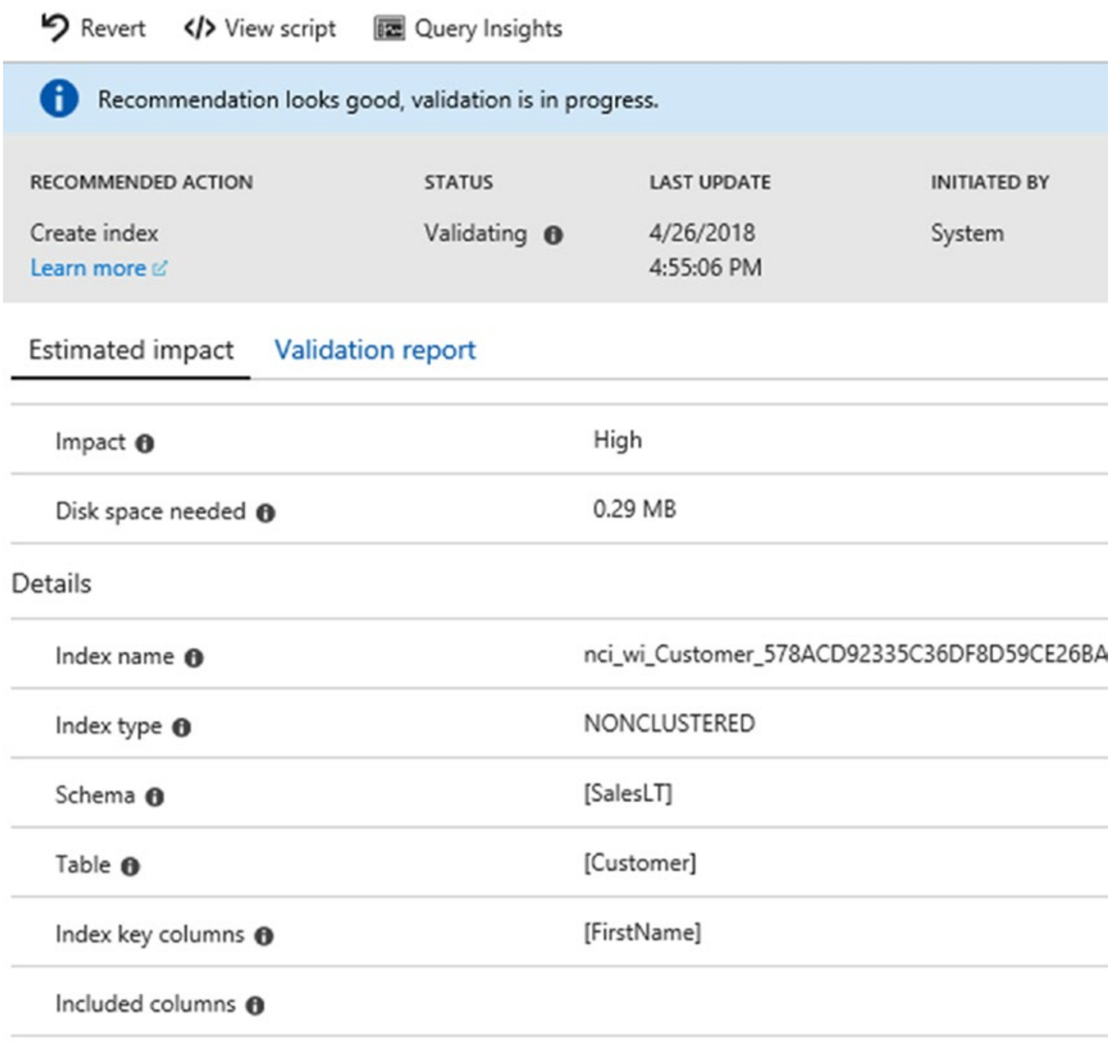
| ACTION | RECOMMENDATION DESCRIPTION | IMPACT |
|---|---|---|
| There are no active recommendations at the moment. | | |

Tuning history

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

***Figure 25-13.*** *Performance recommendations and tuning history*

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



Recommendation looks good, validation is in progress.

