Microsoft

Optimization Guide for Mainframe App/Data recompiled to .NET & SQL Server

Optimization Guide for Mainframe App/Data recompiled to .NET & SQL Server

1/3/2019

Version 1.0

Contributors

Chris Godfrey, Jeremy Meng, Alexandra Ciortea, Mukesh Kumar, Jon Jung

Table of Contents

1 Executive Summary

1.1 Scope & Objective

This guide offers optimization advice for executing point-lookups against SQL Server from .NET as efficiently as possible. Customers wishing to migrate from mainframe databases to SQL Server may desire to migrate existing mainframe-optimized design patterns, especially when using 3rd party tools (such as Raincode Compiler) to automatically migrate mainframe code (COBOL/JCL etc) to T-SQL and C# .NET.

Mainframe data accesses commonly employ Point-Lookups (AKA 'Singleton Reads') to loop through and join multiple large tables. This data access pattern is optimal for mainframe, but is suboptimal for querying SQL Server via .NET, where batching is recommended.

1.2 High-Level Recommendations

- Adhere as much as possible to documented .NET and SQL best practices
- Use Memory Optimized Tables with Natively Compiled Stored Procedures
- Use the ODBC driver with SQL Native Client
- Use appropriate data types for optimal query performance
- Configure Windows Power settings for High Performance

1.3 Observations

- High-precision timing must be managed carefully
- At high precision, there are performance costs to accessing DB stats
- External factors outside DB & Application can impact performance

1.4 Next Steps

When planning a data migration from mainframe to SQL Server, carefully consider the observations and recommendations made in this guide.

2 Background

2.1 Scenario

To minimize manual conversion of mainframe database queries, customers may employ third part compilers to translate mainframe code automatically into T-SQL & .NET

Although the guidance is always to follow Microsoft's documentation and adhere to best practice when migration applications, sometimes this is not possible due to costs and other restrictions. Mainframes typically host a large amount of code - one customer had over 4,000 applications. In these cases, a third-party product offering to automate and simplify the code migration becomes very appealing to the customer.

A typical data access pattern for mainframes would be to have multiple large and wide database tables joined and accessed via cursor loops of very fast Point-Lookups (also known as 'Singleton' reads). This is an acceptable design pattern for mainframes due to inherent advantages in their architecture (data is held close to the applications that need it within a monolithic but expensive machine), however once migrated to SQL Server, this becomes an 'anti-pattern' as .NET data accesses to SQL Server perform best with for batch operations.

These cursor loops often join hundreds of tables in highly complex patterns. Although they may be performant on mainframe, such activity would be considered 'chatty' in an SMP system due to the additional costs in network latencies, communication, serialization, and deserialization.

3 Recommendations

3.1 Overview

The below findings represent the combined global effort of multiple Microsoft teams working together over a period of several weeks. During this time, several in-house test environments were created, with many optimization techniques tested and analyzed.

For the sake of brevity, only the optimizations that had a noticeable effect upon overall query time are included. They are offered in descending order of impact.

3.1.1 In-Memory Tables & Natively-Compiled SPs

In-Memory Tables & Natively-Compiled Stored Procedures are usually not recommended for point lookups, however when running many point lookups in a loop, this feature scales to offer improved performance by avoiding any Latch wait types.

Unlike traditional (AKA "Interpreted / On-Disk") Stored Procedures, Named Parameters have a comparatively detrimental effect on performance when used with Natively-Compiled SPs.

Use of ordinal (nameless) parameters when calling Natively-Compiled stored procedures for the most efficient execution;

- To avoid the server having to map parameter names and convert types
- Match the types of the parameters passed to the procedure with the types in the procedure definition

Note - Documentation states "Nameless, also called ordinal, parameters are not supported by the .NET Framework Data Provider for SQL Server. "

However, nameless parameters are supported for other providers, including ODBC, which is recommended for this use case;

https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlparameter(v=vs.110).aspx

Note - Inefficiencies in parameters with natively compiled stored procedures can be detected through the XEvent natively_compiled_proc_slow_parameter_passing:

- Mismatched types: reason=parameter_conversion
- Named parameters: reason=named_parameters
- DEFAULT values: reason=default

https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/best-practices-for-calling-natively-compiled-stored-procedures

3.1.2 Use the ODBC Driver with SQL Native Client

The most efficient way to call a SQL Server stored procedure more than once is through prepared RPC procedure calls. Prepared RPC calls are performed as follows using the ODBC driver in SQL Server Native Client:

- Open a connection to the database
- Bind the parameters using **SQLBindParameter**
- Prepare the procedure call using **SQLPrepare**
- Execute the stored procedure multiple times using **SQLExecute**

