

```

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

```

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

```

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

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

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

DROP TABLE dbo.StateProvinceStaging;

INSERT dbo.CountryRegion (CountryRegionCode,
                          Name)

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

```

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

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

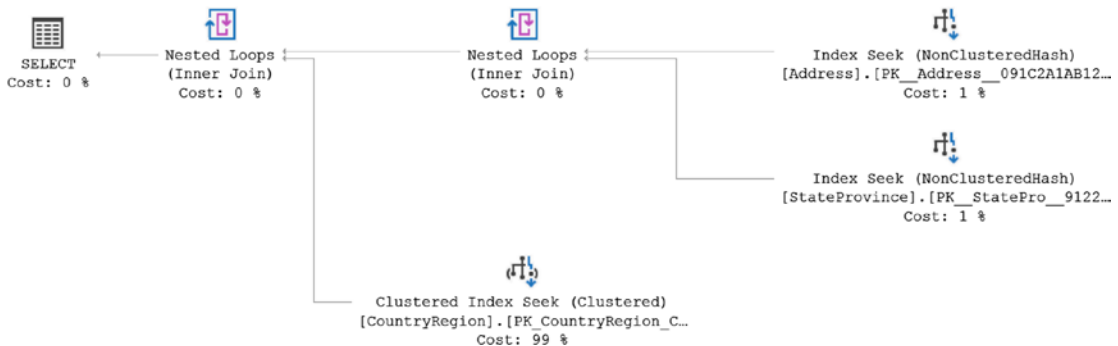


Figure 24-2. An execution plan showing both in-memory and standard tables

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

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

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

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

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

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

Figure 24-4. Query metrics for a regular table

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

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

In-Memory Indexes

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

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

Hash Index

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

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



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

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

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

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

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

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

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

Figure 24-6 shows the results of this query.

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

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

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

Nonclustered Indexes

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

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

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

Currently the performance looks like Figure 24-7.

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

Figure 24-7. Metrics of query without an index

Figure 24-8 shows the execution plan.

