

# 1

## Optimizing for performance, scalability and real-time insights

Companies are optimizing their computing resources to get more transactional performance out of the same hardware resources. At the same time, the demand and pace of business and customer focus is increasing; they need real-time insights on the transactional data.

In recent years, many companies have turned to No-SQL solutions that allow very high write performance of transactions while allowing eventual consistency, but that later require data mining and analysis.

Microsoft SQL Server has taken on this challenge and, with every release, continues to expand the workloads in many dimensions. This chapter will discuss many of the features that allow both high-performance transaction processing while simultaneously allowing real-time analytics on transactional data without the need for a separate set of ETL processes, a separate data warehouse, and the time to do that processing.

Microsoft SQL Server 2019 is built on a database engine that is number one for TPC-E (On-Line Transaction Processing Benchmark) and TCP-H (Decision Support Benchmark). See <http://www.tpc.org> for more information.

Changes in hardware architecture allow dramatic speed increases with Hybrid Buffer Pool, which utilizes persistent memory (PMEM), also known as **Storage Class Memory (SCM)**.

Microsoft SQL Server 2019 can be used in the most demanding computing environments required today. Using a variety of features and techniques, including in-memory database operations, can make dramatic increases in your transaction processing rate while still allowing near-real-time analysis without having to move your transaction data to another "data warehouse" for reporting and analysis.

Microsoft SQL Server 2019 has also expanded the number of opportunities to tune database operations automatically, along with tools and reports to allow monitoring and optimization of queries and workloads. Comprehensive diagnostic features including Query Store allow SQL Server 2019 to identify performance issues quickly.

By upgrading to SQL Server 2019, the customer will be able to boost query performance without manual tuning or management. **Intelligent Query Processing (IQP)** helps many workloads to run faster without making any changes to the application.

### **Hybrid transactional and analytical processing (HTAP)**

**Hybrid transactional and analytical processing (HTAP)**, is the application of tools and features to be able to analyze live data without affecting transactional operations.

In the past, data warehouses were used to support the reporting and analysis of transactional data. A data warehouse leads to many inefficiencies. First, the data has to be exported from the transactional database and imported into a data warehouse using ETL or custom tools and processes. Making a copy of data takes more space, takes time, may require specialized ETL tools, and requires additional processes to be designed, tested, and maintained. Second, access to analysis is delayed. Instead of immediate access, business decisions are made, meaning the analysis may be delayed by hours or even days. Enterprises can make business decisions faster when they can get real-time operational insights. In some cases, it may be possible to affect customer behavior as it is happening.

Microsoft SQL Server 2019 provides several features to enable HTAP, including memory-optimized tables, natively compiled stored procedures, and Clustered Columnstore Indexes.

This chapter covers many of these features and will give you an understanding of the technology and features available.

A more general discussion of HTAP is available here: [https://en.wikipedia.org/wiki/Hybrid\\_transactional/analytical\\_processing\\_\(HTAP\)](https://en.wikipedia.org/wiki/Hybrid_transactional/analytical_processing_(HTAP)).

## Clustered Columnstore Indexes

Clustered Columnstore indexes can make a dramatic difference and are the technology used to optimize real-time analytics. They can achieve an order of magnitude performance gain over a normal row table, a dramatic compression of the data, and minimize interference with real-time transaction processing.

A columnstore has rows and columns, but the data is stored in a column format.

A rowgroup is a set of rows that are compressed into a columnstore format – a maximum of a million rows (1,048,576).

There are an optimum number of rows in a rowgroup that are stored column-wise, and this represents a trade-off between large overhead, if there are too few rows, and an inability to perform in-memory operations if the rows are too big.

Each row consists of column segments, each of which represents a column from the compressed row.

Columnstore is illustrated in *Figure 1.1*, showing how to load data into a non-clustered columnstore index:

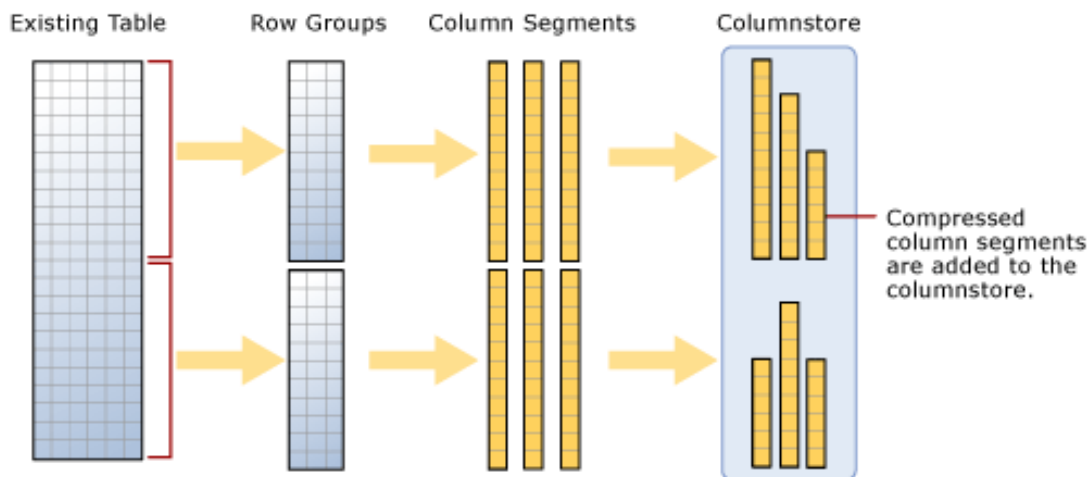


Figure 1.1: Loading data into a non-clustered columnstore index

A clustered columnstore index is how the columnstore table segments are stored in physical media. For performance reasons, and to avoid fragmenting the data, the columnstore index may store some data in a deltastore and a list of the IDs of deleted rows. All deltastore operations are handled by the system and not visible directly to the user. Deltastore and columnstore data is combined when queried.

A delta rowgroup is used to store columnstore indexes until there are enough to store in the columnstore. Once the maximum number of rows is reached, the delta rowgroup is closed, and a background process detects, compresses, and writes the delta rowgroup into the columnstore.

There may be more than one delta rowgroup. All delta rowgroups are described as the deltastore. While loading data, anything less than 102,400 rows will be kept in the deltastore until they group to the maximum size and are written to the columnstore.

Batch mode execution is used during a query to process multiple rows at once.

Loading a clustered columnstore index and the deltastore are shown in *Figure 1.2*.

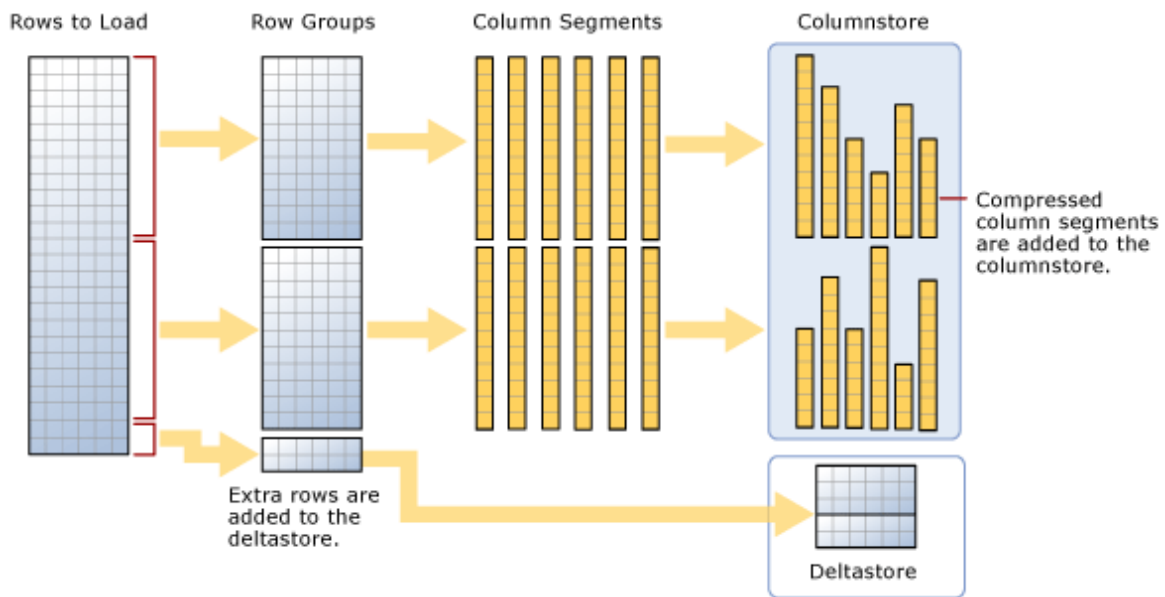


Figure 1.2: Loading a clustered columnstore index

Further information can be found here: <https://docs.microsoft.com/en-us/sql/relational-databases/indexes/get-started-with-columnstore-for-real-time-operational-analytics?view=sql-server-2017>.

## Adding Clustered Columnstore Indexes to memory-optimized tables

When using a memory-optimized table, add a non-clustered columnstore index. A clustered columnstore index is especially useful for running analytics on a transactional table.

A clustered columnstore index can be added to an existing memory-optimized table, as shown in the following code snippet:

```
-- Add a clustered columnstore index to a memory-optimized table
```

```
ALTER TABLE MyMemOpttable
```

```
ADD INDEX MyMemOpt_ColIndex clustered columnstore
```

## Disk-based tables versus memory-optimized tables

There are several differences between memory-optimized and disk-based tables.

One difference is the fact that, in a disk-based table, rows are stored in 8k pages and a page only stores rows from a single table. With memory-optimized tables, rows are stored individually, such that one data file can contain rows from multiple memory-optimized tables.

Indexes in a disk-based table are stored in pages just like data rows. Index changes are logged, as are data row changes. A memory-optimized table persists the definition of the index but is regenerated each time the memory-optimized table is loaded, such as restarting the database. No logging of index "pages" is required.

Data operations are much different. With a memory-optimized table, all operations are done in memory. Log records are created when an in-memory update is performed. Any log records created in-memory are persisted to disk through a separate thread. Disk-based table operations may perform in-place updates on non-key-columns, but key-columns require a delete and insert. Once the operation is complete, changes are flushed to disk.

With disk-based tables, pages may become fragmented. As changes are made, there may be partially filled pages and pages that are not consecutive. With memory-optimized tables, storing as rows removes fragmentation, but inserts, deletes, and updates will leave rows that can be compacted. Compaction of the rows is executed by means of a merge thread in the background.