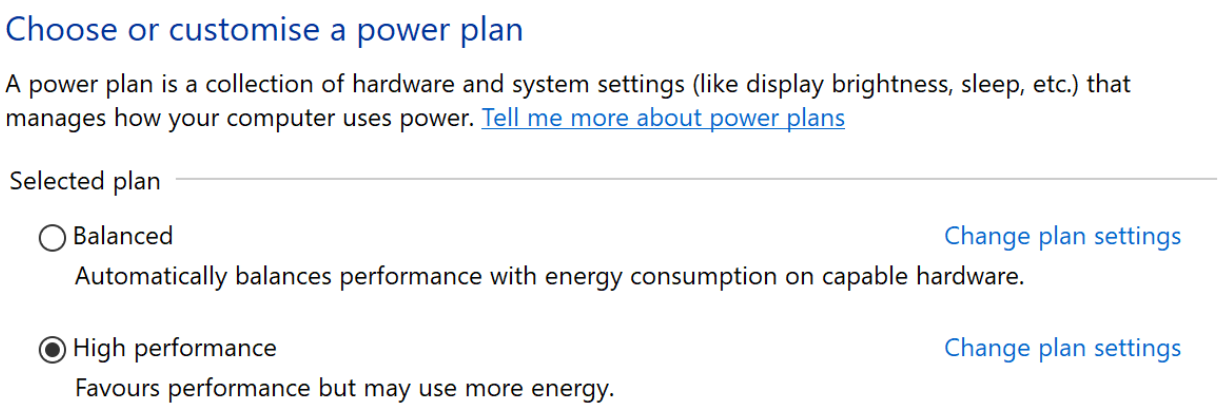
https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/calling-natively-compiled-stored-procedures-from-data-access-applications

3.1.3 Use Appropriate Data Types

As with any database migration, make sure that appropriate data types are configured for the target tables. Implicit conversions and rounding can negatively impact query performance after migration.

3.1.4 Check Windows Power settings

The Power management plan of Windows influences overall system performance, including SQL Server and .NET processes. To ensure performance is not impeded by a low power plan, check that Windows is running in High Performance mode.



4 Observations

4.1 Overview

Accurately measuring performance with microsecond precision is challenging and requires a detailed understanding of the features and metrics available for measuring time and minimizing false-positives or bottlenecks.

4.1.1 Guidance for acquiring Time Stamps

Windows has and will continue to invest in providing a reliable and efficient performance counter.

When you need time stamps with a resolution of 1 microsecond or better and you don't need the time stamps to be synchronized to an external time reference, choose one of the following;

- **QueryPerformanceCounter** (as used by StopWatch)
- **KeQueryPerformanceCounter**
- **KeQueryInterruptTimePrecise**.

When you need UTC-synchronized time stamps with a resolution of 1 microsecond or better, choose **GetSystemTimePreciseAsFileTime** or **KeQuerySystemTimePrecise**.

There are two high-precision (100 nanosecond resolution) clocks available in Windows.
These are Win32 APIs, and .NET code needs to use PInvoke to call them.

- **GetSystemTimePreciseAsFileTime**: 100ns resolution, synchronized to UTC
  Retrieves the current system date and time with the highest possible level of precision (<1us). The retrieved information is in Coordinated Universal Time (UTC) format.
- **QueryPerformanceCounter**: 100ns resolution, not synchronized to UTC
  Retrieves the current value of the performance counter, which is a high resolution (<1us) time stamp that can be used for time-interval measurements.

https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx

4.1.2 SQL DMV Timing Precision

Be aware of SQL DMVs which report in microseconds but are only accurate to milliseconds.