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

| | |
|---|---|
| Impact | High |
| Disk space needed | 0.29 MB |

Details

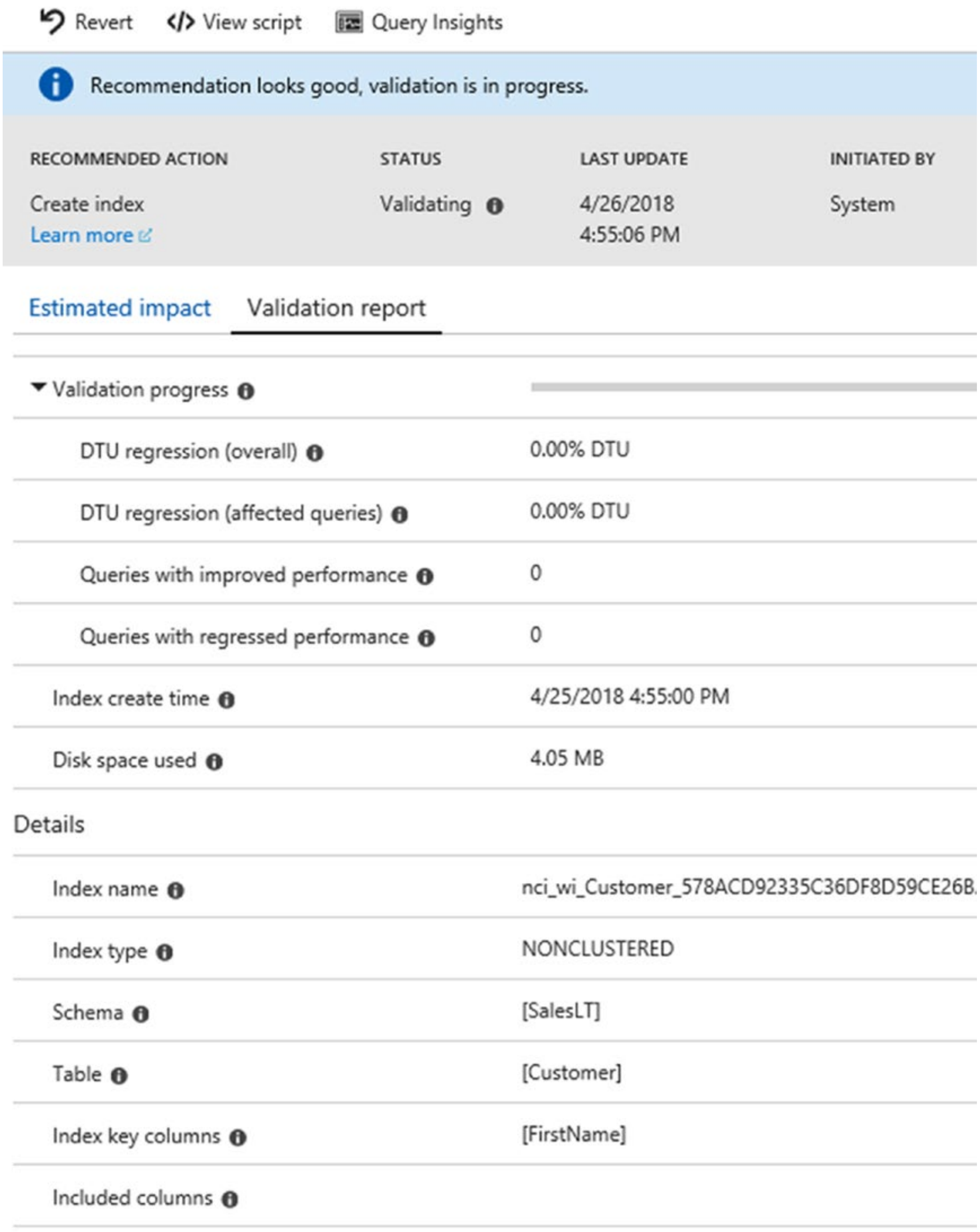
| | |
|---|---|
| Index name | nci_wi_Customer_578ACD92335C36DF8D59CE26BA |
| Index type | NONCLUSTERED |
| Schema | [SalesLT] |
| Table | [Customer] |
| Index key columns | [FirstName] |
| Included columns | |

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

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

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

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



Recommendation looks good, validation is in progress.