Additional information can be found at this Microsoft docs link:

<https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/comparing-disk-based-table-storage-to-memory-optimized-table-storage?view=sql-server-2017>.

## In-memory OLTP

In-memory **on-line transaction processing (OLTP)** is available in Microsoft SQL Server for optimizing the performance of transaction processing. In-memory OLTP is also available for all premium Azure SQL databases. While dependent on your application, performance gains of 2-30x have been observed.

Most of the performance comes from removing lock and latch contention between concurrently executing transactions and is optimized for in-memory data. Although performed in-memory, changes are logged to disk so that once committed, the transaction is not lost even if the machine should fail.

To fully utilize in-memory OLTP, the following features are available:

- Memory-optimized tables are declared when you create the table.
- Non-durable tables, basically in-memory temporary tables for intermediate results, are not persisted so that they do not use any disk I/O. A non-durable table is declared with **DURABILITY=SCHEMA\_ONLY**.
- Table values and table-valued parameters can be declared as in-memory types as well.
- Natively compiled stored procedures, triggers, and scalar user-defined functions are compiled when created and avoid having to compile them at execution time, thereby speeding up operations.

Additional information can be found at the following links:

- <https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/in-memory-oltp-in-memory-optimization?view=sql-server-2017>
- <https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/survey-of-initial-areas-in-in-memory-oltp?view=sql-server-2017>

## Planning data migration to memory-optimized tables

Microsoft **SQL Server Management Studio (SSMS)** contains tools to help analyze and migrate tables to memory-optimized storage.

When you right-click on a database in SSMS and click on **Reports | Standard Reports | Transaction Performance Analysis Overview**, a four-quadrant report of all tables in the database will be made:

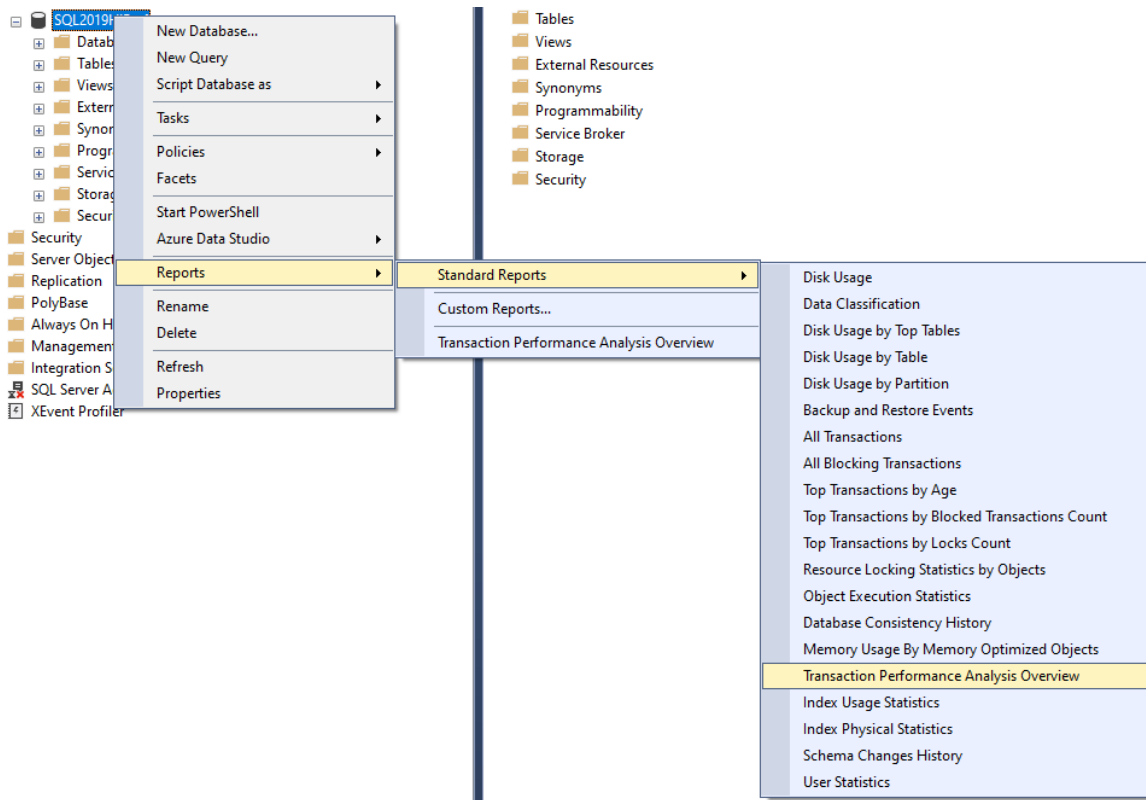


Figure 1.3: Choosing Transaction Performance Analysis

The report will look at each table and place it on the chart to show the ease of migration versus the expected gain by migrating the table to be memory-optimized:

### Recommended Tables Based on Usage

[SQL2019HiPerf]

on PLOVER8SSD\SQL2019RC at 8/31/2019 11:07:04 AM

The following chart contains the top candidate tables for memory optimization based on the access patterns of your workload. The horizontal axis represents decreasing effort of memory optimization, while the vertical axis represents increasing benefits of memory optimization in your workload. You should prioritize the tables in the top right corner of the chart for memory optimization.

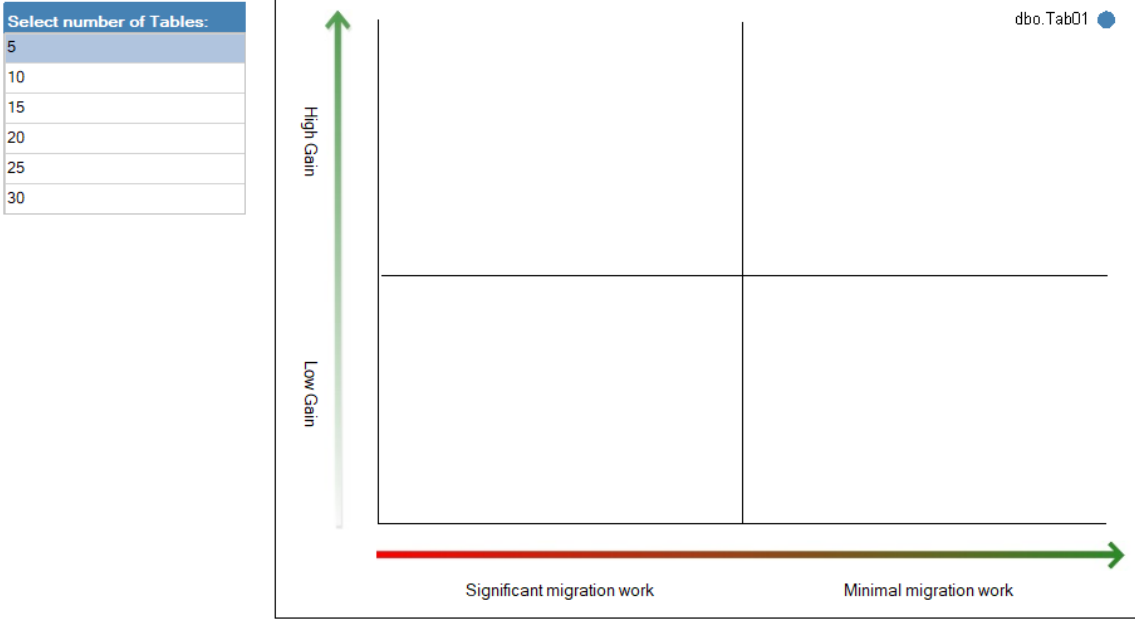


Figure 1.4: Recommended Tables Based on Usage



Once you have identified tables that might benefit, you can right-click on individual tables and run the Memory Optimization Advisor:

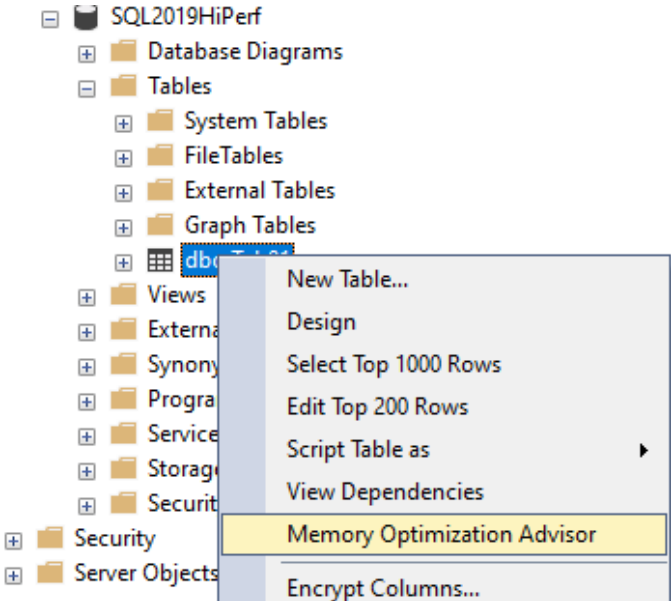


Figure 1.5: Selecting the Memory Optimization Advisor

The **Table Memory Optimization Advisor** is a "wizard" style of user interface that will step you through the configurations:

