Post-migration Validation and Optimization Guide

Article • 03/04/2023

Applies to: SQL Server

SQL Server post migration step is very crucial for reconciling any data accuracy and completeness, as well as uncover performance issues with the workload.

Common Performance Scenarios

Below are some of the common performance scenarios encountered after migrating to SQL Server Platform and how to resolve them. These include scenarios that are specific to SQL Server to SQL Server migration (older versions to newer versions), as well as foreign platform (such as Oracle, DB2, MySQL and Sybase) to SQL Server migration.

Query regressions due to change in CE version

Applies to: SQL Server to SQL Server migration.

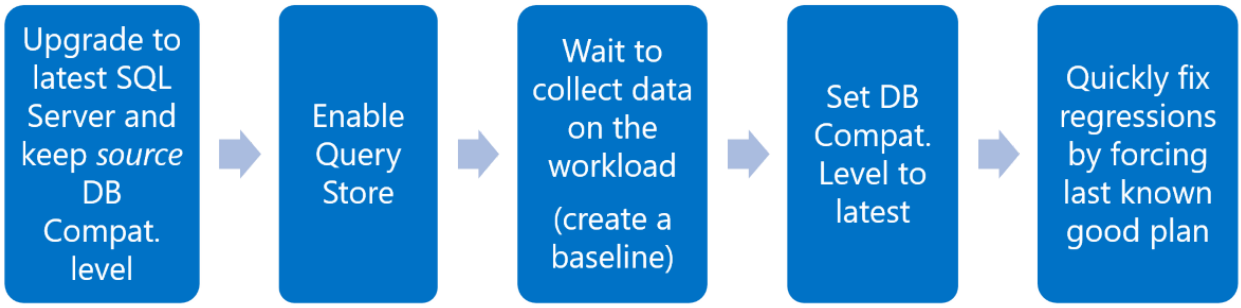
When migrating from an older version of SQL Server to SQL Server 2014 (12.x) or newer, and upgrading the database compatibility level to the latest available, a workload may be exposed to the risk of performance regression.

This is because starting with SQL Server 2014 (12.x), all Query Optimizer changes are tied to the latest database compatibility level, so plans are not changed right at point of upgrade but rather when a user changes the `COMPATIBILITY_LEVEL` database option to the latest one. This capability, in combination with Query Store gives you a great level of control over the query performance in the upgrade process.

For more information on Query Optimizer changes introduced in SQL Server 2014 (12.x), see Optimizing Your Query Plans with the SQL Server 2014 Cardinality Estimator.

Steps to resolve

Change the database compatibility level to the source version, and follow the recommended upgrade workflow as shown in the following picture:



For more information on this topic, see Keep performance stability during the upgrade to newer SQL Server.

Sensitivity to parameter sniffing

Applies to: Foreign platform (such as Oracle, DB2, MySQL and Sybase) to SQL Server migration.

> **Note**
>
> For SQL Server to SQL Server migrations, if this issue existed in the source SQL Server, migrating to a newer version of SQL Server as-is will not address this scenario.

SQL Server compiles query plans on stored procedures by using sniffing the input parameters at the first compile, generating a parameterized and reusable plan, optimized for that input data distribution. Even if not stored procedures, most statements generating trivial plans will be parameterized. After a plan is first cached, any future execution maps to a previously cached plan. A potential problem arises when that first compilation may not have used the most common sets of parameters for the usual workload. For different parameters, the same execution plan becomes inefficient. For more information on this topic, see Parameter Sniffing.

Steps to resolve

1. Use the `RECOMPILE` hint. A plan is calculated every time adapted to each parameter value.
2. Rewrite the stored procedure to use the option `(OPTIMIZE FOR(<input parameter> = <value>))`. Decide which value to use that suits most of the relevant workload, creating and maintaining one plan that becomes efficient for the parameterized value.

3. Rewrite the stored procedure using local variable inside the procedure. Now the optimizer uses the density vector for estimations, resulting in the same plan regardless of the parameter value.
4. Rewrite the stored procedure to use the option `(OPTIMIZE FOR UNKNOWN)`. Same effect as using the local variable technique.
5. Rewrite the query to use the hint `DISABLE_PARAMETER_SNIFFING`. Same effect as using the local variable technique by totally disabling parameter sniffing, unless `OPTION(RECOMPILE)`, `WITH RECOMPILE` or `OPTIMIZE FOR <value>` is used.

> **Tip**
>
> Leverage the Management Studio Plan Analysis feature to quickly identify if this is an issue. More information available **here**.

Missing indexes

Applies to: Foreign platform (such as Oracle, DB2, MySQL and Sybase) and SQL Server to SQL Server migration.

Incorrect or missing indexes causes extra I/O that leads to extra memory and CPU being wasted. This maybe because workload profile has changed such as using different predicates, invalidating existing index design. Evidence of a poor indexing strategy or changes in workload profile include:

- Look for duplicate, redundant, rarely used and completely unused indexes.
- Special care with unused indexes with updates.