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

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

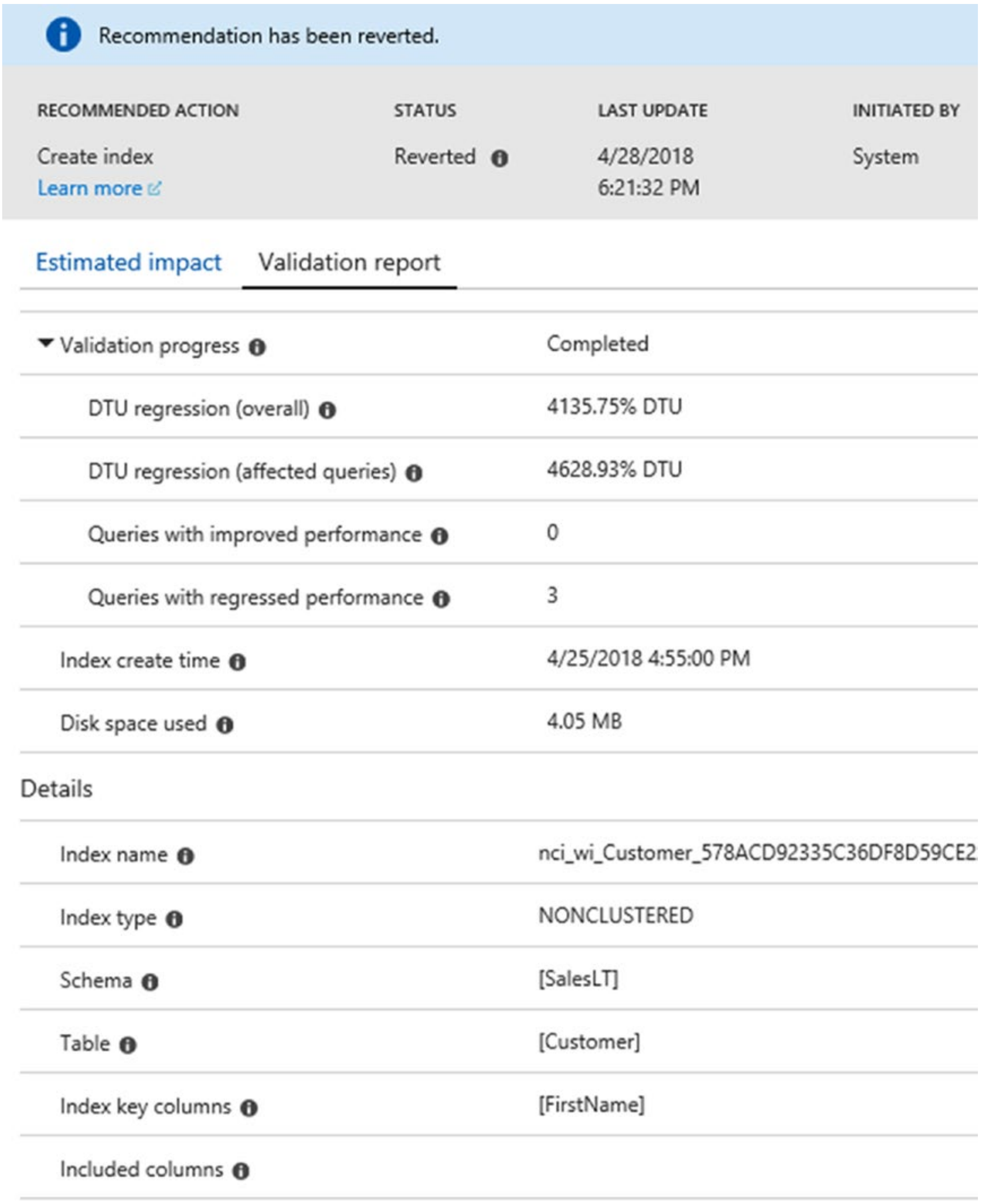
Details

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

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

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

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



Recommendation has been reverted.

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

Estimated impact | Validation report

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

Details

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

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

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

## Adaptive Query Processing

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

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

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

## Batch Mode Memory Grant Feedback

Batch mode, as of this writing, is supported only by queries that involve a columnstore index, clustered or nonclustered. Batch mode itself is worth a short explanation. During row mode execution within an execution plan, each pair of operators has to negotiate each row being transferred between them. If there are ten rows, there are ten negotiations. If there are ten million rows, there are ten million negotiations. As you can imagine, this gets quite costly. So, in a batch mode operation, instead of processing each row one at a time, the processing occurs in batches, generally distributed up to 900 rows per batch, but there is quite a bit of variation there. This means, instead of ten million negotiations to move ten million rows, there are only this many:

10000000 rows / ~ 900 rows per batch = 11,111 batches

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

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

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

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

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

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

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

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

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

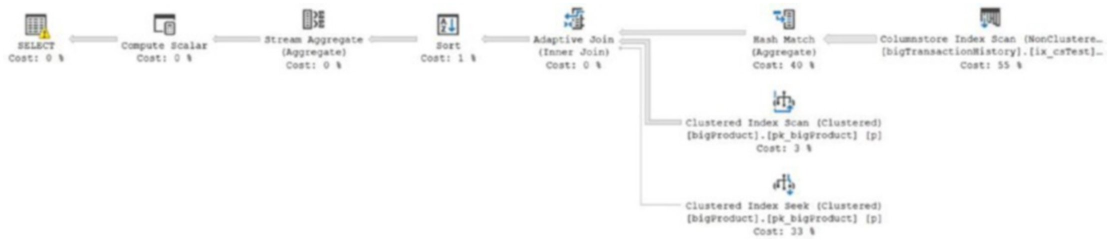
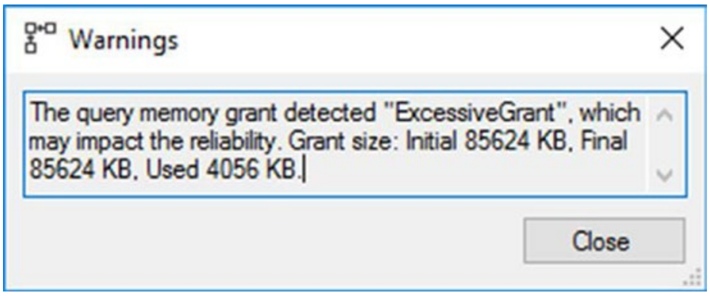

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Stream Aggregate
(Aggregate)
Cost: 0 %
Sort
Cost: 1 %
Adaptive Join
(Inner Join)
Cost: 0 %
Hash Match
(Aggregate)
Cost: 40 %
Columnstore Index Scan (NonClustere…
[bigTransactionHistory].[ix_csTest]…
Cost: 55 %
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct]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

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

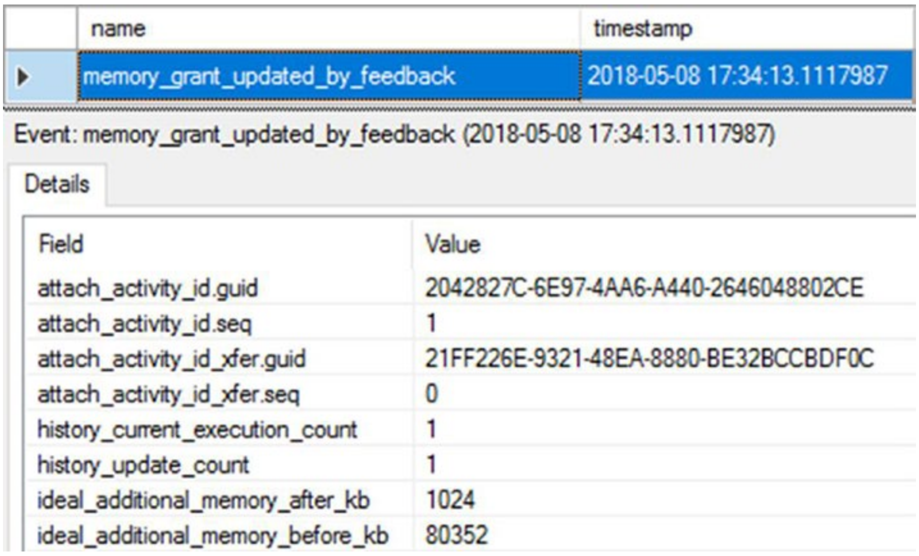
What should immediately draw your eye with this query is the warning on the `SELECT` operator. We can open the Properties window to see all the warnings for a plan. Note that the tooltip only ever shows the first warning, as shown in Figure 25-18.