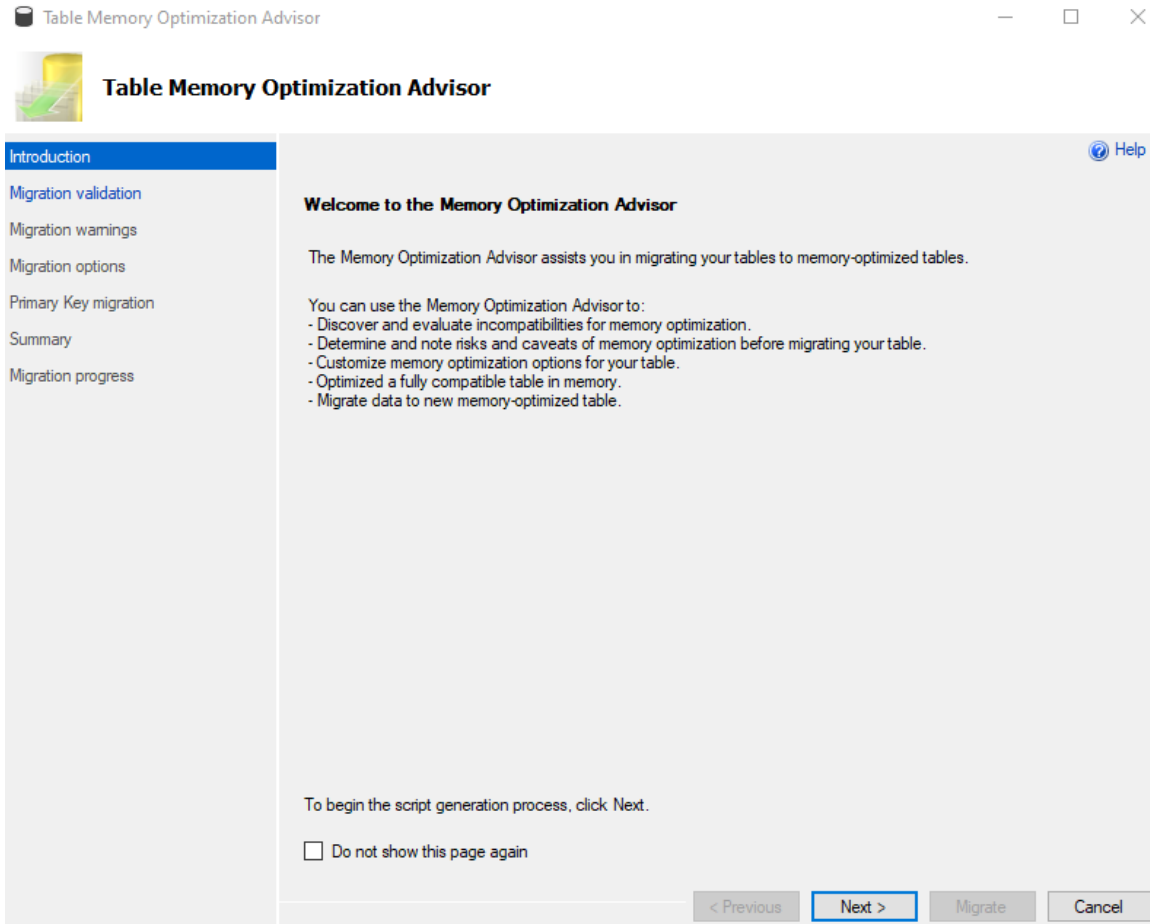


Figure 1.6: The Table Memory Optimization Advisor dialogue

The wizard will take you through a checklist with any failed issues:

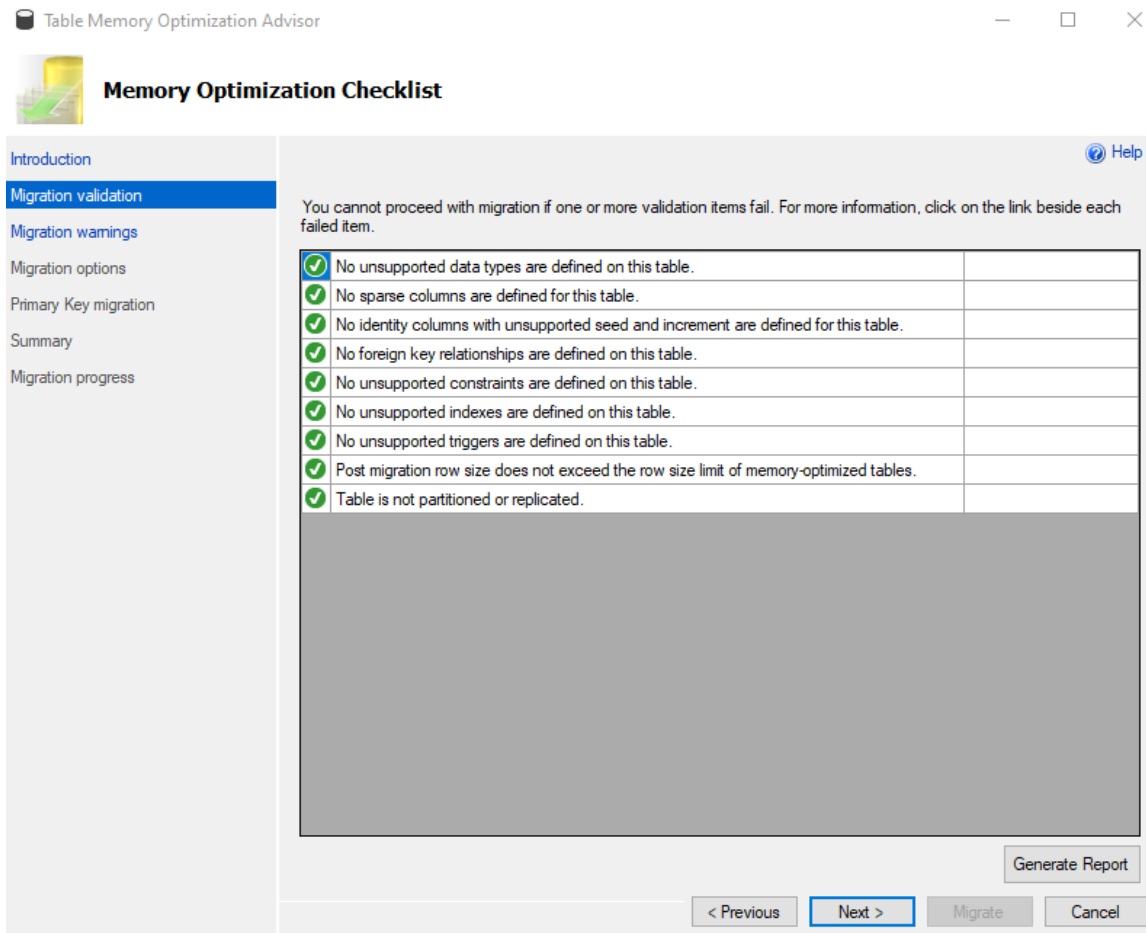


Figure 1.7: Memory Optimization Checklist

The warnings dialogue will flag up other important issues.

The screenshot shows a dialog box titled "Table Memory Optimization Advisor" with a "Memory Optimization Warnings" section. The dialog has a sidebar on the left with navigation options: Introduction, Migration validation, Migration warnings (selected), Migration options, Primary Key migration, Summary, and Migration progress. The main content area displays a list of warnings, each with an information icon and a "More information" link. Below the list is a "Generate Report" button and a set of navigation buttons: "< Previous", "Next >" (highlighted), "Migrate", and "Cancel".

Table Memory Optimization Advisor

### Memory Optimization Warnings

Introduction  
Migration validation  
**Migration warnings**  
Migration options  
Primary Key migration  
Summary  
Migration progress

For more information, click on the link beside each warning.

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

Generate Report

< Previous   **Next >**   Migrate   Cancel

Figure 1.8: Memory Optimization Warnings

Next enter file names and check paths in the migration option dialogue.

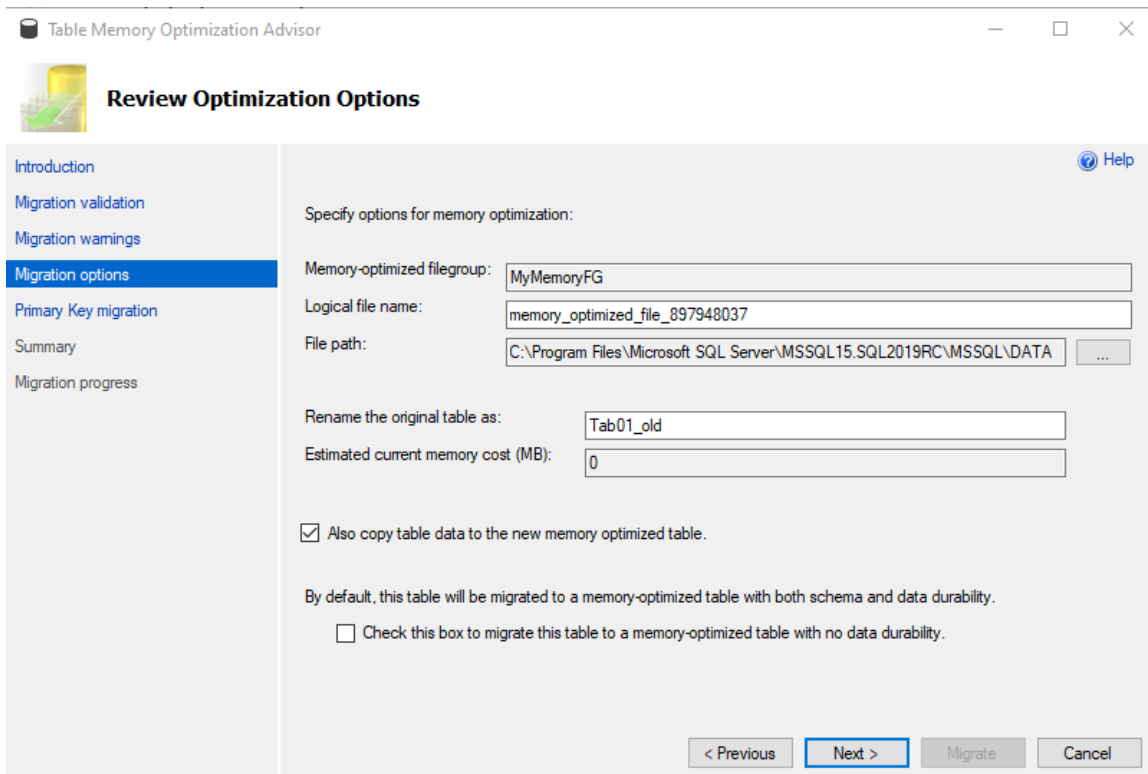


Figure 1.9: Review Optimization options

The wizard will detect the primary keys and populates the list of columns based on the primary key metadata. To migrate to a durable memory-optimized table, a primary key needs to be created. If there is no primary key and the table is being migrated to a non-durable table, the wizard will not show this screen.

