

SQL Server Technical Article

In-Memory OLTP – Common Workload Patterns and Migration Considerations

**Summary:** In-Memory OLTP is a high performance, memory-optimized engine integrated into SQL Server 2014 and designed for modern hardware trends. In-Memory OLTP allows users to move data into memory resident tables while keeping other data in traditional, disk-based table structures. For performance critical workloads, users can also migrate Transact-SQL code to natively compiled stored procedures. This can provide additional performance gains. This paper aims to help readers understand some of the common architectural patterns where In-Memory OLTP has provided significant benefits. The paper also discusses considerations for migrating applications to In-Memory OLTP.

**Writers:** Mike Weiner, Ami Levin (SolidQ)

**Contributors:** Shep Sheppard

**Technical Reviewers:** Sunil Agarwal, Carl Nolan, David Schwartz, Yorihito Tada, Lindsey Allen, Jos de Bruijn, Jamie Reding, Denzil Ribeiro, Rama Ramani, George Li, Emanual Rivera, Rick Kutschera (BWin.party), Sanjay Mishra, Michael Steineke (Edgenet)

**Published:** **April 2014**

**Updated: July 2014**

**Applies to: SQL Server 2014**

# Content

[Content 2](#_Toc393177597)

[Introduction 2](#_Toc393177598)

[SQL Server In-Memory OLTP Overview 3](#_Toc393177599)

[OLTP Workload Challenges 4](#_Toc393177600)

[High Data Ingestion Rate 5](#_Toc393177601)

[Read/Write Contention 5](#_Toc393177602)

[Low Latency 5](#_Toc393177603)

[Typical Bottlenecks 6](#_Toc393177604)

[Lock, Latch, and Spinlock Contention 6](#_Toc393177605)

[Transaction Log I/O 6](#_Toc393177606)

[Hardware Resources 7](#_Toc393177607)

[Overcoming Challenges Utilizing In-Memory OLTP 7](#_Toc393177608)

[Locking and Latching 7](#_Toc393177609)

[I/O and Transaction Logging 7](#_Toc393177610)

[Latency and Scale 8](#_Toc393177611)

[In-Memory OLTP Common Implementation Scenarios 8](#_Toc393177612)

[High Data Insert Rate 9](#_Toc393177613)

[Read Performance and Scale 12](#_Toc393177614)

[Compute Heavy Data Processing 15](#_Toc393177615)

[Low Latency Execution 17](#_Toc393177616)

[Session State Management 18](#_Toc393177617)

[Migrating applications to In-Memory OLTP 20](#_Toc393177618)

[Assessing Workloads for Migration 21](#_Toc393177619)

[Further Implementation Considerations 26](#_Toc393177620)

[Scenarios Less Suitable for Migration 31](#_Toc393177621)

[Conclusion 32](#_Toc393177622)

[For more information: 32](#_Toc393177623)

# Introduction

The architectural considerations that drove the initial design of relational database management systems in the early 1980s were largely based on the available hardware resources and the business needs of the era. Most of these architectural paradigms are still dominant in today’s modern database engines. However, business needs and available hardware resources have changed dramatically since that time. Therefore, some of these paradigms must be revised in order to keep up with the increasing demands of today.

Organizations frequently encounter extremely challenging scenarios, which require them to significantly scale their applications to process ever increasing data volumes and numbers of concurrent users. This trend is evident across all industries including retail, online banking, online gaming, and so on. At the same time, many modern business workloads require lower latencies at this scale. This is particularly true for scenarios where only computer systems (i.e., not responding to a person) originate and process data. Examples of this “machine-born data” include financial trading, sensor and smart-metering, manufacturing, telemetry, security monitoring/analytics, and many others.

Hardware trends have also changed the characteristics of a typical database server. While there is a continued increase in the number of multi-core processors, their clock speed has not changed significantly in recent years. However, memory is now less of a constraint. Servers can now accommodate much more memory into a commodity class server at a much lower price point. This gives applications the ability to use more main memory instead of incurring disk IO. The past paradigm where data sizes were far larger than available memory is not as applicable. Additionally, the complex logic for paging that had to be implemented in RDBMS to support efficient memory utilization is no longer as relevant.

Even in terms of IO, faster interfaces and the addition of solid-state devices provide greater IO performance and minimize the IO bottleneck. This growth in multi-core systems allows for a larger number of concurrent threads to execute. IO improvements also provide faster response times. Together, these advances push the contention points within the classic relational database architecture to its limits in many cases.

It is these trends, along with the recognition that typical OLTP behavior in these environments is focused on a “hot dataset” ranging from a few GB to a few hundred GB in size which called for a fundamental re-evaluation of the current relational database architecture. In-Memory OLTP is this new architecture designed to alleviate the current database bottlenecks, and take advantage of modern hardware in a dramatically new way while still integrating into the well-known SQL Server relational engine.

# SQL Server In-Memory OLTP Overview

In-Memory OLTP is a specialized, memory-optimized relational data management engine and native stored procedure compiler, integrated into SQL Server. Microsoft designed In-Memory OLTP to handle the most demanding OLTP workloads. To accomplish this, In-Memory OLTP introduces two fundamentally new concepts: memory-optimized tables and natively compiled stored procedures. Data in memory-optimized tables resides in memory, and while transactions are logged for recovery, there is no paging to disk like traditional disk-based tables. Memory-optimized tables provide highly optimized data access structures using hash and nonclustered ordered indexes. The internal structures of these indexes are different from traditional B-trees and provide a new, high performance way of accessing in-memory data. Data access and transaction isolation are handled through a multi-version, concurrency control mechanism that provides an optimistic, non-blocking implementation. While implemented differently from traditional RDBMS, In-Memory OLTP still provides ACID compliance.

The need for critical performance within the engine requires not only a change in the physical storage architecture, but also optimization of the associated data access programming surface. Natively compiled stored procedures offer a highly optimized querying mechanism for accessing in-memory data. Unlike traditional, interpreted Transact-SQL, natively compiled stored procedures are optimized and compiled into machine language when they are created. This allows a much shorter execution code path, which in turn significantly reduces CPU utilization and overall latency.

The In-Memory OLTP engine is fully integrated in SQL Server. This provides an easy way for migration of traditional disk-based tables to memory-optimized tables and interpreted stored procedures to natively compiled stored procedures. These objects reside in the In-Memory OLTP engine along with traditional disk-based tables and stored procedures. The integration allows access to data stored in memory-optimized tables and in disk-based tables using standard Transact-SQL calls (interpreted T-SQL). This integration can help minimize application changes in converting to In-Memory OLTP.

When migrating SQL Server applications to the In-Memory OLTP engine, you can move performance critical data into memory-optimized tables and maintain the rest in disk-based tables. Similarly, you can move the performance critical Transact-SQL code, which interacts with memory-optimized tables, into natively compiled store procedures. Ad-hoc Transact-SQL or non-natively compiled stored procedures can still interact with memory-optimized tables. SQL Server Management Studio, backup/restore, AlwaysOn Failover Cluster Instances, and Availability Groups, among others, are integrated for simplified management.

Companies with one or more of the following conditions should seriously consider the potential benefits of migrating to In-Memory OLTP:

* Existing SQL Server (or other relational database) applications that require performance and scalability gains.
* A RDBMS that is experiencing database bottlenecks – most prevalently around locking/latching or code execution.
* Environments that do not use a relational database in the critical performance path due to the perceived performance overhead.

# OLTP Workload Challenges

The OLTP workload is the target scenario for which Microsoft developed In-Memory OLTP. OLTP workloads are typically characterized by the following attributes:

* Short transactions that require fast response times.
* Queries are constrained to only a few tables and access small datasets.
* Have high concurrency requirements.

These workloads will typically define their performance requirements by:

* Transaction throughput.
* Number of concurrent users.
* Transaction latency (time to complete a particular piece of code execution).

This document, in many cases, will also utilize the term “business transaction” to describe a unit of measure to define application performance. This term refers to a unit of work from an application or line-of-business perspective. It does not refer to something distinct to a tier. Also, it does not refer to a piece of the application, such as a performance counter measuring database transactions.

Some of these workload characteristics can vary based on the specific implementation. However, in all the scenarios in this paper, similar business and application requirements that push performance bottlenecks into the database are exhibited. This is particularly relevant for applications that manage business-critical workloads and require extremely demanding levels of performance.

## High Data Ingestion Rate

A very common application pattern is characterized by the need for the database to ingest data at a very high input rate. There are a wide variety of patterns, where the number of data sources, timeframes in which the data is input (burst or steady stream), and the requirements for the amount of data input fluctuates. However, the common characteristic is the need to handle high fluctuations or a constant rate in the incoming workload, which may overwhelm a traditional RDBMS.

There are also a number of applications that have exhibited “ETL-type” behavior. In this scenario, the data might not just be inserted, but updated, deleted, or transformed as part of the process. In this case, there are two common patterns that relate to the data storage:

* Having an intermediate table that handles the initial load and some form of data movement into a target table.
* Loading directly into the final target table and scaling the concurrency of writes and reads.

The overall performance issue with data ingestion remains the same. To establish baseline measurements for this environment, the metrics are typically characterized by transaction throughput or the number of rows loaded per second.

## Read/Write Contention

Another typical challenging characteristic of many OLTP workloads appears when many processes are targeting small regions of data concurrently. Some processes need to read data while others are modifying it. Although these data sets are typically small, the fact that multiple concurrent user sessions perform high frequency read and write requests on these datasets, causes contention. This contention becomes a significant barrier to scale. Some common patterns that display this bottleneck are:

* Small tables with a very high modification rate.
* Small regions of data within large tables that are accessed frequently. For example, an orders table where new orders are inserted and updated while the same orders are typically the ones most queried.

In many cases, these operations are synchronous with the user interaction, and any delay has an immediate, linear impact on the user’s experience. Overall, the contention adds significant challenges for meeting many performance critical application requirements.

## Low Latency

Many applications have specific duration requirements for the execution of a particular unit of work. They may also have time requirements for the end-to-end execution of specific steps that constitute a business transaction. The applications will measure their performance based on the time it takes to execute some work in isolation or under load. The unit of measurement may be for a specific query execution, business transaction, or process. The measurement consists of the time the client application takes to processes the data to the end consumer and all steps in-between. This measurement of these finer grained executions of work is, in many cases, referred to as latency. Typical latency bottlenecks that can be attributed in some form to database execution would include:

* Communication protocol stack.
* Parsing, optimizing, and compiling a statement.
* Overall time to execute a transaction, which at a low level, is based on the number of CPU instructions per operation.
* Write time to persist a durable transaction to the transaction log.

From a business perspective, these bottlenecks are typically identified as a need for faster execution of a business transaction. In many cases the latency is measured in milliseconds or lower. Improvements in latency may also be expressed as greater (business) transaction throughput, greater scale or concurrency, or more efficient processing of data.

# Typical Bottlenecks

The workload patterns we discussed, in performance critical environments, often display similar bottlenecks in OLTP database applications and architectures. In this section, we will briefly discuss the primary bottlenecks that high performance OLTP SQL Server and database deployments experience.

## Lock, Latch, and Spinlock Contention

OLTP applications can suffer from engine-level concurrency bottlenecks such as incompatible locks, latches, or spinlocks for a given resource. All are scenarios that can cause concurrency bottlenecks that lead to application performance and scale challenges. Locks are a well-established construct of relational databases that use a pessimistic concurrency model. Incompatible types will lead to blocking or waits for particular objects or data. Latches are lightweight synchronization primitives that the SQL Server engine uses to guarantee consistency of page structures supporting index, data pages, and internal B-trees. However, the latches can also inhibit concurrency. Finally, spinlocks are similar to latches in serializing access to data structures. SQL Server uses spinlocks internally to protect concurrent access to shared system data structures.