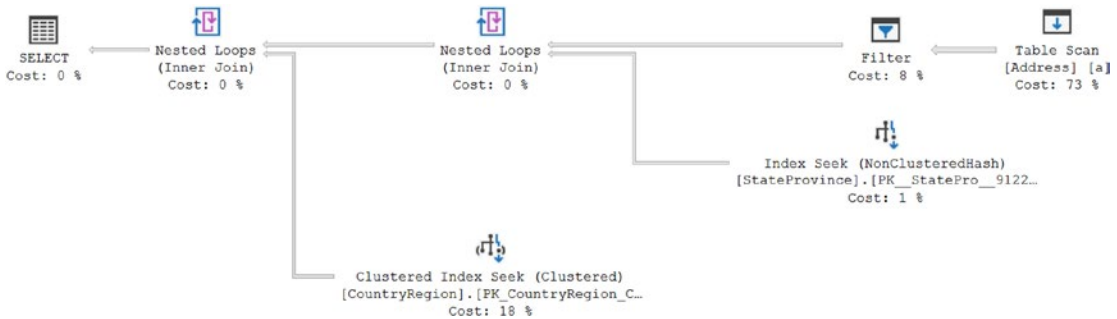


Figure 24-8. Query results in an execution plan that has table scans

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

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

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


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

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

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

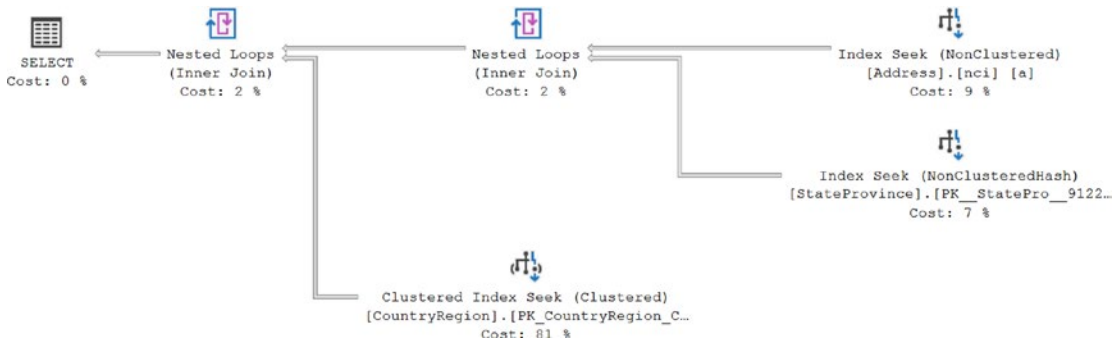


Figure 24-9. An improved execution plan taking advantage of nonclustered indexes

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

Columnstore Index

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

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

Statistics Maintenance

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

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

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

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

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

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

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

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

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

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

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

Natively Compiled Stored Procedures

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

Recommendations

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

Baselines

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

Correct Workload

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

Memory Optimization Advisor

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

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

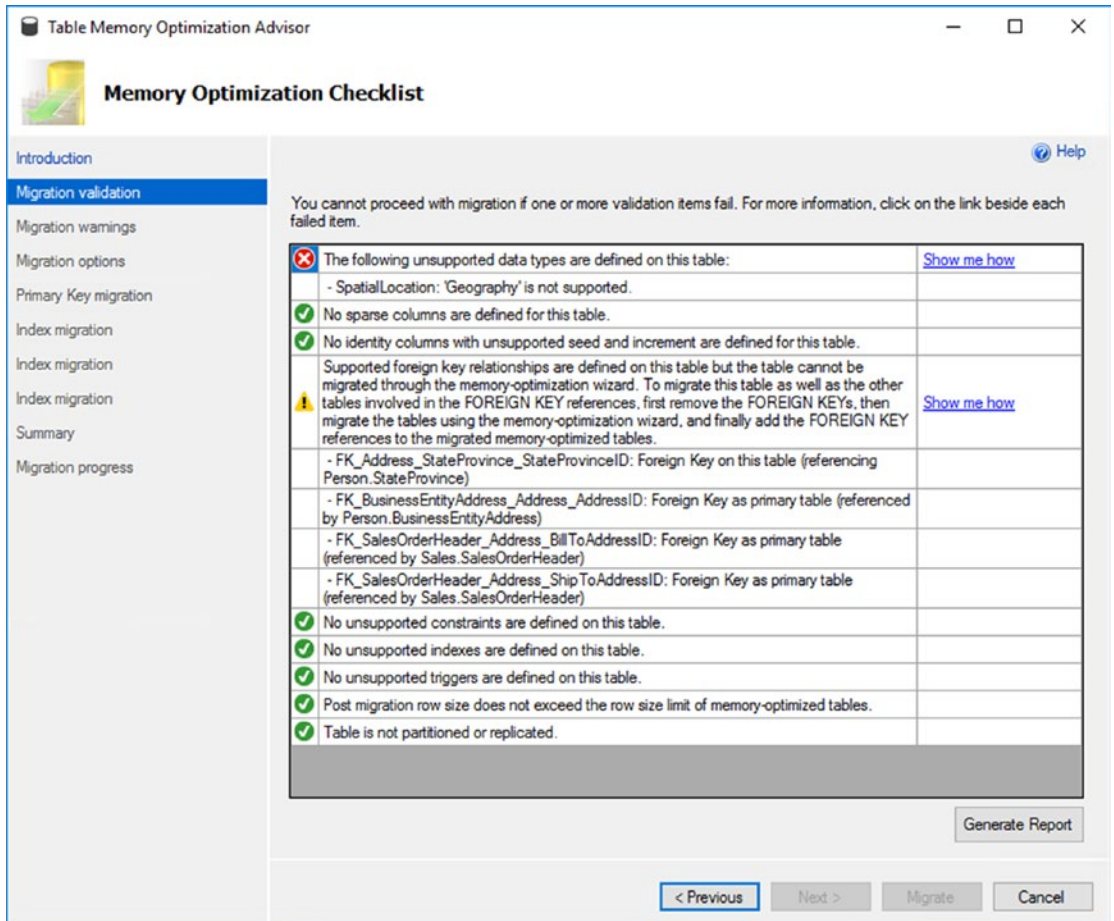


Figure 24-14. Table Memory Optimization Advisor showing all the unsupported data types

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

```
USE InMemoryTest;
GO

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

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










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

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

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






	A user transaction that accesses memory-optimized tables cannot access more than one user database.	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:

Logical file name:

File path:

Rename the original table as:

Estimated current memory cost (MB):

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

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

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

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

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

Please choose the appropriate conversion for this primary key:

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

Select a new name for this primary key:

Select the type of this primary key:

☒ Use NONCLUSTERED HASH index

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

Bucket Count:

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

☐ Use NONCLUSTERED index

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

Sort column and order:

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

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

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

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

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

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

Native Compilation Advisor

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

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

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

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

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

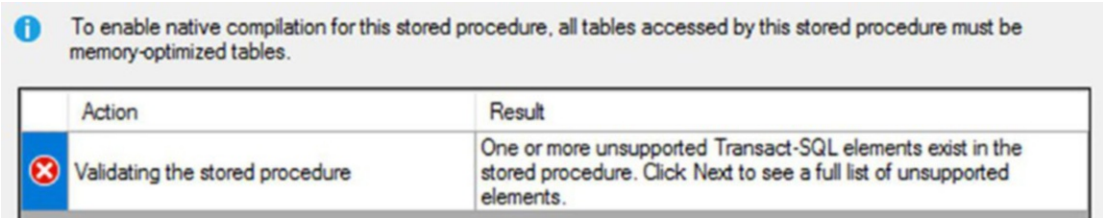


Figure 24-20. The Native Compilation Advisor has identified inappropriate T-SQL syntax