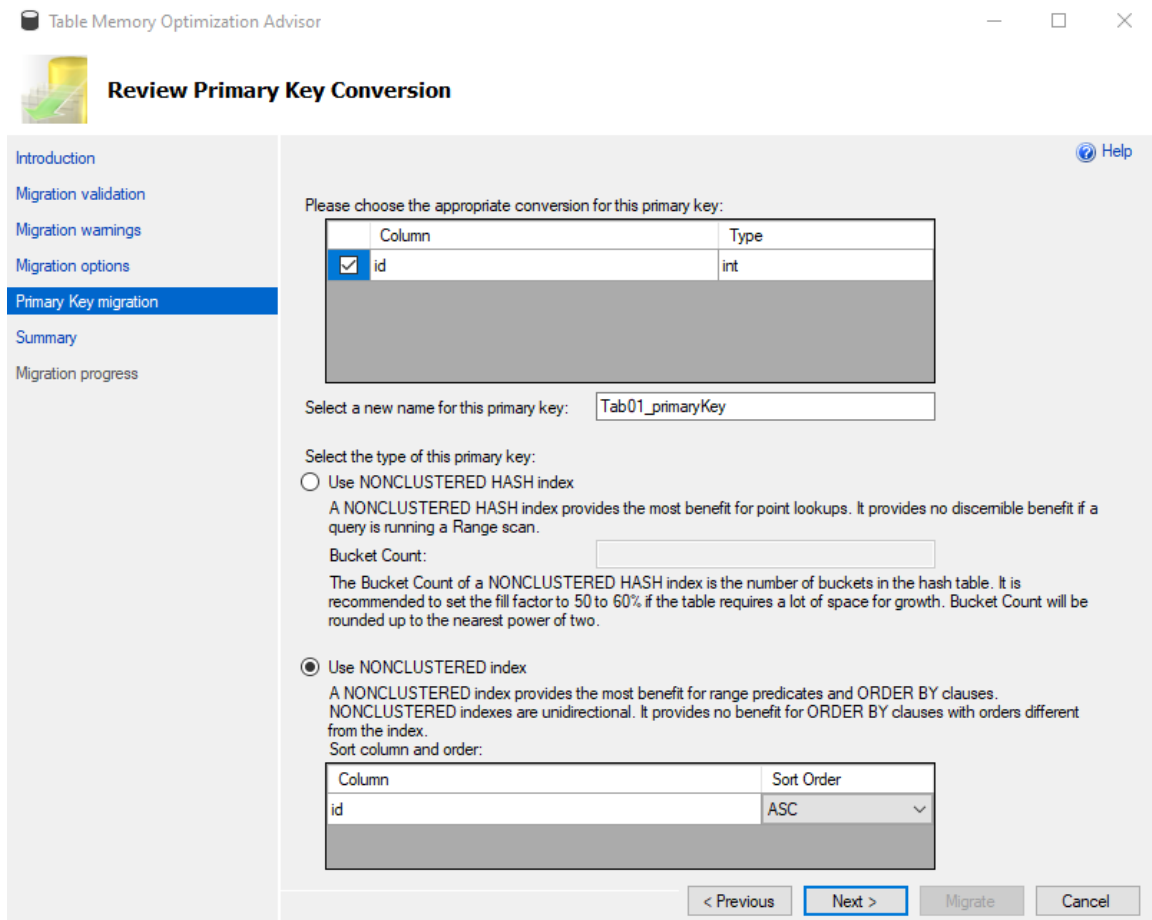


Figure 1.10: Review Primary Key Conversion

By clicking **Script** you can generate a Transact-SQL script in the summary screen.

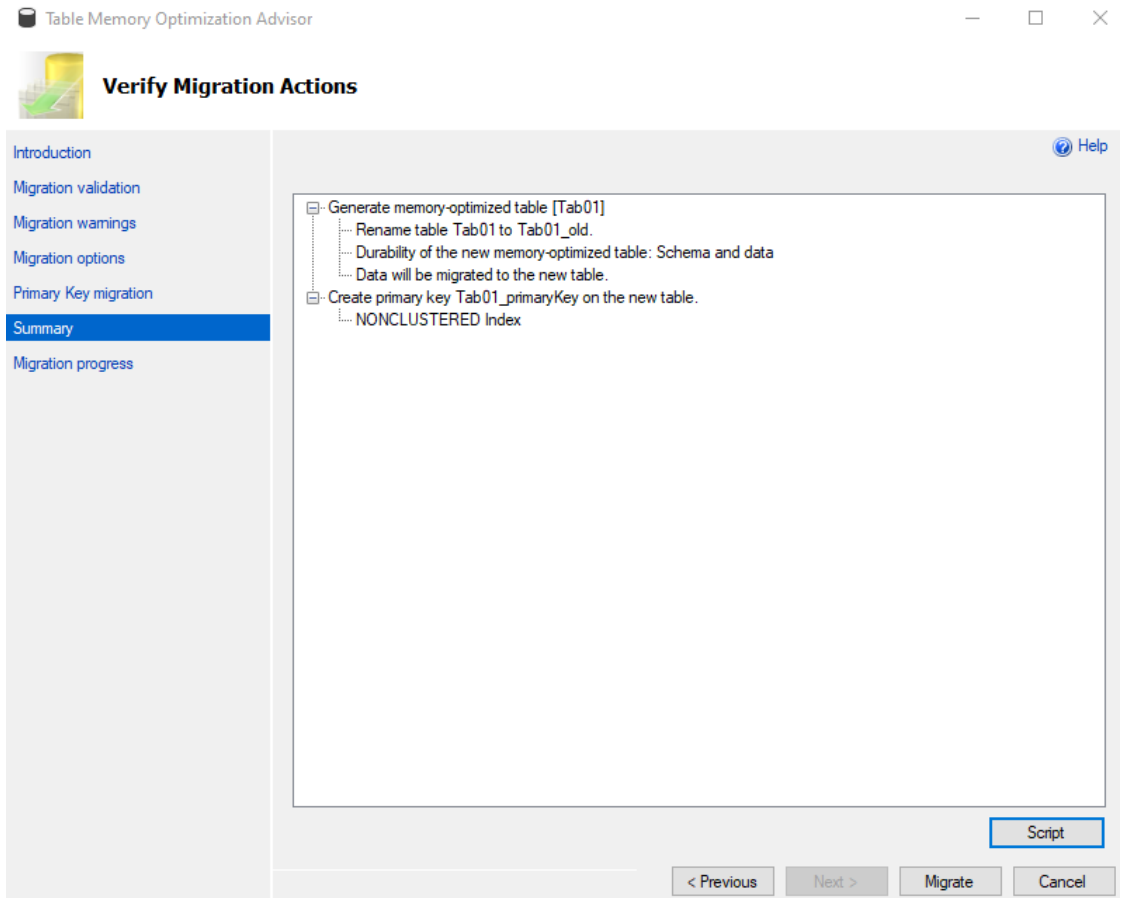
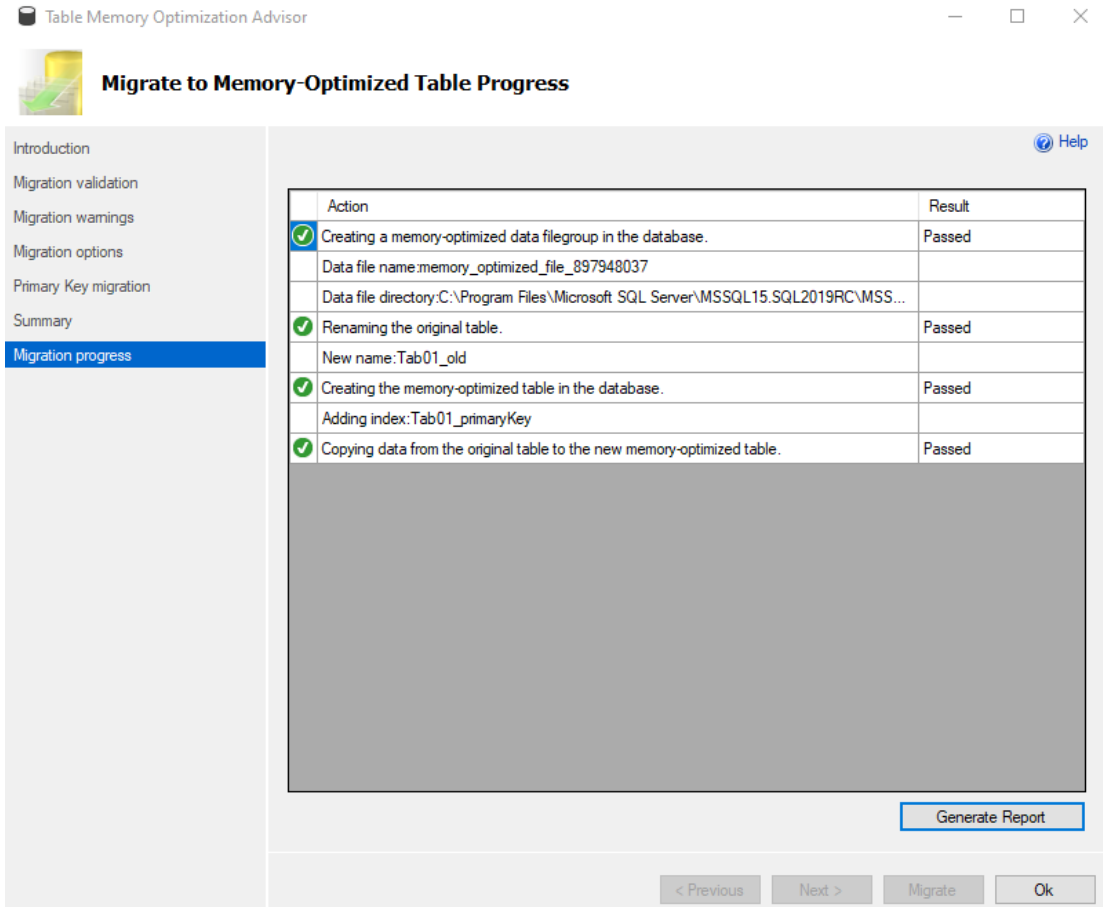


Figure 1.11: Verify Migration Actions Summary Screen

The wizard will the display a report as the table migrates.



The screenshot shows the 'Table Memory Optimization Advisor' window. The title bar reads 'Table Memory Optimization Advisor'. The main window title is 'Migrate to Memory-Optimized Table Progress'. On the left, there is a navigation pane with the following items: Introduction, Migration validation, Migration warnings, Migration options, Primary Key migration, Summary, and Migration progress (which is highlighted in blue). The main area displays a progress report table with the following data:

Action	Result
<input checked="" type="checkbox"/> Creating a memory-optimized data filegroup in the database. Data file name:memory_optimized_file_897948037 Data file directory:C:\Program Files\Microsoft SQL Server\MSSQL15.SQL2019RC\MSS...	Passed
<input checked="" type="checkbox"/> Renaming the original table. New name:Tab01_old	Passed
<input checked="" type="checkbox"/> Creating the memory-optimized table in the database. Adding index:Tab01_primaryKey	Passed
<input checked="" type="checkbox"/> Copying data from the original table to the new memory-optimized table.	Passed

Below the table is a large grey rectangular area, likely a placeholder for a detailed report or logs. At the bottom right of the main area is a 'Generate Report' button. At the bottom of the window are navigation buttons: '< Previous', 'Next >', 'Migrate', and 'Ok'. A 'Help' icon is visible in the top right corner of the main area.