A particularly challenging scenario is one in which the physical access pattern concentrates in small regions of data such as the recent pages of an index. This index supports an ever increasing key value. The most predominant example is a clustered index on an IDENTITY, SEQUENCE, or other sequential value column. Readers and writers require conflicting locks and latches that are typically focused on recent data. This creates latch and lock blocking delays. Overcoming this contention may prove to be extremely challenging and typically requires significant changes to the application schema or application logic. Some common remediation attempts may include using reverse indexes and non-sequential clustering keys. In some cases, these solutions provide partial relief, but will often introduce new challenges and still impose a major barrier to scale.

In some extreme cases, workloads may instantiate multiple threads which generate substantial modifications to one or more tables concurrently. These may introduce additional contention bottlenecks for metadata operations that involve allocation of pages, extents, and internal maintenance operations such as updating PFS pages.

For a detailed discussion of latch and spinlock contention, see “[Diagnosing and Resolving Latch Contention on SQL Server](http://www.microsoft.com/en-us/download/details.aspx?id=26665)” and “[Diagnosing and Resolving Spinlock Contention on SQL Server](http://www.microsoft.com/en-us/download/details.aspx?id=26666)”.

## Transaction Log I/O

SQL Server stores data and index modifications in the database transaction log to maintain durability. The transaction log records must persist to disk before a transaction can commit. Therefore, OLTP workloads experience very frequent, small log buffer flushes that can cause significant I/O overhead. Moreover, the system writes log records serially and there is only one log writer thread that serves the entire database. This makes the overhead of transaction logging a prime candidate for bottlenecks for high modification rate workloads. Typically, the performance of the underlying subsystem, in terms of IOPS and I/O latency, is the primary consideration when trying to minimize this performance overhead. Slow I/O on the disks that hold the transaction log will make the problem much worse. However, even with high performance disk subsystems, multiple transactions can bottleneck on spinlock contention when writing to the log buffer. For further details regarding the log manager limits, read “[Diagnosing Transaction Log Performance Issues and Limits of the Log Manager](http://blogs.msdn.com/b/sqlcat/archive/2013/09/10/diagnosing-transaction-log-performance-issues-and-limits-of-the-log-manager.aspx)”.

## Hardware Resources

In many cases, hardware components can become the primary barrier to scalability and performance in the system. Some common OLTP bottlenecks involving hardware include:

* CPU utilization that may be a bottleneck under a heavy OLTP workload. Many times applications will utilize scale-out scenarios to spread the workload over multiple servers.
* The I/O subsystem capabilities can often become a bottleneck. As mentioned earlier, transaction log I/O may become a bottleneck in transaction execution. Similarly, throughput, as a measure of IOPS, may contribute to the bottleneck for data/index page allocation and logging. Throughput may also bottleneck SQL Server database checkpoint behavior.

There are many times when bottlenecks occur and a company attempts to overcome them by using “faster” or more efficient hardware. This can be successful in limited cases such as I/O subsystem bottlenecks. However, you cannot resolve many of the typical OLTP bottlenecks, for example, the locking or latching contention, using hardware upgrades.

# **Overcoming** Challenges Utilizing In-Memory OLTP

The In-Memory OLTP technology provides ways to resolve the challenges mentioned earlier.

## Locking and Latching

For the core latching, memory-optimized tables do not use page based structures that require latching. In-Memory OLTP implements an optimistic concurrency control that does not use locks. This provides lock and latch-free data modifications that utilize row versions and timestamps to achieve non-blocking concurrency and address the contention bottlenecks. To overcome the allocation contention bottleneck, In-Memory OLTP uses data files and delta files that utilize append-only write access patterns. SQL Server uses the FILESTREAM infrastructure to manage these files.

## I/O and Transaction Logging

Transaction logging in memory-optimized tables can help to minimize I/O bottlenecks. Because indexes exist entirely in memory, memory-optimized index modifications are not logged in the transaction log. As well, the insert and delete transactions will require less space than disk-based transactions in the transaction log. Finally, log records can be combined up to 24k to minimize the number of log writes. Moreover, the system only writes log records at commit time. This limits the number of times a transaction needs to write to the log buffer. Minimizing this log access reduces contention between transactions that are trying to access the log buffer concurrently.

In-Memory OLTP also provides a few configurable options to help minimize I/O bottlenecks. The most extreme option, when compared to fully logging transactions for recovery, is to create the memory-optimized table with the syntax of DURABILITY = SCHEMA\_ONLY. This option provides for recovery of the schema, but not of the data. Therefore, the system does not need to write to the transaction log for modifications to these tables. Scenarios where data is transient in nature may benefit greatly from this option. Several examples where this option may be utilized are discussed later in the paper.

Another configuration option, configured at a database or transaction level, is delayed durability. This feature is not specific to In-Memory OLTP because you can also implement it as a general configuration in SQL Server 2014. With delayed durability, the system does not persist the log records to disk on commit. Instead, the system flushes the log records to disk after a set time (not on every commit) or when the log buffer fills up. This way, fewer, more efficient, and larger I/Os take place instead of many small flushes per transaction. Moreover, the system can commit transactions before it writes the log records to disk. This minimizes the transaction dependency on the physical I/O.

The application design should take into consideration that the Durability = SCHEMA\_ONLY and delayed durability options do not guarantee recovery of all inflight transactions in the event of sever failure. For more details, see the Books Online section “[Control Transaction Durability](http://msdn.microsoft.com/en-us/library/dn449490(v=sql.120).aspx)”.

## Latency and Scale

In-Memory OLTP can provide significant improvements in CPU utilization. It provides these improvements by utilizing a highly optimized code path to execute queries within the In-Memory OLTP engine as natively compiled stored procedures. Some queries may see additional performance gains because of more efficient point lookups with nonclustered hash indexes compared to disk-based B-Trees. Moreover, migrating tables to In-Memory OLTP removes contention issues and allows fast-access to data via in-memory indexes. In-Memory OLTP also removes the requirement to acquire and release locks. With these features, the application can drive the CPU to much higher overall utilization. This can provide for an increase in throughput and transaction rate where otherwise, CPU cycles are used for thread queue management and require more context switches.

In fact, many of the current workarounds that attempt to address these bottlenecks can be quite challenging. They may also involve application changes or, in some cases, require extensive application redesigns. In many cases, these patterns will benefit the most from a migration to In-Memory OLTP.

# In-Memory OLTP Common Implementation Scenarios

The following section discusses a number of common application implementations that experienced the bottlenecks and scenarios described earlier. All of these scenarios benefitted significantly from implementing In-Memory OLTP.

A summary table below provides the pattern characteristics, challenges, and main benefits of In-Memory OLTP for each scenario. This will help you understand the specific implementations that may match your needs. Please note that some scenarios are not mutually exclusive and some implementation details may be applicable in multiple scenarios.

|  |  |  |
| --- | --- | --- |
| Implementation Scenario | Pattern Characteristics and Challenge | Main Benefits of In-Memory OLTP |
| [High Data Insert Rate](#_High_data_insert) | * Primarily append-only store * Inability to ingest write workload | * Eliminate contention * Minimize I/O logging |
| [Read Performance and Scale](#_Read_Performance_and) | * High performance read operations * Unable to meet scale-up requirements | * Eliminate contention * Efficient data retrieval * Minimize code execution time * CPU efficiency for scale |
| [Compute Heavy Data Processing](#_Compute_Heavy_Data) | * Insert/Update/Delete workload * Heavy computation inside database * Read and write contention | * Eliminate contention * Minimize code execution time * Efficient data processing |
| [Low Latency](#_Low_Latency_execution) | * Require low latency business transactions that typical database solutions cannot achieve * High concurrency exacerbates latency | * Eliminate contention * Minimize code execution time * Efficient data retrieval |
| [Session State Management](#_Session_state_management) | * Heavy insert, update, and point lookups * User scale under load from multiple stateless web servers | * Eliminate contention * Efficient data retrieval * Optional I/O reduction/removal |

For each scenario, a discussion is provided that describes the general business and application challenges, and the bottlenecks. We also discuss the considerations for the architecture or design with In-Memory OLTP that can help improve performance of these workloads. Customer adoption references are provided for further context.

## High Data Insert Rate

One of the critical challenges a number of high performance applications have attempted to address is an extremely high data loading or ingestion rate. Taking in a highly fluctuating stream of data may overwhelm the capabilities of a traditional RDBMS. In this scenario, the primary pattern is inserts. Typically, applications try to input a certain number of rows or business transactions into the system per second to satisfy the required transaction throughput. Data insertion for smart metering and system telemetry are two examples of this scenario.

### Typical Bottleneck

Within the database, the bottlenecks that occur are usually latching and locking within the table on a “very hot” (frequently accessed) dataset region. A large number of threads accessing the same page within a standard B-Tree can cause last-page contention. The addition of indexes for improved query performance may also add an additional overhead to the insert execution. In many cases, there are tradeoffs with optimizing for write throughput and indexing for reads. There may be other potential bottlenecks or performance blockers in this scenario including I/O transaction latency.

### Applying In-Memory OLTP – “Shock Absorber”

In this scenario, “Shock Absorber” refers to the table you create as memory-optimized to “absorb” an input rate. This is the “shock” to the system.

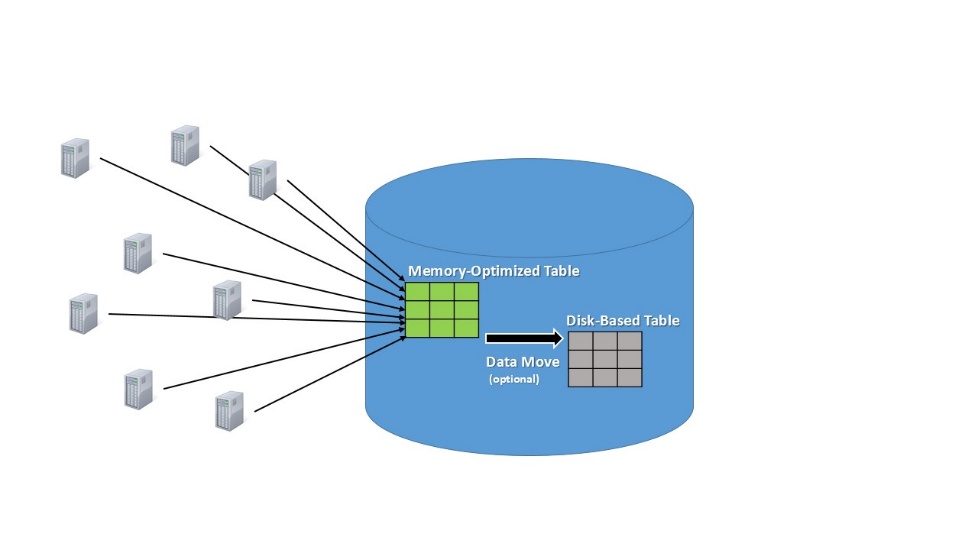
**

Figure - Shock Absorber

Optimizations with regard to memory-optimized tables in this case would include:

* The ability to eliminate lock and latch contention.
* If the workload consists primarily of inserts, the amount of logging required for these inserts is less than for disk-based objects. This is primarily because the system does not log index allocations for memory-optimized tables.
* The ability to eliminate I/Os by creating the memory-optimized tables as DURABILITY = SCHEMA\_ONLY. This would only be the case when the data in the memory-optimized table could be lost (on service failure for example). In the case of such a failure, the data would need to be recreated or reloaded into the table.
* The ability to decrease the transaction time by utilizing delayed durability, with a trade-off of some data loss for potential performance gains.

These optimizations allow the system to achieve greater throughput.

### In-Memory OLTP Design Considerations

In many cases, the implementation of this scenario involves the memory-optimized table being created to store a particular section of the dataset. As an option, you can move the data via Transact-SQL to a destination disk-based table. One of the primary design points to consider would be the intended use and requirements of the dataset that needs to reside in the memory-optimized table.

The memory-optimized table may only store records for a fixed time period (a few minutes, hours, days or more) while disk-based SQL Server tables store the “slightly colder”, less frequently accessed data. Because In-Memory OLTP is integrated into SQL Server, having both memory-optimized tables and disk-based tables in the same database is quite common. Typically, the disk-based table will have the same schema as the memory-optimized table. Inserts (or “appending”) to the table is the primary DML workload.