e.g. sys.dm_exec_query_stats

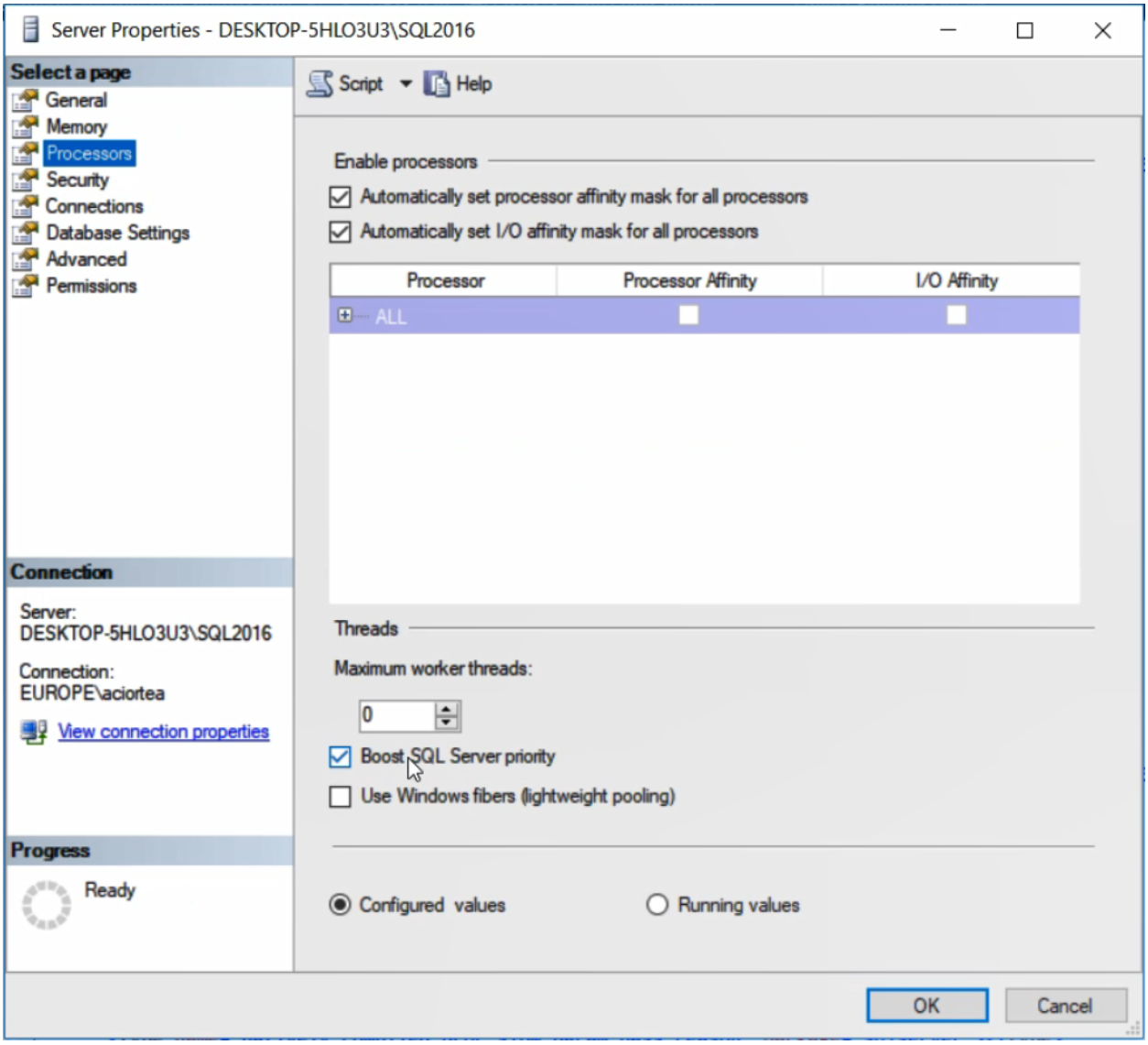
total_worker_time	bigint	Total amount of CPU time, reported in microseconds (but only accurate to milliseconds), that was consumed by executions of this plan since it was compiled. For natively compiled stored procedures, total_worker_time may not be accurate if many executions take less than 1 millisecond.
total_elapsed_time	bigint	Total elapsed time, reported in microseconds (but only accurate to milliseconds), for completed executions of this plan.
last_elapsed_time	bigint	Elapsed time, reported in microseconds (but only accurate to milliseconds), for the most recently completed execution of this plan.
min_elapsed_time	bigint	Minimum elapsed time, reported in microseconds (but only accurate to milliseconds), for any completed execution of this plan.

https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-exec-query-stats-transact-sql

4.1.3 SQL Server Boost Priority

Processes in Windows have different priorities. SQL Server usually has a priority of 7 in Windows. Checking the SQL Boost Priority option will elevate the priority, potentially decreasing the query response time.

This is a deprecated feature and is not usually recommended as it can result in unpredictable behavior on the server, however in certain situations you may want to consider it as a last resort.



https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-priority-boost-server-configuration-option

5 Conclusion

5.1 Next Steps

Further improvements could be reached by altering the data access patterns, or by altering the architecture further.

At the application level, C/C++ may offer a performance benefit above C#.

Experimentation with more powerful CPUs running higher clock speeds is recommended.

Feedback and suggestions

If you have feedback or suggestions for improving this data migration asset, please contact the Data Migration Jumpstart Team (askdmjfordmtools@microsoft.com). Thanks for your support!

Note: For additional information about migrating various source databases to Azure, see the Azure Database Migration Guide.