Figure 1.12: Migration progress report

Memory-optimized tables are a great feature, but you will need to plan carefully to make sure you get the performance and transactional reliability you require.

You can create a new database specifying memory-optimized, or alter an existing database to handle memory-optimized data. In either case, a filegroup for containing the memory-optimized data must be created.



In the following sample, we will create a memory-optimized database using SQL script:

```
-- Create Memory-Optimized Database
USE MASTER;

GO

CREATE DATABASE MemOptDB
    ON (Name = MemOptDB_Data, FileName = 'c:\sqldata\memoptdb_data.mdf', size
    = 10 mb, maxsize = 20 mb, filegrowth = 5 mb)
    LOG ON (Name = MemOptDB_Log, FileName = 'c:\sqldata\memoptdb_log.ldf',
    size = 2 mb, maxsize = 10 mb, filegrowth = 1 mb);

GO

-- Must declare a memory-optimized filegroup
ALTER DATABASE MemOptDB
    ADD FILEGROUP MemOptDB_FG contains MEMORY_OPTIMIZED_DATA;

ALTER DATABASE MemOptDB
    ADD FILE (Name = 'MemOptDB_MOFG', FileName = 'c:\sqldata\memoptdb_mofg')
    TO FILEGROUP MemOptDB_FG;

ALTER DATABASE MemOptDB
    SET MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT = ON;

GO
```

You can also make a memory-optimized database by using SQL Server Management Studio and adding a memory-optimized filegroup:

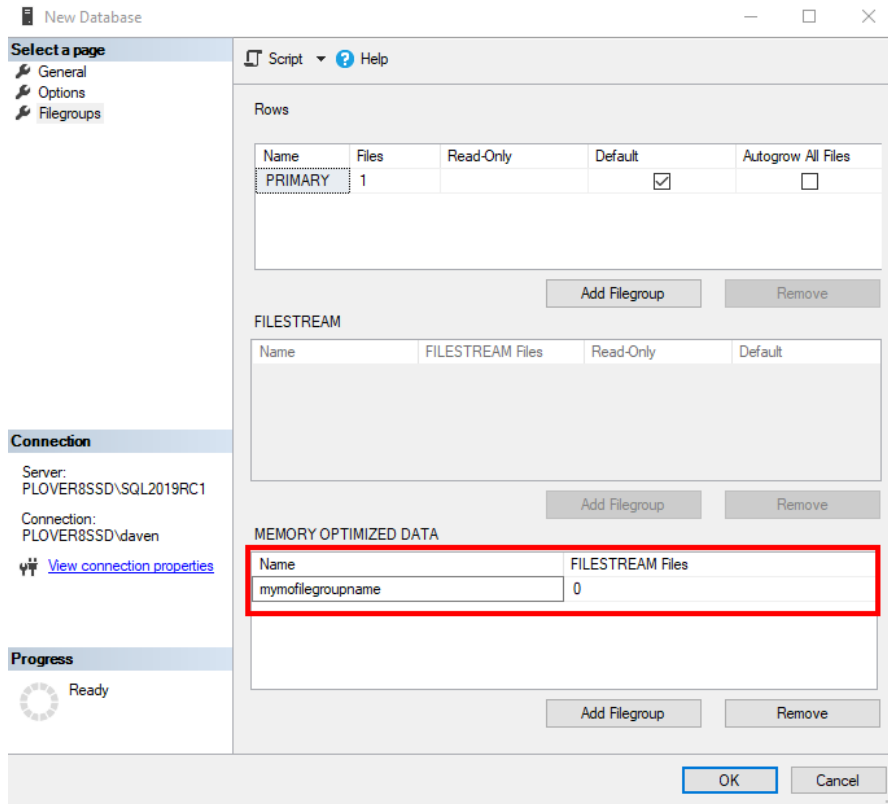


Figure 1.13: The new database dialogue window

## Natively compiled stored procedures

Natively compiled stored procedures are compiled when created and bypass the query execution engine. The procedure is compiled when created, and also manually or when the database or server are restarted.

A few additional concepts are introduced here, including **SCHEMABINDING** and **BEGIN ATOMIC**, both of which are required for natively compiled stored procedures.

SCHEMABINDING locks the table definition to prevent alteration after the stored procedure is created. SCHEMABINDING allows the compiled stored procedure to be certain of the data types involved. The tables involved in the natively compiled stored procedure cannot be altered without dropping the SCHEMABINDING, making changes and then reapplying the SCHEMABINDING. SCHEMABINDING also requires that explicit field names are used in the query; "**select \***..." will not work.

BEGIN ATOMIC is required in a natively compiled stored procedure and is only available for a natively compiled stored procedure. In interactive (non-natively compiled) procedures, you would use a BEGIN TRAN statement block. Using the ATOMIC block and transaction settings will be independent of the current connection/settings as the stored procedure may be used in different execution sessions.

If there is an existing active transaction, BEGIN ATOMIC will set a save point and roll back to that if it fails. Otherwise, a new transaction is created and completed or rolled back.

You indicated a natively compiled stored procedure in the create declaration of the stored procedure using the "NATIVE\_COMPILATION" directive.

In the following sample, we will create a memory-optimized table and a natively stored procedure. Note that memory-optimized tables cannot have clustered indexes. Memory-optimized tables are stored as rows, not in pages, as with a disk-based table:

```
-- Create Memory-Optimized Table
USE MemOptDB;

GO

CREATE TABLE dbo.MyMemOptTable
(
    id int not null,
    dtCreated datetime not null,
```

```
orderID nvarchar(10) not null
CONSTRAINT pk_id PRIMARY KEY NONCLUSTERED (id)
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
```

GO

```
-- Create Natively Stored Procedure
CREATE PROCEDURE dbo.myNativeProcedure (@id int)
WITH NATIVE_COMPILATION, SCHEMABINDING
AS BEGIN ATOMIC WITH ( TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE =
N'us_english' )
```

```
SELECT id, dtCreated, orderID
FROM dbo.MyMemOptTable
WHERE id = @id
```

END

GO

The table schema is locked due to the reference to a natively compiled stored procedure. If you try to alter the table, an exception will be thrown, as shown here:

```
-- Try to alter the schema!
ALTER TABLE [dbo].[MyMemOpttable]
ALTER COLUMN orderId nvarchar(20)
GO
```

Msg 5074, Level 16, State 1, Line 55

The object 'myNativeProcedure' is dependent on column 'orderId'.

Msg 4922, Level 16, State 9, Line 55

ALTER TABLE ALTER COLUMN orderId failed because one or more objects access this column.

More information on natively compiled procedures can be found here:

<https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/creating-natively-compiled-stored-procedures?view=sql-server-2017>.

## TempDB enhancements

We have introduced another scalability enhancement with memory-optimized TempDB metadata. Historically, TempDB metadata contention has been a bottleneck to scalability for workloads running on SQL Server.

The system tables used for managing temp table metadata can be moved into latch-free non-durable memory-optimized tables.

### Enabling memory-optimized TempDB metadata

Enabling this feature in SQL Server is a two-step process:

- First, alter the server configuration with T-SQL
- Restart the service

```
ALTER SERVER CONFIGURATION SET MEMORY_OPTIMIZED tempdb_METADATA = ON
```

The following T-SQL command can be used to verify whether **tempdb** is memory-optimized:

```
SELECT SERVERPROPERTY('IsTempdbMetadataMemoryOptimized')
```

### Limitations of memory-optimized TempDB metadata

There are a few limitations associated with using this new feature.

- Toggling the feature on and off requires a service restart.
- A single transaction may not access memory-optimized tables in more than one database. This means that any transactions that involve a memory-optimized table in a user database will not be able to access TempDB System views in the same transaction. If you attempt to access TempDB system views in the same transaction as a memory-optimized table in a user database, you will receive the following error:

```
A user transaction that accesses memory-optimized tables or natively compiled modules cannot access more than one user database or databases model and msdb, and it cannot write to master.
```

- Queries against memory-optimized tables do not support locking and isolation hints, so queries against memory-optimized TempDB catalog views will not honor locking and isolation hints. As with other system catalog views in SQL Server, all transactions against system views will be in READ COMMITTED (or, in this case, READ COMMITTED SNAPSHOT) isolation.
- There may be some issues with columnstore indexes on temporary tables when memory-optimized TempDB metadata is enabled. It is best to avoid columnstore indexes on temporary tables when using memory-optimized TempDB metadata.

## Intelligent Query Processing

**Intelligent Query Processing (IQP)** is a family of features that were introduced in Microsoft SQL Server 2017 as adaptive query processing and has been expanded with new features in Microsoft SQL Server 2019. By upgrading to SQL Server 2019 and with compatibility level 150, most workloads will see performance improvements due to added intelligence in the query optimizer.

Intelligent Query Processing features are automatically enabled based on the "COMPATIBILITY\_LEVEL" of the database. To take advantage of the latest IQP features, set the database compatibility to 150.

Most of these are also available in Azure SQL, but it is best to check current documentation on exactly what is available there as this changes.

The following table summarizes some of the IQP features.

Feature Name	Available	Feature Description
Memory Grant Feedback - Batch Mode	SQL 2017	This feature will adjust the amount of memory granted in a query's execution plan and, after the first execution, will set a new minimum memory for the query plan that may be much less than the original estimate. This feature helps to conserve memory while providing an optimum amount for query execution.
Memory Grant Feedback - Row Mode	SQL 2019	Building on the batch mode memory grant feedback in SQL 2017, row mode will adjust the query plan for the next execution if the actual memory used is less than 50% of the granted memory.
Batch Mode on Rowstore	SQL 2019	Batch mode works in conjunction with columnstore indexes to process columnstore queries. The query processor now will consider batch mode when optimizing rowstore queries. Using batch mode is done dynamically, and the optimizer may decide not to use it.
Approximate Count distinct	SQL 2019	This new function is used to get an approximate on very large datasets where a quick response is more important than the exact number.
Scalar UDF Inlining	SQL 2019	Scalar user-defined function (UDF) is now transformed into a relational expression and optimized along with the execution query. This new processing allows another level of optimization and parallel execution that achieves much better performance.
Table Variable Deferred Compilation	SQL 2019	This change in SQL 2019 waits to compile a table variable used in a statement is executed. At that point, the actual number of rows involved (the cardinality) is known, resulting in better query optimization. Previously the compilation was based on a onerow guess.
Adaptive Joins – Batch Mode	SQL 2017	Dynamically selects nested or hashed joins after reading a specified number of rows. A nested join is appropriate for a small number of rows, but if the join contains a large number of rows, a hashed join is more efficient.