Moving data between a memory-optimized table and a disk-based table can be accomplished using Interpreted Transact-SQL to select data from the memory-optimized table, insert it into the disk-based table, and (most likely) delete that data from the memory-optimized table. This data movement is usually scheduled as a “background” or SQL Agent process that can execute in a timely fashion.

A typical data movement implementation will usually require a known key or reference value in order to identify the data for movement. Examples that have been successfully implemented typically include:

* A date.
* A sequenced value to attribute to some type of “time stamp”.
* A range of values.

This partitioning of both memory-optimized and disk-based tables is often referred to as “manual partitioning”. There are no specific system functions similar to the table and index partitioning operations for disk-based tables in SQL Server. An example of this is provided in the Books Online section [Application Pattern for Partitioning Memory-Optimized Tables](http://msdn.microsoft.com/en-us/library/dn133171(v=sql.120).aspx).

Other attributes of the application deployment will impact what data is stored in memory-optimized tables and how often you move the data to disk-based tables. Among others, these considerations may include:

* The amount of memory that can be allocated to memory-optimized data. The input data will continue to grow, and there are limits on the amount of memory allocated to SQL Server. Moving data to disk-based tables can free up some allocated memory. In most cases, this will not be immediate because of asynchronous garbage collection. If the application has a need to store a large amount of data, it may not need the entire data-set to be maintained in memory-optimized tables. Be aware that queries on disk-based tables may also require memory to be allocated for pages in the SQL Server buffer pool.
* There are certain desired attributes for the data that memory-optimized tables do not support. This may include data encryption, specialized index types, or other options that memory-optimized tables do not support.
* Consider the types of queries and performance requirements of certain queries which may be best addressed using tables and indexes outside of In-Memory OLTP. For point-lookup queries, hash indexes can be quite efficient and memory-optimized tables provide these. However, the workload pattern may be a mix of OLTP and data warehouse style workloads. These data warehouse style workloads might benefit from solutions like In-Memory Columnstore for Data Warehouse, or from a parallel execution plan. Neither this index nor the parallel plan can be utilized when memory-optimized tables are involved. In these cases, moving this data after a period of time into disk-based tables may prove to be an optimal solution.

As a layer of abstraction between memory-optimized and disk-based tables, a view could also be utilized to address both tables. In this case, interpreted Transact-SQL would be required to access the view.

Another scenario where In-Memory OLTP can be implemented, and requires the same considerations around fast data load, is staging tables, typically for the initial data loading into a data warehouse. While OLTP workloads are the target scenario for In-Memory OLTP optimizations can be realized in this scenario as well. The system fully supports data imports into memory-optimized tables using bcp.exe, Bulk Insert, or SSIS packages. However, TABLOCK hints or other locking behaviors, which may be implemented for fast data loading, are not compatible with In-Memory OLTP and should be removed.

Staging tables are typically used as an intermediate holding area for loading data. Once the data is processed and moved to its final the destination, the data in the staging table is no longer needed and may be discarded. Additionally, the data can be recreated and reloaded if it was lost during the loading process. Therefore, in many cases, the memory-optimized tables can be created with DURABILITY = SCHEMA\_ONLY to avoid all logging and I/O.

Also consider that you typically load and then delete, or truncate staging tables. In the case of a memory-optimized table, you would delete the data because there is no truncate support. Deleting from memory-optimized tables should be quite fast. However, unlike deleting data from disk-based tables, row versions will still utilize memory until the garbage collection process cleans up the rows. Understand that this will temporarily increase the memory requirement. This may have an adverse effect when a large number of rows are deleted. Therefore, an alternative option would be to drop and recreate the staging tables.

Finally, consider the location of the tables involved in the data movement. The In-Memory OLTP engine does not support any transaction that interacts with multiple databases and involves a memory-optimized table. Therefore, data sources for the staging table must be from an external client connection, such as SSIS, or from a table within the database. Similarly, the destination table must be in the same database as the staging tables.

In a number of cases, this configuration has proven successful for ingesting the high volume and “spikes” of the workload. This configuration has also provided a successful overall data-tier design solution to handle these demanding workloads. Customer success and testing with this pattern has also shown, in many cases, that data loading and data movement back to disk-based tables are faster. That is, there is less overall latency than directly inserting and querying from a disk-based table.

### Customer Adoption

An internal Microsoft application, named Event Reporting, was deployed to provide near real-time information, including performance data and event logging, from approximately 7,500 data generating machines. The data streams through a Web API. During high spikes in the input rates, the database cannot ingest all of the data. Utilizing In-Memory OLTP in the “Shock Absorber” scenario, the **Event Reporting application was able to increase its data ingestion rate six-fold. It increased the number of transactions from 3,500 to over 23,000 transactions/sec. This resolved the data ingestion spikes that inhibited the application from achieving the desired throughput.**

## Read Performance and Scale

In several cases, the performance critical portion of the workload focuses on read execution. Many times there will be a piece of the overall dataset that is heavily queried. It may be read-only or have somewhat infrequent updates or changes. Overall, in this pattern, there is a need to service a large number of users. Typically, they access a small set of read-mostly data with a well-defined (typically non-ad-hoc) query workload.

There are a few types of architectural implementation scenarios that can be beneficial for heavy reads. The first approach would be for applications to utilize a middle-tier caching solution to service fast reads. A second approach, in many cases, has been to scale-out read workloads for performance. This approach involves having multiple copies of query critical data. Multiple copies minimize data latency between clients and may be part of a high availability architecture. Typically these solutions are implemented by using SQL Server AlwaysOn Availability Groups with Readable Secondaries or SQL Server Transactional Replication. Scale-out solutions typically work well, but incur high purchase and operational costs.

Scenarios where this workload pattern and architecture are prevalent include large scale retailers, social network browsing, applications that may display leaderboards, or top recommendation engines.

### Typical Bottleneck

There are a few types of bottlenecks that can be challenging in this scenario. Even with all data-pages loaded in memory and optimized indexes, the execution of the Transact-SQL call may still be too slow. In some cases, the overhead of parsing and compiling the query can add extra time to the execution as well. At scale, latching and locking for concurrent read-write scenarios can also delay the query execution.

A hardware bottleneck, such as CPU utilization, may limit scale and require the use of a number of read-only servers to service the user load.

Middle-tier caching solutions can add additional complexity and overhead in managing multiple tiers. Performance overhead for data movement between tiers can also add to the overall time it takes to place data in a cache.

### Applying In-Memory OLTP

In-Memory OLTP memory-optimized tables and natively compiled stored procedures can be implemented as a “cache” solution. Applications that require extremely-fast, read-only access to small subsets of data can use this “cache”. In particular, natively compiled stored procedures can reduce the execution time of targeted queries, which provides a faster overall query execution time. If the workload contains singleton row lookups, using nonclustered hash indexes will provide additional performance gains.

Implementing In-Memory OLTP with existing solutions that now require better read performance will typically require minimal changes to the application or database code. Migrating the data to In-Memory OLTP minimizes the latency associated with buffer cache management and allocation metadata retrieval. This allows for faster data transfers. However, the main benefits in this case, are typically realized by migrating the database code that accesses these data sets to natively compiled stored procedures. This will minimize execution times by eliminating parsing, optimizing, compiling, plan caching, etc. It will not affect some other aspects of the inherent latency such as communication protocols, network overhead, login time, etc.

In-Memory OLTP also provides integrated support with AlwaysOn Availability Groups and memory-optimized tables that serve as transactional replication subscriptions. Both of these scenarios would integrate well into scale-out read solutions. They would provide further scalability and performance gains for the read-replicas/subscribers. In these cases, the integration of In-Memory OLTP can also potentially provide a mechanism for server consolidation through improved CPU efficiency or read-scale. This improvement would allow each read-server to handle more load than it previously could.

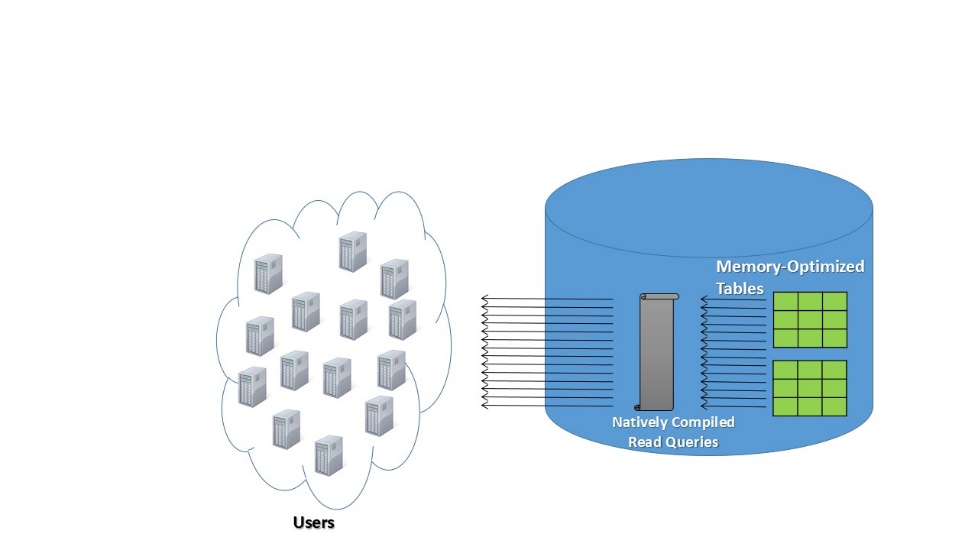
****

Figure - Read cache

### In-Memory OLTP Design Considerations

In some cases, the memory-optimized tables may be the primary table for both reads and writes. If so, the primary gains are the scalability in terms of latching/locking and the potential for performance gains from memory-optimized tables. Because this table holds the only copy of the data, you would typically create the memory-optimized tables as durable (SCHEMA\_AND\_DATA) tables. This provides for read performance similar to a cache, but with durability and “relational” characteristics.

Other cases would typically utilize a middle-tier cache. In this case the data might reside in a backend database server and a subset of the data is pushed at time intervals to update the mid-tier cache. In this scenario, there are a number of considerations for applying In-Memory OLTP. First, in many cases, the memory-optimized table would be configured to mimic or replace the mid-tier cache. The primary source tables may have durable characteristics (and might not even be memory-optimized tables). However, the “cache” table could be created as SCHEMA\_ONLY and, if needed, you can recover the dataset from the source. If updates are infrequent and I/O is not an issue, creating the objects with SCHEMA\_AND\_DATA would allow for an “automated” recovery via the engine on failure.

Some of the more compelling arguments for using In-Memory OLTP instead of another caching system would include:

* In-Memory OLTP provides a familiar development and management experience. All access to data is through Transact-SQL. There is no other programming interface required.
* All the availability and management features are contained in a single source (the database and SQL Server). This would require less overhead when trying to come up with an HA/DR strategy across tiers and applications. In those situations, the implementation and management may be unique to each application.
* Creating a “relational cache” within the database would require less network round trips between mid-tier and server data stores.
* This would minimize the number of servers needed to maintain the environment. Users could potentially consolidate and purchase more memory within a box to service the memory-optimized tables and also utilize them for other resources at the data tier.
* Performance is critically important. Therefore, In-Memory OLTP provides natively compiled stored procedures and optimized indexes, such as the nonclustered hash, for point-lookups. These objects provide very fast access to the critical data-sets.

Queries against memory-optimized tables execute using a single thread, with no support for parallel plan execution. Therefore, targeting queries that can operate efficiently without a parallel plan is critical. In many cases, users may consider “pre-staging” the data by doing calculations. In many cases, users may consider “pre-staging” the data by doing calculations or large aggregations within the disk-based tables as part of the load into the memory-optimized tables. Therefore executing targeted lookups that do not involve as many joins or aggregations on the memory-optimized table. Even if certain queries do execute in parallel, it is possible that conversion to natively compiled stored procedures will reduce the execution time. This reduction may be enough to justify migration to In-Memory OLTP. Not all queries against memory-optimized tables must be natively compiled stored procedures. Targeting only the performance-critical queries for migration to natively compiled stored procedures allows applications to keep executing Interpreted T-SQL against the memory-optimized table.