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

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

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



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

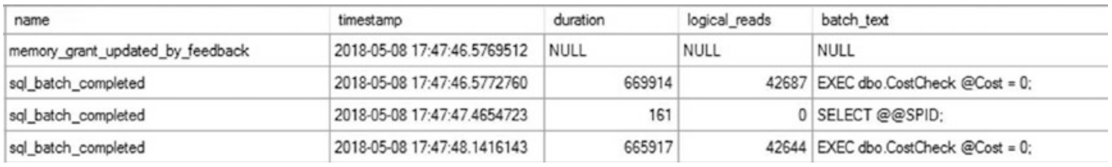
Event: memory_grant_updated_by_feedback (2018-05-08 17:34:13.1117987)

Details

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

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

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



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

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

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

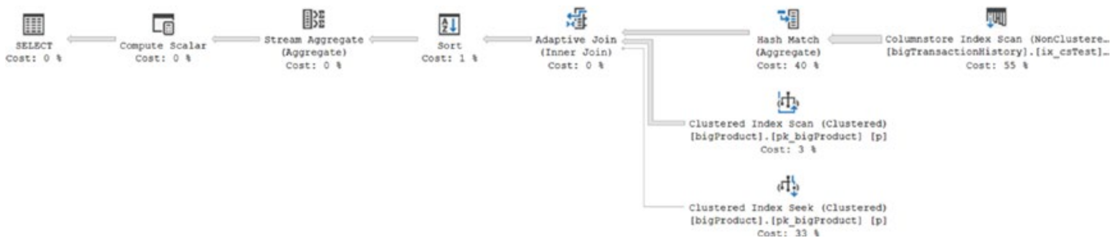

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Stream Aggregate
(Aggregate)
Cost: 0 %
Sort
Cost: 1 %
Adaptive Join
(Inner Join)
Cost: 0 %
Hash Match
(Aggregate)
Cost: 40 %
Columnstore Index Scan (NonClustere…
[bigTransactionHistory].[ix_csTest]…
Cost: 55 %
Clustered Index Scan (Clustered)
[bigProduct].[pk_bigProduct]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

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

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

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

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

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

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

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

## Interleaved Execution

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

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

```
CREATE OR ALTER FUNCTION dbo.SalesInfo ()
RETURNS @return_variable TABLE (SalesOrderID INT,
                                OrderDate DATETIME,
                                SalesPersonID INT,
                                PurchaseOrderNumber dbo.OrderNumber,
                                AccountNumber dbo.AccountNumber,
                                ShippingCity NVARCHAR(30))
AS
BEGIN;
    INSERT INTO @return_variable (SalesOrderID,
                                  OrderDate,
                                  SalesPersonID,
                                  PurchaseOrderNumber,
                                  AccountNumber,
                                  ShippingCity)
    SELECT soh.SalesOrderID,
           soh.OrderDate,
           soh.SalesPersonID,
           soh.PurchaseOrderNumber,
           soh.AccountNumber,
           a.City
    FROM Sales.SalesOrderHeader AS soh
        JOIN Person.Address AS a
            ON soh.ShipToAddressID = a.AddressID;
    RETURN;
END;
GO

CREATE OR ALTER FUNCTION dbo.SalesDetails ()
RETURNS @return_variable TABLE (SalesOrderID INT,
                                SalesOrderDetailID INT,
                                OrderQty SMALLINT,
                                UnitPrice MONEY)
AS
BEGIN;
    INSERT INTO @return_variable (SalesOrderID,
                                  SalesOrderDetailID,
                                  OrderQty,
                                  UnitPrice)
    SELECT sod.SalesOrderID,
           sod.SalesOrderDetailID,
           sod.OrderQty,
           sod.UnitPrice
    FROM Sales.SalesOrderDetail AS sod;
    RETURN;
END;
GO
```

```
CREATE OR ALTER FUNCTION dbo.CombinedSalesInfo ()
RETURNS @return_variable TABLE (SalesPersonID INT,
                                ShippingCity NVARCHAR(30),
                                OrderDate DATETIME,
                                PurchaseOrderNumber dbo.OrderNumber,
                                AccountNumber dbo.AccountNumber,
                                OrderQty SMALLINT,
                                UnitPrice MONEY)
AS
BEGIN;
    INSERT INTO @return_variable (SalesPersonID,
                                  ShippingCity,
                                  OrderDate,
                                  PurchaseOrderNumber,
                                  AccountNumber,
                                  OrderQty,
                                  UnitPrice)
    SELECT si.SalesPersonID,
           si.ShippingCity,
           si.OrderDate,
           si.PurchaseOrderNumber,
           si.AccountNumber,
           sd.OrderQty,
           sd.UnitPrice
    FROM dbo.SalesInfo() AS si
        JOIN dbo.SalesDetails() AS sd
            ON si.SalesOrderID = sd.SalesOrderID;
    RETURN;
END;
GO
```