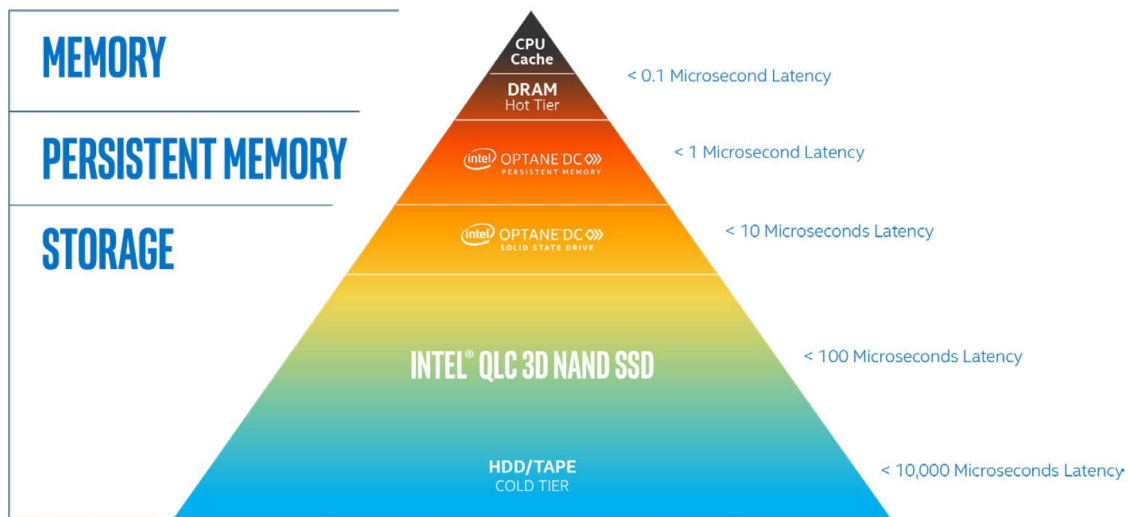
Table 1.14: Table summarizing IQP features

- These features can be disabled and monitored.
- For more information, refer to <https://docs.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing?view=sql-server-2017>.

## Hybrid Buffer Pool

Microsoft SQL Server 2019 introduces Hybrid Buffer Pool. This feature allows access to **Persistent Memory (PMEM)** devices. These persistent memory devices add a new layer to server memory hierarchy and filling the gap between high performance / high cost of DRAM (Dynamic Random Access Memory) and the lower cost lower performance of file storage drives using SSD.

This memory architecture has been implemented by Intel as Intel® Optane™ Technology; refer to <https://www.intel.com/content/www/us/en/products/docs/storage/optane-technology-brief.html> for more information:



*Intel® Optane™ technology fills memory and performance gaps in the data center*

Figure 1.15: Intel memory architecture

Persistent memory is integrated at the memory controller level of the CPU chip and will retain data even when the server is powered off.

While many aspects of persistent memory devices can be realized without any software changes, features such as Hybrid Buffer Pool can take advantage of the new storage hierarchy and provide direct memory access to files.

For clean database pages, those that have not been modified, SQL server can directly access them as memory. When an update is made, and then marked as dirty, the page is copied to DRAM, changes persisted, and the page is then written back into the persistent memory area.



To enable Hybrid Buffer Pool, the feature must be enabled at the instance level of SQL Server. It is off by default. After enabling, the instance must be restarted:

```
ALTER SERVER CONFIGURATION  
SET MEMORY_OPTIMIZED HYBRID_BUFFER_POOL = ON;
```

Furthermore, the Hybrid Buffer Pool will only operate on memory-optimized databases:

```
ALTER DATABASE <databaseName> SET MEMORY_OPTIMIZED = ON;
```

Or, in order to disable, execute the following command:

```
ALTER DATABASE <databaseName> SET MEMORY_OPTIMIZED = OFF;
```

To see the Hybrid Buffer Pool configurations and memory-optimized databases on an instance, you can run the following queries:

```
SELECT * FROM sys.configurations WHERE name = 'hybrid_buffer_pool';
```

```
SELECT name, is_memory_optimized_enabled FROM sys.databases;
```

There are many considerations when configuring a server with persistent memory, including the ratio of DRAM to PMEM. You can read more here:

- <https://docs.microsoft.com/en-us/windows-server/storage/storage-spaces/deploy-pmem>
- <https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/hybrid-buffer-pool?view=sql-server-2017>

## Query Store

The Query Store in SQL Server, first introduced in SQL Server 2016, streamlines the process of troubleshooting query execution plans. The Query Store, once enabled, automatically captures query execution plans and runtime statistics for your analysis. You can then use the `sys.dm_db_tuning_recommendations` view to discover where query execution plan regression has occurred and use the stored procedure, `sp_query_store_force_plan`, to force a specific plan that performs better.

In SQL Server 2019, we now have made some additional enhancements to the default Query Store features. In this section, we will discuss the following topics:

- Changes to default parameter values when enabling Query Store
- A new `QUERY_CAPTURE_MODE` custom
- Support for fast forward and static cursors

You can configure Query Store with SQL Server Management Studio (SSMS) or with T-SQL statements. SSMS configuration includes turning it on and off by setting the operation mode (off, read-only, or read/write), the Query Store size, and other settings. You can find Query Store parameters in the properties of a database by right-clicking on the database and selecting Query Store:

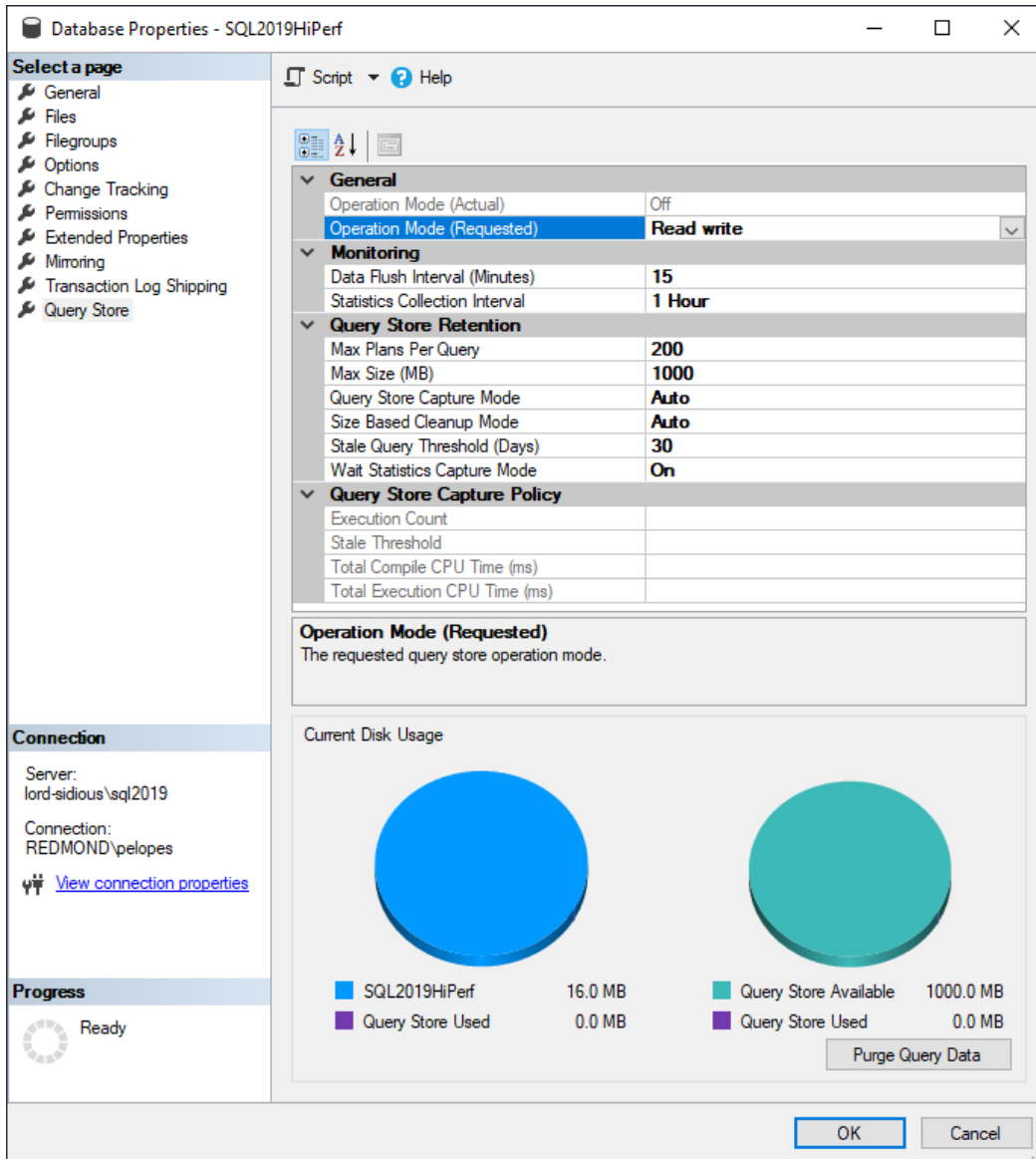


Figure 1.16: Database properties dialogue window

## Changes to default parameter values

Two of the existing parameters have new default values compared to SQL Server 2017. These parameters are **MAX\_STORAGE\_SIZE\_MB** and **QUERY\_CAPTURE\_MODE**. The new default values as of SQL Server 2019 are listed here:

- **MAX\_STORAGE\_SIZE\_MB** has a default value of 1000 (MB)
- The **QUERY\_CAPTURE\_MODE** has a default value of **AUTdO**

## QUERY\_CAPTURE\_MODE

In previous versions of SQL Server, the default value for the **QUERY\_CAPTURE\_MODE** was set to **ALL**, and therefore all query plans were captured and stored. As mentioned in the previous section, the default value has now been changed to **AUTO**.