Memory-optimized tables that store key pieces of data for fast read access will still require a manual update of the data within the cache. The update could be re-occurring and time based using a scheduled job or through a user initiated ad-hoc Transact-SQL execution. This is similar to working with two table objects within SQL Server. You can further optimize operations that only access memory-optimized tables using natively compiled stored procedures.

Some scenarios require a second copy of the table data in whole or pieces. For this, In-Memory OLTP integrates with scale-out architectures such as AlwaysOn Availability Groups with Readable Secondaries and transactional replication read-only subscriptions. AlwaysOn Availability Groups is primarily a high availability and disaster recovery implementation. It does allow up to 8 secondary replicas per Availability Group and provides for reads to be serviced from those replicas. This ability provides an architecture that supports pushing read workloads to the replicas for scale out as well.

The configuration of In-Memory OLTP for objects within an Availability Group is the same as for a disk-based table. Availability Groups use the SQL Server transaction log to synchronize replicas. If data is needed on the secondary replica for reads, the tables must be created as SCHEMA\_AND\_DATA.

The initial configuration of In-Memory OLTP objects and transactional replication requires some specific initial configuration steps. These steps are in the Books Online section “[Replication to Memory-Optimized Table Subscribers](http://msdn.microsoft.com/en-us/library/dn600379(v=sql.120).aspx)”.

### Customer Adoption

Edgenet® is a SaaS provider that develops solutions to provide optimized product data for suppliers, retailers, and search engines including Bing and Google®. Used online and in stores, Edgenet solutions ensure that businesses and consumers can make decisions based on timely, accurate product information. **By implementing In-Memory OLTP, Edgenet was able to remove their dependency on an application cache and consolidate its read workload in the same database to provide near-real-time inventory updates to consumers.** Refer to the “[Edgenet Case Study](http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=710000003026)” for further details.

## Compute Heavy Data Processing

This scenario consists of ingesting data, similar to the *High Data Insert Rate*. However, this scenario requires other attributes including updates, deletions, and additional data processing to quickly make data available to readers. Typically, data is transferred directly to the final destination so that it is to be available for data consumers as quickly as possible. Improving the throughput and scaling concurrent read/write performance are the primary challenges for this pattern.

In many cases, the data is traveling in an Extract-Transform-Load (ETL) or pipeline fashion where data is being delivered to the database. This delivery can occur continually or in a batched manner. The system then processes and transforms the data before writing it to the final target destination. Concurrently, a heavy read workload executes against the target table. Sometimes the query volume may be OLTP-like. The query volume may have point lookups or minimal joins to look up particular change values on one record or a small subset of records. In summary, the distinctive features of this scenario include:

* A workload pattern that involves some compute intensive behavior within the Transact-SQL code.
* The transformation will have more update and delete DML compared to mostly appended inserts.
* Loading directly into the end table and scaling the concurrent writes and reads are important factors.

Some of the business scenarios that may use this solution would include manufacturing supply chains or retailers providing near real-time product catalog information.

### Typical Bottleneck

In this scenario there is the potential for a number of issues with regard to contention. Latching on ingested data is a typical bottleneck. Subsequently, processing and transforming the data within the engine is time consuming. Finally, scaling the concurrent read and write workload creates lock and latch contention as barriers to scale.

### Applying In-Memory OLTP

In this pattern, the final destination table is the memory-optimized table. Usually, this is the replacement for the entire disk-based table and there are no staging or further destination tables within the workload. Moving the performance sensitive tables to memory-optimized tables and Transact-SQL code to natively compiled stored procedures can provide better performance for this workload scenario.

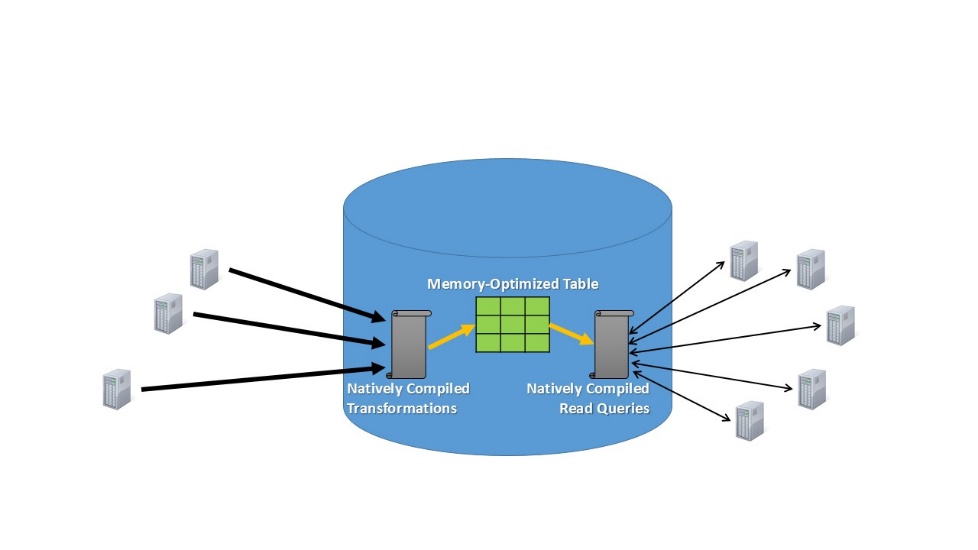
****

Figure - Data transformation pipeline

The implementation of memory-optimized tables and multi-version optimistic concurrency eliminates latching and locking. This alone can provide many workloads with these characteristics a significant performance gain with regard to scale.

Before data is put into the target table, much of the work focuses on transformations and computation over the data. This is an ideal case for natively compiled stored procedures that can significantly reduce the overall execution time of the transformations and improve the overall throughput. Generally speaking, the more work which can be migrated into the natively compiled stored procedures, the more opportunity for performance gains. Placing single statement execution calls into a natively compiled procedure is less likely to produce significant performance improvements.

### In-Memory OLTP Design Considerations

In a number of these cases, the user query workload focuses on one or a few rows. Therefore, optimizing the point-lookup queries with a hash index may prove very beneficial. This is especially true if the size and cardinality of the data are fairly well defined so that an efficient bucket count can be implemented.

It is important to consider the overall size of the memory-optimized table in terms of the target for data loads and queries. There is a limit on the number of checkpoint files that durable tables can have. In many cases, the database size is in the 100’s of GB maximum range, which is not a problem. If the memory-optimized tables are much larger in size, consider moving “aged” or “colder” data to a disk-based table.

When you are considering migrating to natively compiled stored procedures, make sure to properly assess required changes. Also assess your ability to adjust the code to comply with the supported language constructs. Natively compiled stored procedures only support a subset of the Transact-SQL surface area.

Consider setting delayed durability on the critical transactions or at the database level. Doing so may risk the loss of the most recent commits, but it may help reduce the transaction execution time.

### Customer Adoption

Edgenet is a SaaS provider which develops solutions to provide optimized product data for suppliers, retailers, and search engines including Bing and Google. Used online and in stores, Edgenet solutions ensure that businesses and consumers can make decisions based on timely, accurate product information. **By implementing In-Memory OLTP Edgenet was able to gain between 8x and 11x transaction throughput into SQL Server, dramatically reducing their time to get updated data into the database.** Refer to the “[Edgenet Case Study](http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=710000003026)” for further details.

## Low Latency Execution

The primary focus of many performance sensitive applications is end-to-end latency. In very low latency scenarios, you measure and analyze every component in the critical execution path for performance gains. Here the impact of saving fractions of a second are critical to the business.

In many application architectures concerned with latency, relational databases may not exist in the critical execution path. A system may write to a relational database in an asynchronous manner because its impact on latency is too significant.

Industries or solutions that typically have low latency requirements include capital markets and investment trading, high scale online gaming platforms, etc.

### Typical Bottleneck

Latency is a key performance indicator for all the layers that transactions go through. All of the scenarios mentioned earlier featured some sort of latency as a result of a bottleneck, such as lock and latch contention. Migration to In-Memory OLTP is a proven method that provides benefits in terms of overall latency.

You can view latency in two ways. First, there is the time it takes to execute a certain unit of work, in isolation, without any other work running on the system. Often, the execution of some unit of work, even in isolation, will not happen fast enough to meet the needs of the application. As mentioned earlier in the [OLTP workload challenges](#_Low_latency) section, this latency can derive from:

* The network and communication protocol stack.
* Parsing, optimizing, and compiling a statement.
* The overall time to execute a transaction, which, at a low level, is based on the number of CPU instructions required to perform each operation.
* The write time required to execute a durable transaction to the transaction log.

Second, there is the behavior of the system under load. Here scenarios such as latching and locking or resource utilization, such as CPU, may increase the execution time of particular units of work. All this adds latency to the process.

### Applying In-Memory OLTP

In-Memory OLTP can dramatically reduce overall latency in the following ways:

* Minimize the code execution path and provide CPU efficiencies in code execution. It eliminates the parsing, optimizing, and compiling of a query at execution by using natively compiled stored procedures.
* Improves the performance of point lookup queries by utilizing nonclustered hash indexes.
* Improves I/O performance for logging as demonstrated in the previous scenarios.
* Provides for non-latching and locking data access under load. This removes barriers to scale and execution time for high concurrency workloads.

### In-Memory OLTP Design Considerations

When considering In-Memory OLTP for latency optimization, remember that you reap most of the benefits using natively compiled stored procedures and in-memory objects. Therefore, migrating as much of the execution path as possible into these engine components will have a decisive impact on the overall result. You have different I/O performance enhancement options that may also impact durability in a standard SCHEMA\_AND\_DATA table. The I/O latency of ensuring transaction durability is still part of the synchronous transaction execution path.

Most OLTP applications issue large amounts of very small queries that add an additional layer of latency overhead from the client side connection stack. This stack establishes the connection to the database engine, retrieves results, handles errors, etc. In-Memory OLTP does not provide optimizations in the networking layer. There are no optimizations for low level client server communication, such as TDS, or higher level, such as client network libraries. The overhead of this network communication can sometimes be significant. A typical workaround may involve consolidation, or “batching”, multiple transactions into a single execution unit. The unit executes in a single “trip” to and from the database engine. Also, consider using the SQL Server native client and prepared execution to minimize the latency of individual calls. Calls from the client and server may not require synchronization on a single thread. If so, open up another client thread to gain performance using scale. In both cases, application changes are required in order to further minimize this network overhead.

It is important to remember that In-Memory OLTP uses an optimistic concurrency control mechanism. This mechanism can suffer from write conflicts when two or more transactions attempt to modify the same row. When a write conflict occurs, one of the participating transactions will be rolled back (at the time of conflict and before the commit). This rollback will incur an overhead and add latency to the business transaction. In a similar manner, validation of transactions running under certain isolation levels, repeatable read and serializable, will also occur before the commit and statements can be rolled back. The failed transaction will not be automatically re-submitted to the engine. Instead, the system throws an error that the application will need to process further. We recommend that you write retry logic to deal with these conflicts. However, minimizing the potential conflicts from the start is the ideal solution to reduce the number of retries. In terms of latency, implementing the retry logic in the client will help push all of the work executing in SQL Server into the In-Memory OLTP engine. This is similar to how you would deal with deadlock detection. Using the prepared execution model of the SQL Server Native Client will minimize the overhead associated with repeated calls to the slower interpreted Transact-SQL. For a detailed discussion, please read “[Prepared Execution](http://msdn.microsoft.com/en-us/library/ms131667(v=sql.120).aspx)”.

### Customer Adoption

SBI Liquidity Market provides services for online foreign currency exchange trading. ***By i*mplementing In-Memory OLTP, SBI was able to increase throughput and scale and develop its platform to minimize latency.** For more information, see the “[SBILM Case Study](http://www.microsoft.com/casestudies/Microsoft-SQL-Server-2014/SBI-Liquidity-Market/Leading-Japanese-Financial-Firm-Accelerates-Trading-Platform-with-In-Memory-OLTP/710000003429)”.