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

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

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

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

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

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

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

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

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

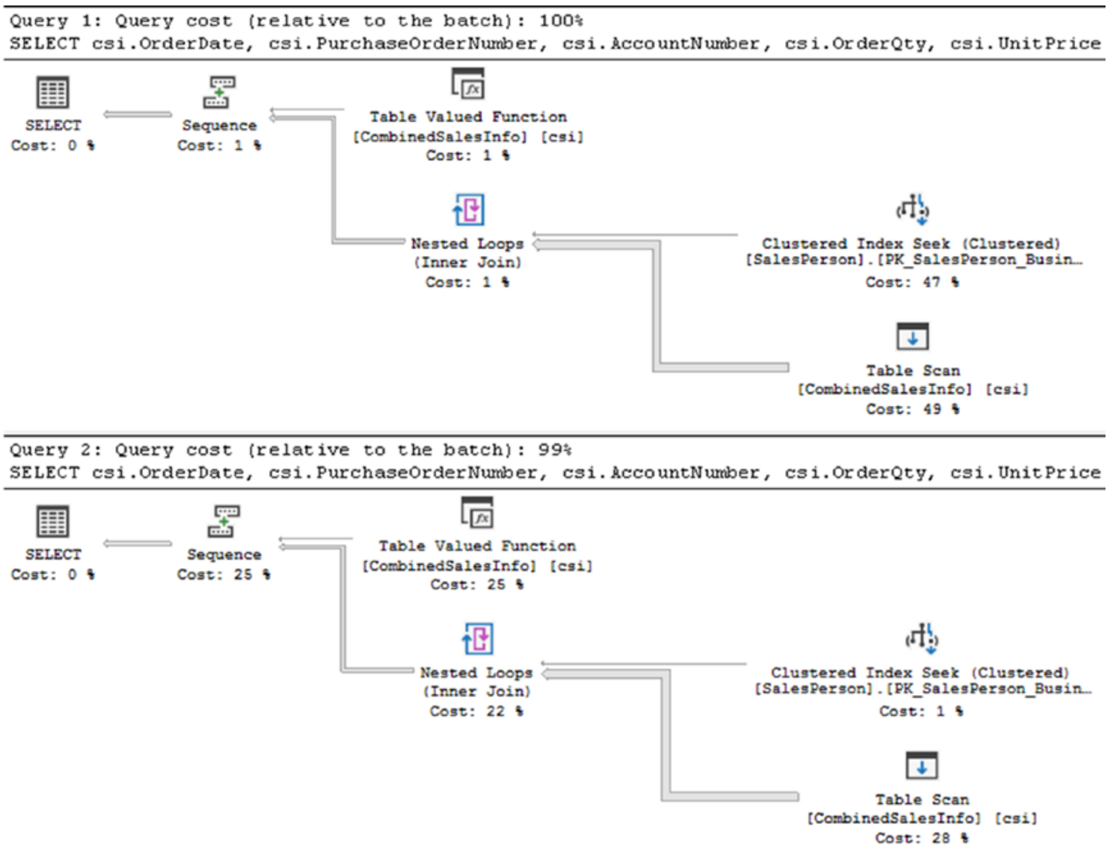
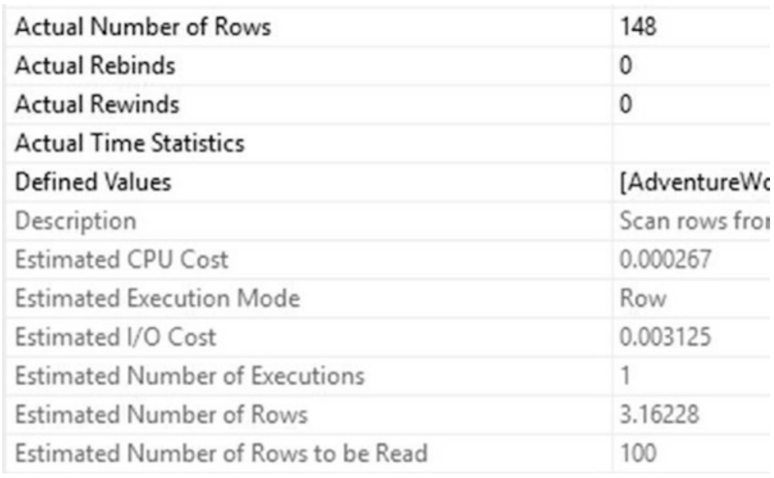