Setting the **QUERY\_CAPTURE\_MODE** to **AUTO** means that no query plans or associated runtime statistics will be captured for the first 29 executions in a single day. Query plans and runtime statistics are not captured until the 30th execution of a plan. This default setting can be changed by using the new custom mode.

## QUERY\_CAPTURE\_MODE: CUSTOM

Before 2019, there were three available values for the **query\_capture\_mode**; those values were **NONE**, **ALL**, and **AUTO**. We have now added a fourth option, which is **CUSTOM**.

The **CUSTOM** mode provides you with a mechanism for changing the default settings of the Query Store. For example, the following settings can be modified when working in **CUSTOM** mode:

- **EXECUTION\_COUNT**
- **TOTAL\_COMPILE\_CPU\_TIME\_MS**
- **TOTAL\_EXECUTION\_CPU\_TIME\_MS**
- **STALE\_CAPTURE\_POLICY\_THRESHOLD**

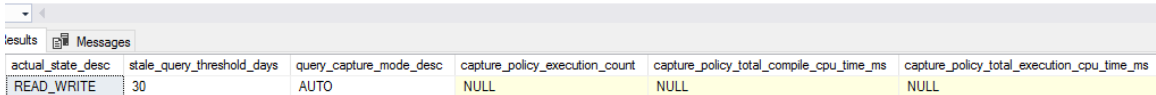
First, you can verify and validate the current Query Store settings by using the `sys.database_query_store_options` view:

```
SELECT actual_state_desc, stale_query_threshold_days, query_capture_mode_desc,
       capture_policy_execution_count, capture_policy_total_compile_cpu_time_ms,
       capture_policy_total_execution_cpu_time_ms
FROM sys.database_query_store_options
```

The output is as follows:

```
SELECT
  actual_state_desc,
  stale_query_threshold_days,
  query_capture_mode_desc,
  capture_policy_execution_count,
  capture_policy_total_compile_cpu_time_ms,
  capture_policy_total_execution_cpu_time_ms
FROM sys.database_query_store_options
```

USE MASTER



actual_state_desc	stale_query_threshold_days	query_capture_mode_desc	capture_policy_execution_count	capture_policy_total_compile_cpu_time_ms	capture_policy_total_execution_cpu_time_ms
READ_WRITE	30	AUTO	NULL	NULL	NULL

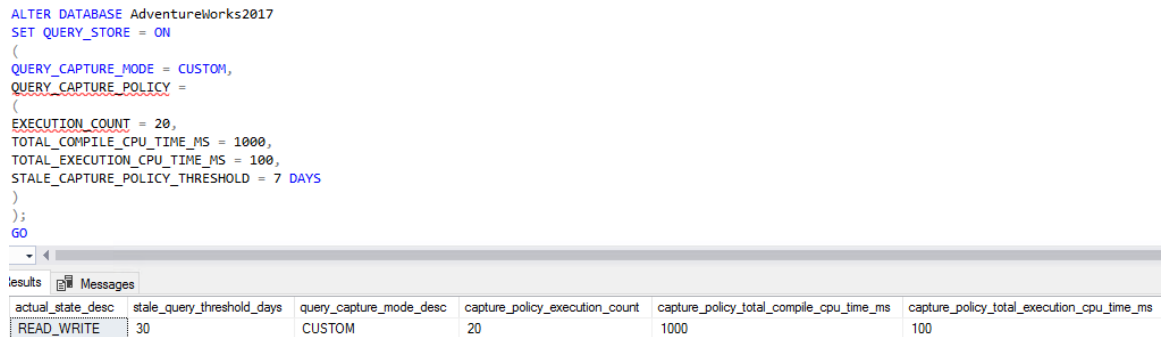
Figure 1.17: Verifying and validating the Query Store settings

To modify the default settings, you will first change the query capture mode to custom and then apply changes to the default values. Look at the following code by way of an example:

```
ALTER DATABASE AdventureWorks2017
SET QUERY_STORE = ON
(
  QUERY_CAPTURE_MODE = CUSTOM, QUERY_CAPTURE_POLICY =
  (
    EXECUTION_COUNT = 20,
    TOTAL_COMPILE_CPU_TIME_MS = 1000,
    TOTAL_EXECUTION_CPU_TIME_MS = 100,
    STALE_CAPTURE_POLICY_THRESHOLD = 7 DAYS
  )
);
```

The output is as follows:

```
ALTER DATABASE AdventureWorks2017
SET QUERY_STORE = ON
(
  QUERY_CAPTURE_MODE = CUSTOM,
  QUERY_CAPTURE_POLICY =
  (
    EXECUTION_COUNT = 20,
    TOTAL_COMPILE_CPU_TIME_MS = 1000,
    TOTAL_EXECUTION_CPU_TIME_MS = 100,
    STALE_CAPTURE_POLICY_THRESHOLD = 7 DAYS
  )
);
GO
```



actual_state_desc	stale_query_threshold_days	query_capture_mode_desc	capture_policy_execution_count	capture_policy_total_compile_cpu_time_ms	capture_policy_total_execution_cpu_time_ms
READ_WRITE	30	CUSTOM	20	1000	100

Figure 1.18: Modifying the default settings

## Support for FAST\_FORWARD and STATIC Cursors

We have added another exciting update to the Query Store. You can now force query execution plans for fast forward and static cursors. This functionality supports T-SQL and API cursors. Forcing execution plans for fast forward and static cursors is supported through SSMS or T-SQL using `sp_query_store_force_plan`.

## Automatic tuning

Automatic tuning identifies potential query performance problems, recommends solutions, and automatically fixes problems identified.

By default, automatic tuning is disabled and must be enabled. There are two automatic tuning features available:

- Automatic plan correction
- Automatic index management

### Automatic plan correction

To take advantage of automatic plan correction, the Query Store must be enabled on your database. Automatic plan correction is made possible by constantly monitoring data that is stored by the Query Store.

Automatic plan correction is the process of identifying regression in your query execution plans. Plan regression occurs when the SQL Server Query Optimizer uses a new execution plan that performs worse than the previous plan. To identify plan regression, the Query Store captures compile time and runtime statistics of statements being executed.

The database engine uses the data captured by the Query Store to identify when plan regression occurs. More specifically, to identify plan regression and take necessary action, the database engine uses the `sys.dm_db_tuning_recommendations` view. This is the same view you use when manually determining which plans have experienced regressions and which plans to force.

When plan regression is noticed, the database engine will force the last known good plan.

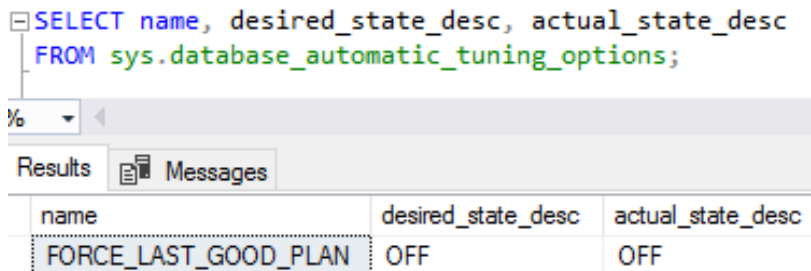
The great news is that the database engine doesn't stop there; the database engine will monitor the performance of the forced plan and verify that the performance is better than the regressed plan. If the performance is not better, then the database engine will unforce the plan and compile a new query execution plan.

### Enabling automatic plan correction

Automatic plan correction is disabled by default. The following code can be used to verify the status of automatic plan correction on your database:

```
SELECT name, desired_state_desc, actual_state_desc
FROM sys.database_automatic_tuning_options
```

The output is as follows:



```
SELECT name, desired_state_desc, actual_state_desc
FROM sys.database_automatic_tuning_options;
```

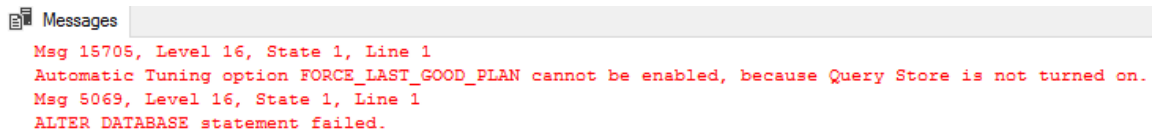
name	desired_state_desc	actual_state_desc
FORCE_LAST_GOOD_PLAN	OFF	OFF

Figure 1.19: Automatic plan correction is turned off

You enable automatic plan correction by using the following code:

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

If you have not turned the Query Store on, then you will receive the following error:



```

Messages
Msg 15705, Level 16, State 1, Line 1
Automatic Tuning option FORCE_LAST_GOOD_PLAN cannot be enabled, because Query Store is not turned on.
Msg 5069, Level 16, State 1, Line 1
ALTER DATABASE statement failed.

```

Figure: 1.20: Error report if the Query Store is off

## Automatically forced plans

The database engine uses two criteria to force query execution plans:

- Where the estimated CPU gain is higher than 10 seconds
- The number of errors in the recommended plan is lower than the number of errors in the new plan

Forcing execution plans improves performance where query execution plan regression has occurred, but this is a temporary solution, and these forced plans should not remain indefinitely. Therefore, automatically forced plans are removed under the following two conditions.

- Plans that are automatically forced by the database engine are not persisted between SQL Server restarts.
- Forced plans are retained until a recompile occurs, for example, a statistics update or schema change.

The following code can be used to verify the status of automatic tuning on the database:

```

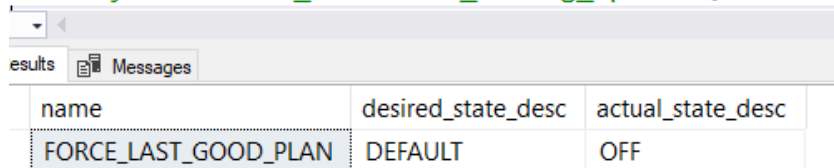
SELECT name, desired_state_desc, actual_state_desc
FROM sys.database_automatic_tuning_options;

```

```

SELECT name, desired_state_desc, actual_state_desc
FROM sys.database_automatic_tuning_options;

```



name	desired_state_desc	actual_state_desc
FORCE_LAST_GOOD_PLAN	DEFAULT	OFF

Figure 1.21: Verifying the status of automatic tuning on the database

## Lightweight query profiling

**Lightweight query profiling (LWP)** provides DBAs with the capability to monitor queries in real time at a significantly reduced cost of the standard query profiling method. The expected overhead of LWP is at 2% CPU, as compared to an overhead of 75% CPU for the standard query profiling mechanism.