## Session State Management

Unlike the previously discussed broader patterns, the need to maintain session information is a specific function for many applications. Session state management involves maintaining states for stateless protocols, the most prominent example being HTTP. When users interact with a website, their preferences, actions, and internal metadata need to persist across multiple HTTP requests. This persistence is often implemented in a database. When using load balanced web servers, where many servers may service the same session, centralizing the state storage can result in increased contention. This workload behavior will result in a significant amount of updates to users’ session data and usually utilizes lookup queries focused on a single row. Some web platforms, such as ASP.Net, provide the option to maintain session state in-process or use a SQL Server relational database to persist this session state data. The latter is typically used for high scale websites.

The overall size of the session state implementations is usually quite small. However, the data is very dynamic in nature. Some threads are actively writing and modifying data while other threads read the previously saved data. As the number of threads that try to access this relatively small region of data increases, lock and latch contention bottlenecks begin to manifest. For heavily loaded systems utilizing a large number of CPU cores, this can be a significant barrier to scale. The time it takes to update and query these tables is critical to the user experience.

### Typical Bottleneck

These patterns typically begin to suffer from the concurrent read/write contention challenge that introduces latch and lock bottlenecks. In many cases, this scale bottleneck would cause application slow-downs or force the implementation of a logical scale-out. Both scenarios could be quite impactful to the business.

### Applying In-Memory OLTP

Memory-optimized tables do not suffer from latching because the core In-Memory OLTP engine uses lock-free algorithms and structures that introduce no latching. Migrating the tables with latch hotspots to memory-optimized tables eliminates this scale bottleneck. Of note, the optimistic concurrency model provides a new paradigm that applications will have to consider. Typically, session state tables do not suffer from conflicts between writers because each web session will only update its own rows. However, similar patterns may have concurrent transactions that modify the same row, which can cause conflicts without locks. If the conflict occurs, one of the transactions will fail. Preparing for conflicts, by using try/catch logic in Transact-SQL or application retry code for example, is critical.

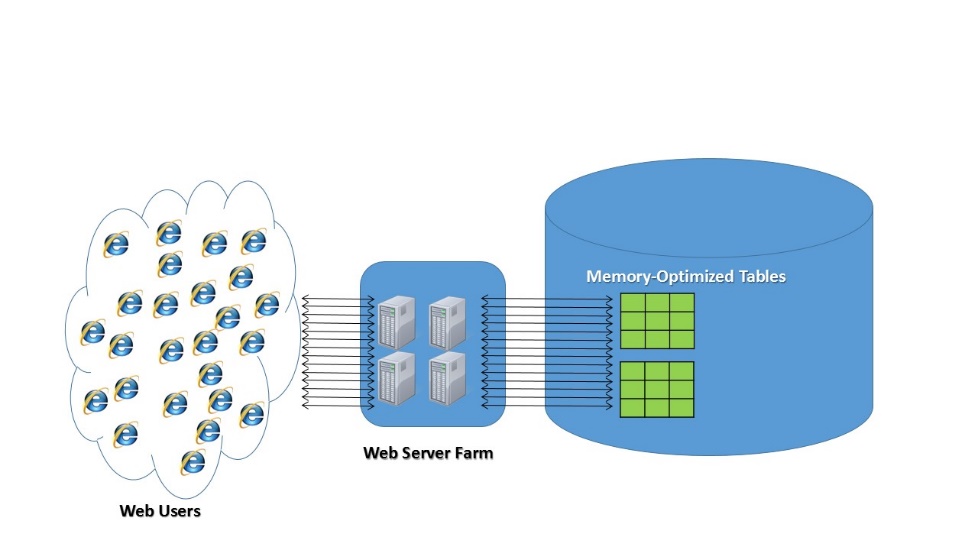
**

Figure - Session state

### In-Memory OLTP Design Considerations

Evaluate the durability requirements for the data. The In-Memory OLTP engine supports full ACID compliance and table durability by logging content into the database transaction log. It also provides the table level configuration for creating tables as SCHEMA\_ONLY or non-durable. In session state management applications, the impact of losing this data is often that a user needs to log back in to the web server. Therefore, in some cases, SCHEMA\_ONLY tables can be implemented. In other scenarios where the data was not considered transient or the overall reduction of I/O was not as impactful declaring the tables as SCHEMA\_AND\_DATA was chosen for durability.

In some cases, the state management tables contain LOB data or have a row size larger than 8060 bytes. Memory-optimized tables do not support LOB columns and only support row sizes smaller than 8060 bytes. Application design patterns for this would include splitting the table and moving the LOB data or larger data type columns into a disk-based table. This approach cannot provide much benefit if the bottlenecks is in accessing the LOB data. Therefore, if you frequently access the LOBs and they are part of the bottleneck, consider the overall benefit carefully. Alternatively, splitting up the large columns (i.e. if the datatype is varbinary) into multiple rows and “re-assembling” them for retrieval using natively compiled stored procedures has proven to be an efficient solution. Finally, in other scenarios, applications have chosen to limit the data input to fewer than 8060 bytes. For further information on these scenarios, see “[Implementing LOB Columns in a Memory-Optimized Table](http://technet.microsoft.com/en-us/library/dn296676(v=sql.120).aspx)”.

Hash indexes are highly optimized for point-lookups. As almost all the data modifications and queries are on single rows, implementing the memory-optimized nonclustered hash index can help improve the performance of queries.

The latency of the standard interpreted Transact-SQL calls can also impact the overall end-user performance. Reducing the time for each business transaction can be an important goal in terms of overall performance. Migrating the Transact-SQL code into natively compiled stored procedures and reducing the latency of transaction executions are critical factors in improving the overall user experience.

### ASP.NET Session State Provider for In-Memory OLTP

An ASP.NET session state provider, along with Transact-SQL scripts, have been developed to make it easier to take advantage of SQL Server In-Memory OLTP when using ASP.NET session state.

The binaries and code have been published in the following locations:

1. The package can be obtained from: <https://www.nuget.org/packages/Microsoft.Web.SessionState.SqlInMemory/>

2. The source code is also accessible through Codeplex: <https://msftdbprodsamples.codeplex.com/releases/view/125282>

### Customer Adoption

Utilizing In-Memory OLTP for state management has been a popular scenario that several architectures have utilized.

For BWin.party, this session state implementation was one of the most mission critical pieces of its business. **By migrating to In-Memory OLTP, batch requests increased from a limit of 15,000 to 250,000 for a gain of 16X in throughput. This allowed BWin.party to scale-up its environment and consolidate from 18 logically partitioned server instances of SQL Server to one. To accomplish this, BWin.party did not modify any code outside of SQL Server.** For further details, read the “[Bwin.party Case Study](http://www.microsoft.com/casestudies/Microsoft-SQL-Server-2014/bwin.party/Gaming-Site-Can-Scale-to-250-000-Requests-Per-Second-and-Improve-Player-Experience/710000003117)”.

# Migrating applications to In-Memory OLTP

In-Memory OLTP, while integrated into SQL Server, has some unique functionality that differs from traditional relational-database systems and SQL Server specifically. The goal of this section is to provide the reader with a number of these considerations. This section does not provide comprehensive guidance on migration and many in-depth discussions about specific requirements, supported features, and surface area are out-of-scope for this paper.

## Assessing Workloads for Migration

As discussed previously, many applications have experienced significant benefits from implementing In-Memory OLTP. However, different workloads may gain different degrees of improvement. Some applications may require minimal changes while others may need more extensive code modification, for example to convert certain Transact-SQL to natively compiled stored procedures. There are also some scenarios that are not suitable for In-Memory OLTP. It is therefore critical to understand the characteristics of the workload and the environment. This assessment will help you determine if an application is a viable candidate for In-Memory OLTP. The following sections will suggest an approach for reviewing application workloads and provide observations about the viability of these implementations for In-Memory OLTP.

### Migration Methodology

Before migrating to In-Memory OLTP ensure that you understand the requirements or goals of the application thoroughly. Additionally, determine if the performance bottleneck is something that In-Memory OLTP can address. As the diagram below suggests, there are certain aspects of the SQL Server engine for which In-Memory OLTP can provide gains. Other components that interact with the engine are unlikely to gain from a migration to In-Memory OLTP.

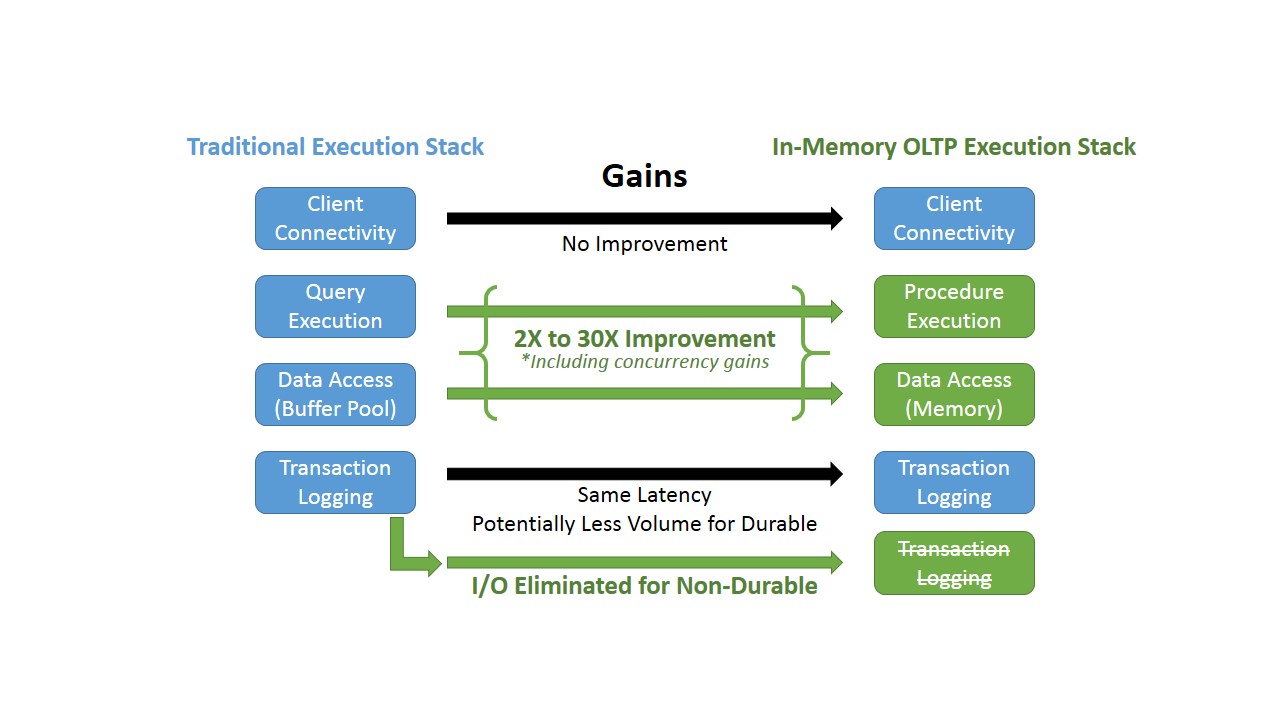


Figure – Performance gain areas for In-Memory OLTP

As shown in Figure 6, In-Memory OLTP addresses the engine components of the data access layer (tables and index objects) and query execution. If the current bottleneck is in this space, moving those data-sets or Transact-SQL into the In-Memory OLTP engine can enhance performance. In-Memory OLTP may also produce less log volume than SQL Server transactions compared to disk-based tables because no records are written for index allocations or UNDO requirements. In any case, the log latency on I/O to the disk subsystem is still a part of the transaction commit. Features such as SCHEMA\_ONLY tables (non-durable) and delayed durability can eliminate or minimize this overhead in unique situations. Additionally, no enhancements are made to the client connectivity layer. There are ways of developing to cope with this, but In-Memory OLTP does not directly address or resolve this bottleneck.