Steps to resolve

1. Leverage the graphical execution plan for any Missing Index references.
2. Indexing suggestions generated by Database Engine Tuning Advisor.
3. Leverage the Missing Indexes DMV or through the SQL Server Performance Dashboard .
4. Leverage pre-existing scripts that can use existing DMVs to provide insight into any missing, duplicate, redundant, rarely used and completely unused indexes, but also if any index reference is hinted/hard-coded into existing procedures and functions in your database.

> **Tip**

Examples of such pre-existing scripts include **Index Creation** and **Index Information** .

Inability to use predicates to filter data

Applies to: Foreign platform (such as Oracle, DB2, MySQL and Sybase) and SQL Server to SQL Server migration.

> ⓘ **Note**
>
> For SQL Server to SQL Server migrations, if this issue existed in the source SQL Server, migrating to a newer version of SQL Server as-is will not address this scenario.

SQL Server Query Optimizer can only account for information that is known at compile time. If a workload relies on predicates that can only be known at execution time, then the potential for a poor plan choice increases. For a better-quality plan, predicates must be **SARGable**, or **S**earch **Arg**ument**able**.

Some examples of non-SARGable predicates:

- Implicit data conversions, like VARCHAR to NVARCHAR, or INT to VARCHAR. Look for runtime CONVERT_IMPLICIT warnings in the Actual Execution Plans. Converting from one type to another can also cause a loss of precision.
- Complex undetermined expressions such as `WHERE UnitPrice + 1 < 3.975`, but not `WHERE UnitPrice < 320 * 200 * 32`.
- Expressions using functions, such as `WHERE ABS(ProductID) = 771` or `WHERE UPPER(LastName) = 'Smith'`
- Strings with a leading wildcard character, such as `WHERE LastName LIKE '%Smith'`, but not `WHERE LastName LIKE 'Smith%'`.

Steps to resolve

1. Always declare variables/parameters as the intended target data type.

- This may involve comparing any user-defined code construct that is stored in the database (such as stored procedures, user-defined functions or views) with system tables that hold information on data types used in underlying tables (such as sys.columns).

2. If unable to traverse all code to the previous point, then for the same purpose, change the data type on the table to match any variable/parameter declaration.
3. Reason out the usefulness of the following constructs:

- Functions being used as predicates;
- Wildcard searches;
- Complex expressions based on columnar data - evaluate the need to instead create persisted computed columns, which can be indexed;

> ⓘ **Note**
>
> All of the above can be done programmatically.

Use of Table Valued Functions (Multi-Statement vs Inline)

Applies to: Foreign platform (such as Oracle, DB2, MySQL and Sybase) and SQL Server to SQL Server migration.

> ⓘ **Note**
>
> For SQL Server to SQL Server migrations, if this issue existed in the source SQL Server, migrating to a newer version of SQL Server as-is will not address this scenario.

Table Valued Functions return a table data type that can be an alternative to views. While views are limited to a single `SELECT` statement, user-defined functions can contain additional statements that allow more logic than is possible in views.

> ⓘ **Important**
>
> Since the output table of an MSTVF (Multi-Statement Table Valued Function) is not created at compile time, the SQL Server Query Optimizer relies on heuristics, and not actual statistics, to determine row estimations. Even if indexes are added to the base table(s), this is not going to help. For MSTVFs, SQL Server uses a fixed estimation of 1 for the number of rows expected to be returned by an MSTVF (starting with SQL Server 2014 (12.x) that fixed estimation is 100 rows).

Steps to resolve

1. If the Multi-Statement TVF is single statement only, convert to Inline TVF.

   SQL

   ```
   CREATE FUNCTION dbo.tfnGetRecentAddress(@ID int)
   RETURNS @tblAddress TABLE
   ([Address] VARCHAR(60) NOT NULL)
   AS
   BEGIN
     INSERT INTO @tblAddress ([Address])
     SELECT TOP 1 [AddressLine1]
     FROM [Person].[Address]
     WHERE  AddressID = @ID
     ORDER BY [ModifiedDate] DESC
   RETURN
   END
   ```

   The inline format example is displayed next.

   SQL

   ```
   CREATE FUNCTION dbo.tfnGetRecentAddress_inline(@ID int)
   RETURNS TABLE
   AS
   RETURN (
     SELECT TOP 1 [AddressLine1] AS [Address]
     FROM [Person].[Address]
     WHERE  AddressID = @ID
     ORDER BY [ModifiedDate] DESC
   )
   ```

2. If more complex, consider using intermediate results stored in Memory-Optimized tables or temporary tables.

Additional Reading

Best Practice with the Query Store
Memory-Optimized Tables
User-Defined Functions
Table Variables and Row Estimations - Part 1
Table Variables and Row Estimations - Part 2
Execution Plan Caching and Reuse