For a more detailed explanation on the query profiling infrastructure, refer to <https://docs.microsoft.com/en-us/sql/relational-databases/performance/query-profiling-infrastructure?view=sqlallproducts-allversions>.

### New functionality in 2019

In SQL Server 2019, we have now improved LWP with new features and enhancements to the existing capabilities.

- In SQL Server 2016 and 2017, lightweight query profiling was deactivated by default and you could enable LWP at the instance level by using trace flag **7412**. In 2019, we have now turned this feature ON by default.
- You can also now manage this at the database level through Database Scoped Configurations. In 2019, you have a new database scoped configuration, **lightweight\_query\_profiling**, to enable or disable the **lightweight\_query\_profiling** infrastructure at the database level.
- We have also introduced a new extended event. The new **query\_post\_execution\_plan\_profile** extended event collects the equivalent of an actual execution plan based on lightweight profiling, unlike **query\_post\_execution\_showplan**, which uses standard profiling.
- We also have a new DMF **sys.dm\_exec\_query\_plan\_stats**; this DMF returns the equivalent of the last known actual execution plan for most queries, based on lightweight profiling.



The syntax for `sys.dm_exec_query_plan_stats` is as follows:

```
sys.dm_exec_query_plan_stats(plan_handle)
```

For a more detailed analysis, refer to this online documentation: <https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-exec-query-plan-stats-transact-sql?view=sql-server-2017>.

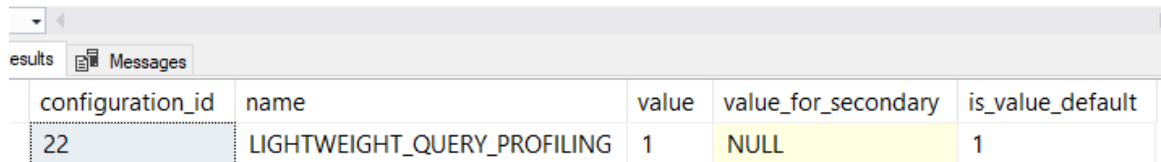
## sys.database\_scoped\_configurations

If you are not certain of the current status of LWP, you can use the following code to check the status of your database scoped configurations. The value column is 1; therefore, using the `sys.database_scoped_configurations` view, you see that Query Plan Stats is currently enabled:

```
SELECT * FROM sys.database_scoped_configurations
WHERE name = 'LAST_QUERY_PLAN_STATS'
```

The output is as follows:

```
SELECT * FROM sys.database_scoped_configurations
WHERE name = 'LIGHTWEIGHT_QUERY_PROFILING'
```



configuration_id	name	value	value_for_secondary	is_value_default
22	LIGHTWEIGHT_QUERY_PROFILING	1	NULL	1

Figure 1.22: Check the status of the database scoped configurations

To enable or disable LWP, you will use the database scoped configuration `lightweight_query_profiling`. Refer to the following example:

```
ALTER DATABASE SCOPED CONFIGURATION
SET LIGHTWEIGHT_QUERY_PROFILING = OFF;
```

## Activity monitor

With LWP enabled, you can now look at active expensive queries in the activity monitor. To launch the activity monitor, right-click on the instance name from SSMS and select Activity Monitor. Below Active Expensive Queries, you will see currently running queries, and if you right-click on an active query, you can now examine the Live Execution Plan!

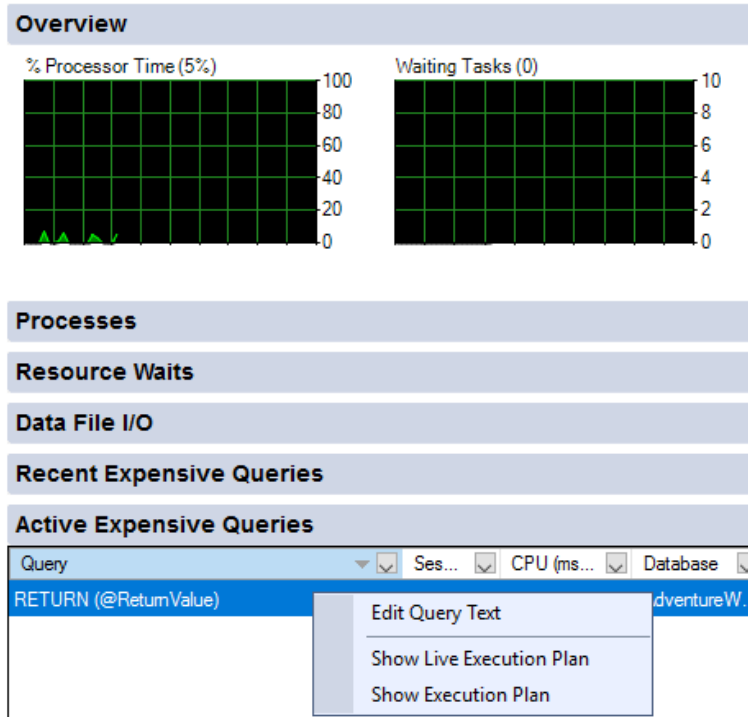


Figure 1.23: The activity monitor

## Columnstore stats in DBCC CLONEDATABASE

**DBCC CLONEDATABASE** creates a clone of the database that contains a copy of the schema and statistics for troubleshooting and diagnostic purposes. More specifically, with **DBCC CLONEDATABASE**, you have a lightweight, minimally invasive way to investigate performance issues related to the query optimizer. In SQL Server 2019, we now extend the capabilities of **DBCC CLONEDATABASE** by adding support for columnstore statistics.

## Columnstore statistics support

In SQL Server 2019, support has been added for columnstore statistics. Before SQL Server 2019, manual steps were required to capture these statistics (refer to the following link). We now automatically capture stats blobs, and therefore, these manual steps are no longer required:

<https://techcommunity.microsoft.com/t5/SQL-Server/Considerations-when-tuning-your-queries-with-columnstore-indexes/ba-p/385294>.

## DBCC CLONEDATABASE validations

DBCC CLONEDATABASE performs the following validation checks. If any of these checks fail, the operation will fail, and a copy of the database will not be provided.

- The source database must be a user database.
- The source database must be online or readable.
- The clone database name must not already exist.
- The command must not be part of a user transaction.

## Understanding DBCC CLONEDATABASE syntax

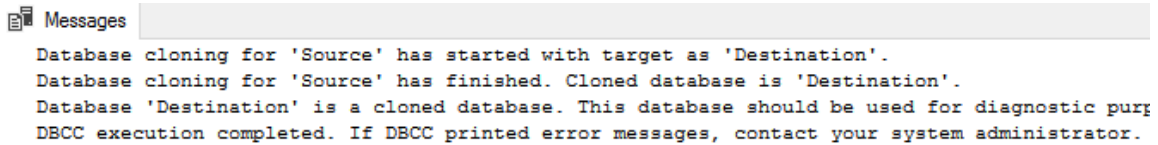
DBCC CLONEDATABASE syntax with optional parameters:

```
DBCC CLONEDATABASE
(
    source_database_name, target_database_name
)
[ WITH { [ NO_STATISTICS ] [ , NO_QUERYSTORE ]
    [ , VERIFY_CLONEDB | SERVICEBROKER ] [ , BACKUP_CLONEDB ] } ]
```

The following T-SQL script will create a clone of the existing database. The statistics and Query Store data are included automatically.

```
DBCC CLONEDATABASE ('Source', 'Destination');
```

The following messages are provided upon completion:



```

Messages
Database cloning for 'Source' has started with target as 'Destination'.
Database cloning for 'Source' has finished. Cloned database is 'Destination'.
Database 'Destination' is a cloned database. This database should be used for diagnostic purp
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
  
```

Figure 1.24: Cloned database output

To exclude statistics, you rewrite the code to include **WITH NO\_STATISTICS**:

```

DBCC CLONEDATABASE ('Source', 'Destination_NoStats')
WITH NO_STATISTICS;
  
```

To exclude statistics and Query Store data, execute the following code:

```

DBCC CLONEDATABASE ('Source', 'Destination_NoStats_NoQueryStore')
WITH NO_STATISTICS, NO_QUERYSTORE;
  
```

### Making the clone database production-ready

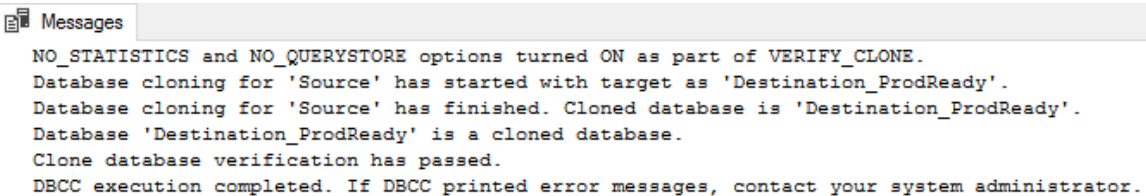
Thus far, the database clones provisioned are purely for diagnostic purposes. The option **VERIFY\_CLONEDB** is required if you want to use the cloned database for production use. **VERIFY\_CLONEDB** will verify the consistency of the new database.

For example:

```

DBCC CLONEDATABASE ('Source', 'Destination_ProdReady')
WITH VERIFY_CLONEDB;
  
```

The output is as follows:



```

Messages
NO_STATISTICS and NO_QUERYSTORE options turned ON as part of VERIFY_CLONE.
Database cloning for 'Source' has started with target as 'Destination_ProdReady'.
Database cloning for 'Source' has finished. Cloned database is 'Destination_ProdReady'.
Database 'Destination_ProdReady' is a cloned database.
Clone database verification has passed.
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
  
```

Figure 1.25: Verifying the cloned database

## Estimate compression for Columnstore Indexes

The stored procedure **sp\_estimate\_data\_compression\_savings** estimates the object size for the requested compression state. Furthermore, you can evaluate potential compression savings for whole tables or parts of tables; we will discuss the available options shortly. Prior to SQL Server 2019, you were unable to use **sp\_estimate\_data\_compression\_savings** for columnstore indexes and, thus, we were unable to estimate compression for columnstore or **columnstore\_archive**.