#### Establishing a baseline

Identifying a baseline is critical to understanding the current performance of a system and measuring the improvement or degradation when you make changes. You can accomplish this in a variety of ways. Many of the tools and methodologies are discussed in the “[Monitor and Tune for Performance](http://msdn.microsoft.com/en-us/library/ms189081(v=sql.120).aspx)” article. While many tools exist to help determine a baseline, choosing a measurement that serves as the baseline is important. Some examples of a baseline measurement are:

* System components such as: disk, CPU, memory, network, for utilization and performance.
* SQL Server code execution time.
* Transaction throughput.
* What the system is waiting on and for how long. Here Dynamic Management View such as sys.dm\_os\_wait\_stats can be helpful in identifying SQL Server resource waits.

The overall application performance should be considered as part of a baseline as well. When determining a baseline also consider the following factors:

* Measurements for business transaction throughput.
* Transaction round trip times.
* User experience.
* Scale.

Using these measurements can help define the criteria for a successful migration.

#### Bottleneck Analysis

In some cases, factors outside the database engine may be causing the bottleneck within the application. Therefore, migration to In-Memory OLTP may not improve the situation. Understanding the current performance bottlenecks in the overall application architecture is critical. Once you identify the bottleneck in the database, focus on these specific components for migration.

##### Using In-Memory OLTP Tools to Analyze Workloads and Help with Migration

The In-Memory OLTP offering provides tools to help with the migration process. The tools are integrated into Management Studio. We sometimes refer to the toolset as the Analyze, Migrate, and Reporting (AMR) toolset. To help with the bottleneck analysis, new data collectors (Transaction Performance Collection Sets) are available. They help collect performance data and workload characteristics. They also recommend heavily used or contentious tables and code for migration to In-Memory OLTP.

The data collectors do significantly more than run a single query to determine a good candidate for In-Memory OLTP migration. The collectors use Management Data Warehouse (MDW) to execute aggregations of the data over the time period in which they are collected. Collectors provide estimates regarding performance gains from migration to In-Memory OLTP. This information appears in the “Transaction Performance Analysis” reports that come with SQL Server 2014 Management Studio. For a detailed discussion on configuring and utilizing the tools, please read the “[Migrating to In-Memory OLTP](http://msdn.microsoft.com/en-us/library/dn247639(v=sql.120).aspx)” page which has drill-down sections on this functionality.

If you cannot utilize the data collection and reporting tools, there are other ways to help understand the contention points within SQL Server. [SQLDiag](http://technet.microsoft.com/en-us/library/ms162833.aspx), [PSSDiag](http://diagmanager.codeplex.com/), and [SQL Nexus](http://sqlnexus.codeplex.com/) are tools that you can use to determine contention issues related to latching, locking, spinlocks and stored procedure execution statistics. SQL Nexus will identify wait resources, latching, locking, and blocking. If a SQL Trace is run as part of the capture, SQL Nexus also captures top stored procedure and Transact-SQL Statements.

If the overhead for monitoring and collecting this information over a time period is too costly you have another option. You can execute manual queries to detect frequently accessed tables, latch contention, or stored procedure execution statistics. The execution of these queries provides a snapshot of the values that exist since the last server restart and in the case of queries, only of what is still in memory. In some cases, it may be more accurate to capture the output at two points in time and determine the delta between them. The queries to collect this information are in the sections that follow.

##### Frequently Accessed Tables

This query provides a list of the most frequently accessed tables in the database since the SQL Server instance was reset or the performance data was cleared. When determining candidates for conversion, consider top lookup counts and scan counts. They give you an idea of usage by evaluating singleton\_lookup\_count and range\_scan\_count from the following query. For migration to In-Memory OLTP, consider moving the heavily accessed tables or parts of the data within the tables into memory-optimized objects.

SELECT TOP (5) b.name AS TableName,

a.database\_id,

a.singleton\_lookup\_count,

a.range\_scan\_count

FROM sys.dm\_db\_index\_operational\_stats(DB\_ID(), NULL, NULL, NULL) AS a

INNER JOIN sys.objects b on a.object\_id = b.object\_id

WHERE b.type <> 'S'

AND

(a.singleton\_lookup\_count > 0 OR a.range\_scan\_count > 0)

ORDER BY a.singleton\_lookup\_count DESC

GO

##### Latch Contention

Evaluate latch contention by reviewing the page\_latch\_wait\_and page\_lock\_wait from the sys.dm\_db\_index\_operational\_stats DMV. Consider migrating tables that have a high latch or lock contention to In-Memory OLTP. The query below will provide the cumulative latch and lock waits since the SQL instance was reset or the performance data was cleared.

SELECT TOP (5) a.database\_id,

so.object\_id,

so.name AS TableName,

a.page\_latch\_wait\_count ,

a.page\_latch\_wait\_in\_ms,

a.page\_lock\_wait\_count,

a.page\_lock\_wait\_in\_ms

FROM sys.dm\_db\_index\_operational\_stats(DB\_ID(), NULL, NULL, NULL) a

INNER JOIN sys.objects AS so

ON a.object\_id = so.object\_id

WHERE so.type = 'U' AND a.page\_io\_latch\_wait\_count > 0

ORDER BY a.page\_latch\_wait\_count DESC;

##### Stored Procedure Execution Statistics

In-Memory OLTP offers natively complied stored procedures. They vastly improve stored procedure execution times by compiling Transact-SQL natively into a DLL and minimizing the instructions which need to be processed at execution. If you cannot use the tools in Management Studio, the two queries below will provide similar information. However, these queries can only retrieve stored procedure and Transact-SQL data that exists in the plan cache at the execution of this call. It is possible that stored procedures may exist that may be better candidates, but they are no longer in the plan cache. Consider running these queries several times over the course of a day or week to determine if other candidates exist.

The following query will provide the stored procedure name with the highest total worker time. If the performance data has been recently cleared, the plan cache will be volatile or if the SQL server instance has been reset the plan cache may not have enough information to be valuable at that time.

SELECT TOP (10) sp.database\_id,

so.name AS StoredProcName,

sp.total\_worker\_time,

sp.execution\_count,

sp.total\_logical\_reads,

sp.total\_logical\_writes,

sp.total\_logical\_reads

FROM sys.dm\_exec\_procedure\_stats AS sp

INNER JOIN sys.objects AS so

ON (sp.object\_id = so.object\_id)

WHERE so.type = 'P' AND sp.database\_id = DB\_ID()

ORDER BY sp.total\_worker\_time DESC;

The data retrieved from the following query is not available in the Management Studio tools. However, it may help determine which statements within the stored procedure are the top consumers of resources. When deciding which stored procedures to migrate to In-Memory OLTP, knowing which statements are top consumers can be valuable. It may beneficial to only move a few high resource consuming statements into a natively complied stored procedure instead of moving the entire stored procedure.

SELECT TOP (50) sp.database\_id,

dbname= DB\_NAME (qt.dbid),

so.name AS StoredProcName,

sp.total\_worker\_time,

sp.execution\_count AS StoredProcedureExecCount,

qs.execution\_count AS StatementExecCount,

SUBSTRING(qt.text,qs.statement\_start\_offset / 2 + 1,

(CASE WHEN qs.statement\_end\_offset = -1

THEN LEN(CONVERT(NVARCHAR(MAX), qt.text)) \* 2

ELSE qs.statement\_end\_offset END - qs.statement\_start\_offset) / 2

) AS query\_text

FROM sys.dm\_exec\_query\_stats AS qs

CROSS APPLY

sys.dm\_exec\_sql\_text(qs.sql\_handle) AS qt

INNER JOIN sys.dm\_exec\_procedure\_stats AS sp

ON (sp.sql\_handle = qs.sql\_handle)

INNER JOIN sys.objects so

ON (sp.object\_id = so.object\_id) and sp.database\_id = DB\_ID()

ORDER BY sp.total\_worker\_time DESC, qs.execution\_count DESC

#### Conducting Tests Using a Representative Workload at Scale

After defining the goal and the bottleneck, it is important to consider a testing criteria to simulate the bottleneck. The test will also determine if you can achieve the goal. There may be some scenarios where a “unit test” or single execution of a particular part of the workload will help confirm the goal. For example, the concern may be the latency of a stored procedure call across a single thread in isolation. However, in many cases, you expose the bottleneck or performance degradation only under load, especially with regard to contention at scale.

If you only observe the bottleneck under a scale scenario, testing executed with and without In-Memory OLTP may display similar performance before you produce the contention. Once you reach the scale to simulate the contention, from that point on, you will observe significant improvements with In-Memory OLTP. In many cases, in the traditional database systems, throughput will reach a plateau or execution time will increase. At that point, additional load or scale on the system is likely to have a flatline, or negative scale, with respect to performance. The alleviation of this bottleneck in In-Memory OLTP can help sustain and improve the overall application performance well past this observed bottleneck.

Figure 7 was created utilizing an [AdventureWorks sample database](http://msdn.microsoft.com/en-us/library/dn511655(v=sql.120).aspx) that was converted to In-Memory OLTP. The image shows the engine performance gains as a workload runs into contention at scale. You can only recognize these benefits by simulating a test to the point where scale affects performance in a typical RDBMS. The circle in the diagram shows where typical RDBMS execution takes significantly more time to execute the same number of transactions. After this point, the In-Memory OLTP workload continues to scale almost linearly while the disk-based workload reaches a scale barrier.

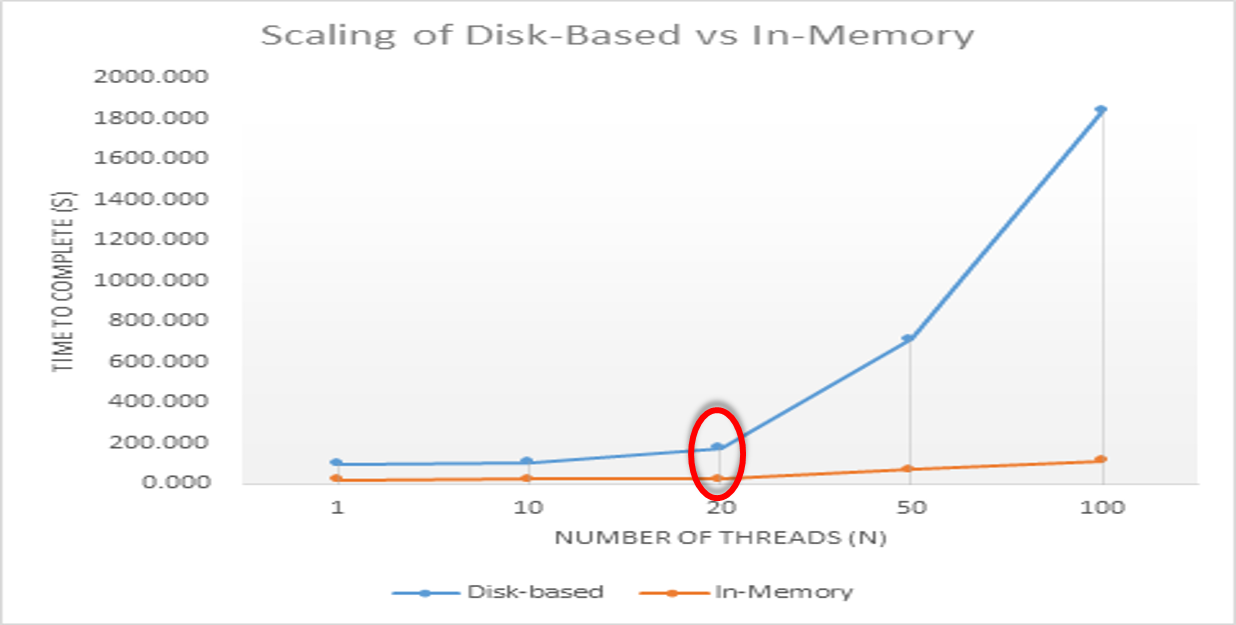


Figure – Time to complete/number of threads from AdventureWorks sample executing DemoInsertSalesOrders.