Query 1: Query cost (relative to the batch): 100%
SELECT csi.OrderDate, csi.PurchaseOrderNumber, csi.AccountNumber, csi.OrderQty, csi.UnitPrice
SELECT
Cost: 0 %
Sequence
Cost: 1 %
Table Valued Function
[CombinedSalesInfo] [csi]
Cost: 1 %
Nested Loops
(Inner Join)
Cost: 1 %
Clustered Index Seek (Clustered)
[SalesPerson].[PK_SalesPerson_Busin…
Cost: 47 %
Table Scan
[CombinedSalesInfo] [csi]
Cost: 49 %
Query 2: Query cost (relative to the batch): 99%
SELECT csi.OrderDate, csi.PurchaseOrderNumber, csi.AccountNumber, csi.OrderQty, csi.UnitPrice
SELECT
Cost: 0 %
Sequence
Cost: 25 %
Table Valued Function
[CombinedSalesInfo] [csi]
Cost: 25 %
Nested Loops
(Inner Join)
Cost: 22 %
Clustered Index Seek (Clustered)
[SalesPerson].[PK_SalesPerson_Busin…
Cost: 1 %
Table Scan
[CombinedSalesInfo] [csi]
Cost: 28 %

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

Looking at the plans, it's actually difficult to see the differences since all the operators are the same. However, the differences are in the estimated cost values. At the top, the table-valued function has an estimated cost of 1 percent, suggesting that it's almost free when compared to the `Clustered Index Seek` and `Table Scan` operations. In the second plan, though, the `Clustered Index Seek`, returning an estimated one row, suddenly costs only an estimated 1 percent of the total, and the rest of the cost is rightly redistributed to the other operations. It's these differences in row estimates that may lead, in some situations, to enhanced performance. However, let's look at the values to see this in action.

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

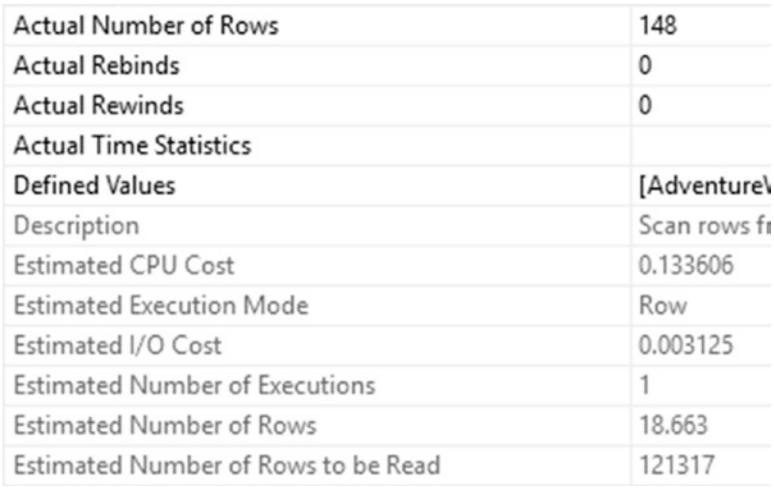


| | |
|---|---|
| Actual Number of Rows | 148 |
| Actual Rebinds | 0 |
| Actual Rewinds | 0 |
| Actual Time Statistics | |
| Defined Values | [AdventureWc |
| Description | Scan rows froı |
| Estimated CPU Cost | 0.000267 |
| Estimated Execution Mode | Row |
| Estimated I/O Cost | 0.003125 |
| Estimated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

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

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

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



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

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

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

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

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

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

```
AS
BEGIN;

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

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

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

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

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

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

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

## Summary

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

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

CHAPTER 26

# Database Performance Testing

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

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

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

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

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

This chapter covers the following topics:

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

## Database Performance Testing

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

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

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

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

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

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

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

## A Repeatable Process

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

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

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

## Distributed Replay

The Distributed Replay tool consists of three pieces of architecture.

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

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

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

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

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

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

## Capturing Data with the Server-Side Trace

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

***Table 26-1.*** *Events to Capture*

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