

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 sc1,
     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.ScrapQty * p.ListPrice) AS MoneyLostPerProduct
  FROM Production.WorkOrder AS wo
    JOIN Production.ScrapReason AS sr
      ON wo.ScrapReasonID = sr.ScrapReasonID
    JOIN Production.Product AS p
      ON wo.ProductID = p.ProductID
  GROUP BY p.ProductID) AS DiscardedProducts;
GO
```

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

```
EXEC dbo.TotalLoss;
```

Figure 23-4 shows the corresponding Extended Events output.

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

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

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

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

Cursor Recommendations

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

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

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

Summary

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

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

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

CHAPTER 24

Memory-Optimized OLTP Tables and Procedures

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

In this chapter, I cover the following topics:

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

In-Memory OLTP Fundamentals

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

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

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

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

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

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

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

System Requirements

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

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

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

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

Basic Setup

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

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

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

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


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

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

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

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

Create Tables

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

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

```

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

```

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

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

- XML
- ROWVERSION
- SQL_VARIANT
- HIERARCHYID
- DATETIMEOFFSET
- GEOGRAPHY/GEOMETRY
- User-defined data types

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

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

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

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

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

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

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

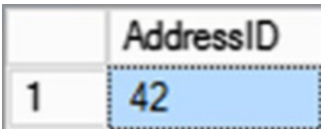
```
INSERT dbo.Address (AddressLine1,
                    AddressLine2,
                    City,
                    StateProvinceID,
                    PostalCode)
```

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

DROP TABLE dbo.AddressStaging;
```

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

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



	AddressID
1	42

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

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

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