Finally, consider that, as you migrate the workload to In-Memory OLTP, it is possible some characteristics of the workload may change. The ideal scenario is to measure the performance of the overall application based on a stable, defined measurement of work. Figure 7, displays the ideal based on “N”, as in the number of user threads (measured against time). Measurements, such as business transactions or the number of concurrent users, are easier to correlate across tests compared to a specific value whose scope may change between executions.

#### Targeted, Iterative Migration Approach

We recommend that you approach migration by focusing on specific areas within the workload that exhibit bottlenecks that In-Memory OLTP can address. Often, you may only need to migrate a subset of the data and objects to In-Memory OLTP to achieve significant gains.

Some migrations of critical tables to memory-optimized structures and Transact-SQL code to natively compiled code are quite simple and require minimal changes. Other cases may require complex changes because of surface area support or the ability to deploy these new database objects.

Two tools, integrated into SQL Server 2014 Management Studio, to help migrate objects are the Memory Optimization Advisor and Native Compilation Advisor. Both tools help you to assess the migration’s difficulty. They evaluate table schema and stored procedure syntax to find possible migration blockers, e.g., an unsupported data type. The tools can then provide information on ways to resolve these migration blockers. If Memory Optimization Advisor detects no migration issues, its wizard can help create the memory-optimized table and help with data migration.

Because In-Memory OLTP is integrated into SQL Server, it allows you to utilize memory-optimized tables and disk-based tables together. In-Memory OLTP also allows Interpreted-Transact-SQL and natively compiled stored procedures to access data stored in memory-optimized tables. If you must migrate specific components over time, use the following incremental migration strategy:

* Identify the contentious, bottlenecked tables that limit scalability.
* Address unsupported features, and migrate the critical data to memory-optimized tables.
* Perform the minimal code changes required to access the memory-optimized tables using interpreted Transact-SQL.

These steps should alleviate most scalability issues associated with locking and latching. If you need additional performance gains, especially with regard to Transact-SQL execution time, then implement migration of Transact-SQL to natively compiled stored procedures.

* Identify the performance critical Transact-SQL that accesses these tables and their dependent objects.
* Address unsupported language constructs and migrate these stored procedures to natively compiled code.

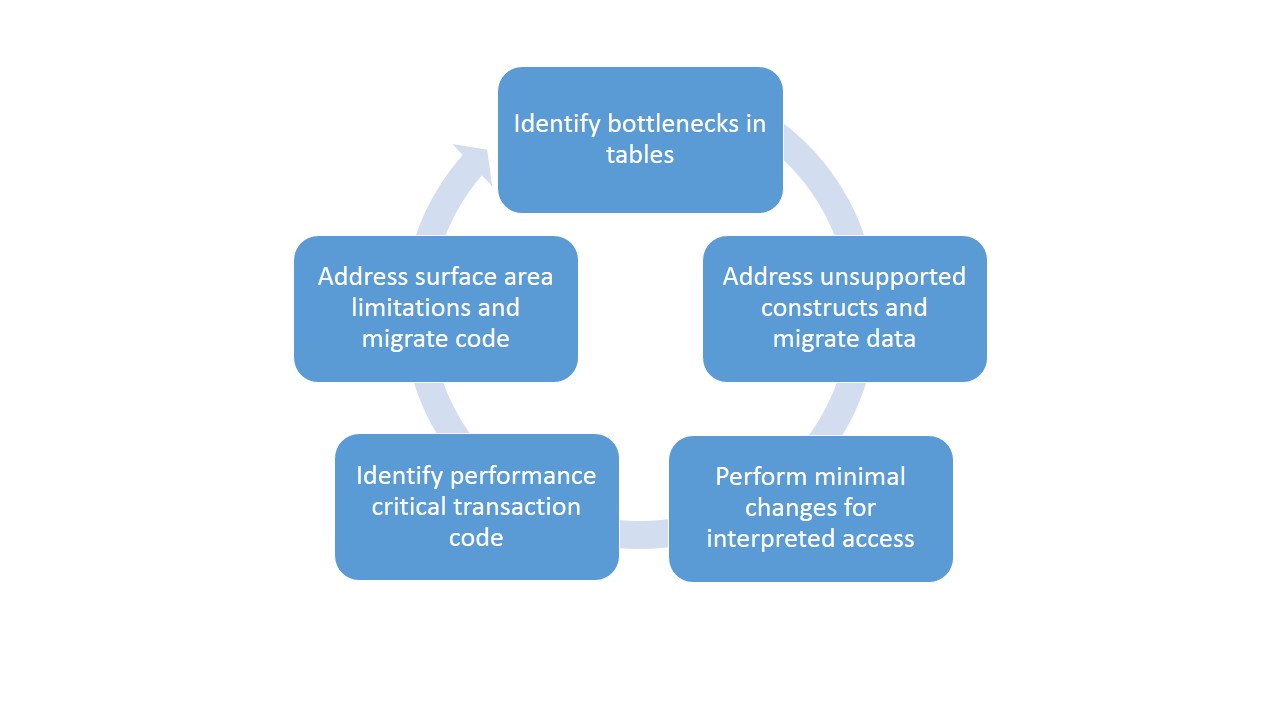


Figure - Migration methodology

As you migrate certain bottleneck areas to In-Memory OLTP, other parts of the system may become the bottleneck. Consider an iterative approach to migration when addressing a particular bottleneck, and then analyze where the next performance bottleneck for the solution may reside.

## Further Implementation Considerations

This section does not provide a comprehensive list of the In-Memory OLTP technology. This section provides guidance on some key considerations for working with features of this new technology. We highlight a number of technologies or decision points that are new or dramatic changes when compared to the traditional SQL Server implementation.

### Hardware or System Impact Considerations

Performance sensitive workloads depend on all components in the deployment including software and hardware. In the following sections, we discuss some critical factors to consider for hardware choices and some implementation considerations when deploying In-Memory OLTP. Microsoft developed In-Memory OLTP based on the concept of allowing applications to use existing and commodity hardware available in the market today. In-Memory OLTP does not require extremely large amounts of sockets/cores or other hardware components. In-Memory OLTP targets optimizations within SQL Server that can interact with standard commodity hardware.

#### Memory

Memory-optimized tables fully reside in memory and do not respond to memory pressure exerted on or within SQL Server. Disk-based tables have buffer pool pages that can be flushed to disk and the memory pool can shrink. However, rows in memory-optimized tables are not flushed to disk, and allocations will reside in memory until garbage is collected. Therefore, you must allocate enough memory to SQL Server to accommodate the entire dataset allocated to memory-optimized tables. This will limit the amount of memory available for other memory pools such as the buffer pool, the plan cache, lock manager, and others.

When evaluating the data and objects for migration, determining the size of the memory that is needed is of critical importance. You can perform the base calculation using the following criteria:

* The number of rows (estimate for growth).
* The size of data (data-types).
* Header information saved in each row including start and end timestamps and index pointers.
* The space for index allocations.

However, one of the most critical and variable considerations will be a function of understanding the workload. Having an idea of how many versions of rows may reside in memory before garbage collection is important because this will require additional memory allocations. Row versions must stay in memory until all the transactions that would utilize older row versions are complete. Also, garbage collection of these rows is asynchronous to the actual process. For example, after a deletion is executed, it can take some time to remove the row versions from memory. Certainly there is variance based on the type of transactions and queries executed in the workload. However, for fairly update heavy, or read/write calls iterating over the same values in an OLTP workload, consider a value of around 2x. This means two row versions for each row that exists in the table. Please note that, under memory pressure, this garbage collection process is expedited. For further information on determining the size of memory-optimized tables, see “[Table and Row Size in Memory-Optimized Tables](http://msdn.microsoft.com/en-us/library/dn205318(v=sql.120).aspx)”.

If you have already migrated tables, there are also ways to view memory taken by memory-optimized tables and indexes. Use the DMV: sys.dm\_db\_xtp\_table\_memory\_stats, or use Perfmon SQL Server Database counters: XTP Memory Used (kb).

Overall, considering how to utilize Resource Governance for memory pools can be helpful in allocating memory for memory-optimized tables in a database. For a detailed discussion, refer to the “[Resource Governor](http://msdn.microsoft.com/en-us/library/bb933866(v=sql.120).aspx)” section of Books Online.

Finally, it is important to consider the size of the dataset relative to durable tables in memory. The amount of data will impact recovery time. Additionally, there is a relationship between the storage on disk, which does have some size limits, and data in memory. The recommended limit for the total size of all durable tables in a database is at or below 250 GB. Durable tables requiring 250 GB of space in memory will require, on average, twice the storage space (500 GB) in the memory-optimized filegroup. This estimate assumes a mix of insert, delete, and update operations. This would lead to approximately 4000 data/delta file pairs allocated in this group.

Bursts of activity in the database may cause checkpoint operations to fall behind for a period of time. This lag will increase the number of required files. As a cushion for such bursts, the storage system supports up to 8000 data/delta file pairs. When the system reaches that limit, it will prevent new transactions in the database until checkpoint operations catch up. This cushion does provide for the possibility of managing durable tables with a size greater than 250GB. However, we do not recommended running above 250 GB for an extended period of time. The cushion only exists to handle bursts of activity and stalls in checkpoint. Exceeding the recommendation can increase the risk of throttling transaction activity on the memory-optimized tables in the database.

#### Storage/Disk Subsystem

For memory-optimized tables created with SCHEMA\_AND\_DATA, durability considerations for disk layout and file placement are important. The storage requirements are also very different from those of disk-based SQL Server tables.

Memory-optimized tables execute I/O to the database transaction log. Therefore, the I/O latency is still a factor in the overall transaction execution time. For the log drive, it is important to consider the disk subsystem latency characteristics.

##### Data and Delta Files - Checkpoint File Pairs (CFPs)

The system stores data for durable tables in data and delta files, also referred to as Checkpoint File Pairs (CFPs). The system stores this data in an append-only manner using a background thread. Data files contain inserted records, and delta files store deleted records. Storing data in this manner eliminates random I/O. These files leverage a FILESTREAM based storage mechanism. File storage in the memory-optimized filegroup for the SQL Server database is very different from the standard storage for disk-based tables. The overall size of the files on disk may be much larger than the amount of memory used by in-memory objects. When creating memory-optimized tables, the system pre-allocates CFPs to minimize any delays in allocating new files as transactions are executing. The data file size is 128 MB (or 16 MB on servers with fewer than 16 GB of memory). The delta files sizes are 8 MB and 1 MB respectively, but they contain no data. The number of CFPs is computed as the number of logical processors or schedulers, and there is a minimum of 8. This is a fixed storage overhead in databases with memory-optimized tables. The overall storage space that CFPs use is a function of the workload, checkpoint, and log truncation characteristics of the workload. The size can vary, however, as an initial point of reference consider allocating disk space that is approximately four times (4x) the size of the memory-optimized tables in memory. It is important to underscore the importance of a log backup plan under the FULL or Bulk-Logged recovery models that meets the business recovery requirements and addresses the storage behavior of memory-optimized tables. The backup plan will help determine log truncation and impact the garbage collection and is therefore critical to maintaining an acceptable number and size of checkpoint files on disk. For more information regarding the data and delta files and checkpoints, see “[Durability for Memory-Optimized Tables](http://msdn.microsoft.com/en-us/library/dn553125(v=sql.120).aspx)” and “[Checkpoint Operations for Memory-Optimized Tables](http://msdn.microsoft.com/en-us/library/dn553124(v=sql.120).aspx)”.

There are also performance considerations to manage when implementing durable memory-optimized tables. The placement of data files is important for:

1. Offline checkpoint performance.
2. Performing the merge for the checkpoint file containers.
3. Recovery in the case of service failure. This will load the containers, via streaming I/O, which loads the files into memory and recreates the table.

In the steady state case of checkpoint writes, merge, and file clean-up, there will be I/O for:

* Checkpoint writes.
* Reads from source files participating in merge operations.
* Writes to the destination merge file.

Having a disk subsystem that can handle this I/O volume is important. In the case of recovery performance, placement of the data and delta containers is important. Consider creating multiple containers on the memory-optimized filegroup and spreading them over different drives. Doing this provides more bandwidth for streaming the data into memory. When you do this, understand that the system allocates data and delta files in a round-robin manner across the given files. The system associates a data file to the first file it creates and associates a delta file to the second file it creates, and so on. To obtain a balanced stream of I/O on recovery, consider placing pairs of files on the same spindles/storage. For example, File 1 and File 2 on drive X and File 3 and File 4 on drive Y. Take into account that both data and delta file pairs reside on drive X and drive Y.

#### Processor

It is vital to consider the deployment of the processors. In particular, you should think about the number of cores and sockets for an In-Memory OLTP solution. We have collected observations from customer and internal testing for the current release. We have observed that the overhead of cross-socket communications in some of the larger socket machines may affect scalability. You should target two or four sockets and under 60 cores to achieve the most successful implementation.

### Multi-version Optimistic Concurrency Considerations

Memory-optimized tables do not suffer from locking, latching, or spinlocks. The reason for this is that the core In-Memory OLTP engine uses lock-free optimistic isolation and latch-free and spinlock-free internal structure protection. Migration of tables that experience latching, locking, or spinlock contention to memory-optimized tables completely eliminates this scale bottleneck.

However, this new paradigm of concurrency control introduces changes in behavior and may require adjustments to either the Transact-SQL error handling, such as try-catch blocks, or to the application code in order to accommodate for conflicts. With the default pessimistic isolation, two transactions that attempt to modify the same resource will block each other. One will get serialized because one will need to wait for the other to release its locks. With optimistic isolation, transactions will not block each other, but may fail to commit because of update or validation conflicts. In this case, the system must resubmit the transaction. Resubmission effectively serializes the transaction order and incurs additional overhead for rollback and resubmission. Writer-writer conflicts are not common in most OLTP environments. For further discussion on retry logic for conflicts, read “[Guidelines for Retry Logic for Transactions on Memory-Optimized Tables](http://msdn.microsoft.com/en-us/library/dn169141(v=sql.120).aspx)”.

In-Memory OLTP supports REPEATABLE READ, SERIALIZABLE, and SNAPSHOT isolation modes only. Many applications were designed to work with the traditional SQL Server default READ COMMITTED isolation level. Use the new “MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOT” database option to map all transactions that use READ COMMITTED automatically to SNAPSHOT. Using this option does not require application code changes.

For more details, see the “[Guidelines for Transaction Isolation Levels with Memory-Optimized Tables](http://msdn.microsoft.com/en-us/library/dn133187(v=sql.120).aspx)” section in Books Online and “[SQL Server 2014 In-Memory OLTP Internals Overview](http://download.microsoft.com/download/5/F/8/5F8D223F-E08B-41CC-8CE5-95B79908A872/SQL_Server_2014_In-Memory_OLTP_TDM_White_Paper.pdf)”.

### Indexing Guidelines

There are two new index types with In-Memory OLTP: memory-optimized nonclustered indexes and nonclustered hash indexes. Both indexes, as mentioned earlier, are created and exist entirely in memory.

Memory-optimized nonclustered indexes are similar in concept to standard B-tree indexes, but they are implemented in a lock and latch free manner. The memory-optimized nonclustered indexes support point/singleton row and range lookups for queries. Because they support ordered scans, they are the index of choice for non-equality range searches. They can also be very useful for queries that require ordering of the result sets and grouping and aggregation. Nonclustered indexes also support partial key lookups. Nonclustered indexes are similar to indexes used for disk-based tables. When compared to hash indexes, nonclustered indexes have a relative lack of requirements for creation. Therefore, we recommend that you use these as a starting point when creating memory-optimized tables.

Memory-optimized hash indexes are very efficient for point-lookup style queries, such as retrieving or modifying a row or a set of rows based on key value. With hash indexes, row pointers are stored in a hash table. Each pointer points to a chain of rows that have the same hash value for the index key. When creating a hash index, you must specify a hash bucket count. This will be used to build the hash table. Specifying too few buckets will cause multiple key values to share a bucket, which could lead to scanning of redundant data. Scanning redundant data will degrade performance. Specifying too many buckets will result in an over allocation of memory because many buckets will be left empty. Generally, creating buckets of up to 5X the number of unique key values for the index columns does not significantly impact performance. Remember that the hash value is computed across all key values. Therefore, a filter on only a subset of the keys (or none) cannot use the index. This will result in a scan of all buckets.

You can create nonclustered hash and nonclustered indexes on the same set of columns to satisfy different query patterns. Both index types are optimal for low density data distributions. With hash indexes, hash collisions can negatively affect performance. With nonclustered memory-optimized indexes, high density values make the index less efficient for lookups but still retains ordered scan benefits.

You determine the bucket count when you create the table and you cannot modify the bucket count afterwards. To change a hash index’s bucket count, you must create a new table, specify the new bucket count, and decide how to migrate the data. This is a non-trivial process that will impact data availability.

### Query Execution

Memory-optimized tables support data distribution statistics that are similar to disk-based tables. The query optimizer uses these statistics to generate an optimal execution plan for queries involving memory-optimized tables. Statistics are referenced at different times depending on the type of code being executed. For interpreted Transact-SQL, the query plan is optimized and statistics are used during the compilation process. For every execution of the statement, statistics may be used to optimize an execution plan. For natively compiled stored procedures, the statistics are only utilized once when the stored procedure is being compiled, at creation time. Further statistics updates will be only utilized again whenever the database goes offline or there is a service restart.

Consider the following aspects of statistics creation and utilization when planning migration to In-Memory OLTP:

* Memory-optimized tables do not support statistics collection sampling and require a full scan of the table. The full scan can have a significant impact on tables with a very large number of rows. The default database setting is to have auto-create statistics ON. With this setting, when a query executes, missing statistics may be created automatically to help produce an optimal plan. With memory-optimized tables that require a full scan, this will result in a longer compile time for queries. For Interpreted T-SQL, this impact will appear when the statement executes. For natively compiled stored procedures, this will affect the time it takes to create the stored procedure. However, it will not affect the execution time of natively compiled stored procedures.
* With natively compiled stored procedures, statistics are used only when the stored procedure is created. This means that if you create the procedure before the table is populated with real data, the execution plan may be suboptimal. The query optimizer will be working with wrong assumptions regarding the data distribution. Therefore, we recommended that you create the procedures only after the tables are fully populated with faithfully representative data and a statistics update is executed.
* Unlike disk-based tables, statistics on memory-optimized tables are not updated automatically. Statistics updates will execute a full sampling of data. Be careful about the timing when you execute the statistics updates, and consider the impact to the workload. Take into account the type of workload that is executing in terms of the number of rows being inserted or modified. Consider how stale the statistics may be and how sensitive the queries are to data distribution changes. If you do not update statistics or update very infrequently, the query optimizer may use wrong distribution statistics and come up with suboptimal plans. On the other hand, updating too frequently will add an unnecessary overhead.

As mentioned earlier, consider the fact that queries that access memory-optimized tables cannot utilize parallel plans. All queries containing memory-optimized tables will use serial plans exclusively. For example, joins between disk-based tables that use B-tree or columnstore indexes can benefit greatly from parallelism. Migrating any of the joined tables to In-Memory OLTP will block the query optimizer from using this optimization. For cases where it is necessary to utilize parallelism for particular queries, consider keeping the dataset servicing those queries in disk-based tables. This solution is similar to the concept discussed in the ‘Shock Absorber’ scenario.

## Scenarios Less Suitable for Migration

In the earlier sections on assessing the workload viability for In-Memory OLTP, we highlighted specific components within the engine that In-Memory OLTP improves. There are enhancements for data access and procedure execution components. For this release, there are no improvements for network and connectivity components. The following sections discuss scenarios that may impede success with In-Memory OLTP in SQL Server 2014 and therefore should be considered by applications looking to take advantage of the performance gains that this technology provides.

### Lacking the Ability to Make Code or Schema Changes

At a minimum, users must create tables as memory-optimized to migrate to the In-Memory OLTP engine. There are scenarios where the schema cannot be modified. In other cases, the cost to modify code to support the memory-optimized table or natively compiled stored procedure programming surface area is prohibitive. Some applications use frameworks (such as Entity Framework) that may enlist calls to Multiple Active Results Sets (MARS) or use distributed transactions. These applications may require code changes to handle transaction enlistment and connection string settings. Finally, locking hints or isolation level incompatibilities with In-Memory OLTP supported isolation levels may also require some application modifications.

### Memory Limitations

As mentioned earlier, data structures that make up memory-optimized tables are all stored in memory, and unlike traditional B-tree objects are not backed by durable storage. Scenarios where sufficient memory is not available to store the memory-optimized rows can be problematic. When evaluating migration, determine the size of memory required. It is also critical to consider the workload that may produce multiple versions of rows, which requires additional memory allocations.

### Workload Pattern is not OLTP

In-Memory OLTP is optimized for OLTP workloads. Long running transactions, queries and reports that involve many tables or execute large aggregations of data are more likely to cause retention of several row versions. These workloads are more likely to put pressure on memory to the point where it may be unable to accommodate these transactions, queries, and reports. Similarly, disk-based tables and indexes may be better for very large datasets and provider greater benefits for these longer query executions. Alternative solutions such as a nonclustered or updatable clustered columnstore indexes may be more appropriate for this part of the workload. Workloads and schemas that are very dynamic may not be the best fit because altering table and index definitions will involve recreating the In-Memory OLTP objects.

Workload types such as those that benefit from Full-Text Search or XML parsing can be better serviced using specialized indexes and data structures created specifically for working with those data structures and patterns.

### Lock Behavior Dependent Applications

Within the In-Memory OLTP engine there is no locking. In many cases, the multi-version optimistic concurrency implementation can provide significant scale benefits. However, in some applications there are dependencies on locking constructs to restrict access to data. With the exception of an application lock, In-Memory OLTP does not offer the ability to lock records like standard SQL Server does. If the application logic depends on the physical locking of records (e.g., queues implemented with READPAST lock-skip hints), this may not be a good migration candidate.

Instead of taking out locks to prevent concurrent access to data, memory-optimized tables use conflict detection to enforce data modification isolation. Workloads that are write heavy and are modifying a single row may experience a number of these conflicts. In some cases, it is possible to minimize these conflicts. However, sometimes there are a significant number of conflicts that are unavoidable. In this case, the overhead for handling the conflicts and retrying may eliminate most of the performance gains from In-Memory OLTP. A “queue” pattern with a large number of concurrent users trying to update the first row on the stack is an example of such a case. Under contentious workloads this suffered from a large number of conflicts.

# Conclusion

There are number of scenarios that require high performance systems that traditional relational database management systems are unable to deliver. In-Memory OLTP can help workloads achieve and exceed their required performance metrics while integrating with other familiar features of SQL Server 2014.

An understanding of the current bottlenecks and database contention in the application is critical to determining how to best utilize the In-Memory OLTP components to achieve the overall application performance goals. SQL Server provides a set of tools, including data collection sets and reports, to help assess candidates for migration. SQL Server also provides advisor tools to help with the actual migration. When migrating tables, data, and Transact-SQL code to In-Memory OLTP, target the performance critical database sections of the application. Performing the migration in an iterative manner can result in less overall application disruption.

# For more information:

* [SQL Server 2014 In-Memory OLTP Internals Overview Whitepaper](http://download.microsoft.com/download/5/F/8/5F8D223F-E08B-41CC-8CE5-95B79908A872/SQL_Server_2014_In-Memory_OLTP_TDM_White_Paper.pdf)
* <http://msdn.microsoft.com/en-us/library/ms130214(v=sql.120).aspx>: SQL Server Books Online
* <http://www.microsoft.com/sqlserver/>: SQL Server Web site
* <http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter
* <http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter
* <http://msdn.microsoft.com/en-us/library/dn133189(v=sql.120).aspx> SQL Server Support for In-Memory OLTP.

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high because it has good examples, excellent screen shots, clear writing, or another reason?
* Are you rating it low because of poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of the white papers we release.

[Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20[Title%20of%20White%20Paper]).