# In-Memory OLTP (In-Memory Optimization)

In-Memory OLTP can significantly improve the performance of transaction processing, data ingestion and data load, and transient data scenarios.

* *Memory-optimized tables* are used for storing user data. You declare a table to be memory-optimized at create time.
* *Non-durable tables* are used for transient data, either for caching or for intermediate result set (replacing traditional temp tables). A non-durable table is a memory-optimized table that is declared with DURABILITY=SCHEMA\_ONLY, meaning that changes to these tables do not incur any IO. This avoids consuming log IO resources for cases where durability is not a concern.
* *Memory-optimized table types* are used for table-valued parameters (TVPs), as well as intermediate result sets in stored procedures. These can be used instead of traditional table types. Table variables and TVPs that are declared using a memory-optimized table type inherit the benefits of non-durable memory-optimized tables: efficient data access, and no IO.
* *Natively compiled T-SQL modules* are used to further reduce the time taken for an individual transaction by reducing CPU cycles required to process the operations. You declare a Transact-SQL module to be natively compiled at create time. At this time, the following T-SQL modules can be natively compiled: stored procedures, triggers, and scalar user-defined functions.

**OVERVIW OF MEMORY OPTIMIZED TABLES**

* Memory Table Definition
* Tables constructed and maintained entirely in system memory
* A copy of the table can be maintained on disk for durability
* Non-durable tables supported.
* No locking during data modifications
* At least One Index
* Coexist with Disk based tables
* Queried using T-SQL
* Text and Image Datatypes are not supported

Memory optimization tables used row versioning entirely to implement changes.

**Use Cases for Memory Optimized Tables**

* **High-throughput and low-latency transaction processing**

This is the core scenario for which we built In-Memory OLTP: support large volumes of transactions, with consistent low latency for individual transactions.

* **Data ingestion, including IoT (Internet-of-Things)**

In-Memory OLTP is good at ingesting large volumes of data from many different sources at the same time. And it is often beneficial to ingest data into a SQL Server database compared with other destinations, because SQL Server makes running queries against the data fast, and allows you to get real-time insights.

Common application patterns are:

* Ingesting sensor readings and events, and allow notifications as well as historical analysis.
* Managing batch updates, even from multiple sources, while minimizing the impact on the concurrent read workload.
* **Tempdb object replacement**

Leverage non-durable tables and memory-optimized table types to replace your traditional TempDB based structures, such as temporary tables, table variables, and table-valued parameters (TVPs).

Memory-optimized table variables and non-durable tables typically reduce CPU and completely remove log IO, when compared with traditional table variables and #temp table.

* **ETL (Extract Transform Load)**

ETL workflows often include load of data into a staging table, transformations of the data, and load into the final tables.

#### Implementation considerations

Use non-durable memory-optimized tables for the data staging. They completely remove all IO, and make data access more efficient.

If you perform transformations on the staging table as part of the workflow, you can use natively compiled stored procedures to speed up these transformations. If you can do these transformations in parallel you get additional scaling benefits from the memory-optimization.

* **Concurrent table access**
* Latch-bound workloads
* Modification of large number of rows.
* **HOT Pages**
* A Clustered Index with an increment Key value – All Inserts always occurs on the most recent pages of the index.

-- ADD Memory Optimized files group and container

Alter Database DOCD ADD fileGROUP [FileGroupName] CONTAINS MEMORY\_OPTIMIZED\_DATA

ALter Database DOCD ADD FILE (name= 'file\_name',filename='c:\file\_name')

ALter Database DOCD SET MEMORY\_OPTIMIZED\_ELEVATED\_TO\_SNAPSHOT=ON

CREATE TABLE Table1

(

id INT Identity(1,1) Primary Key NONCLUSTERED,

Col1 INT NOT NULL

) With(Memory\_Optimized=ON,Durability = SCHEMA\_ONLY) -- SCHEMA\_AND\_DATA

**Memory Optimized table statistics**

* Column Statistics
  + Query Plans
  + Improved Query Performance
* Statistics Automatically Created
* Statistics Automatically Updated

**Recommendations for Memory Optimized Table Statistics**

* Database Compatibility Level (130- SQL server 2016)
* Natively Compiled Stored Procedures
* Manual recompile using sp\_recompile

**Querying Memory- Optimized Tables**

* **Two Primary Methods**
* Interpreted Transact-SQL
  + Traditional Access to Disk based tables
  + Interop layer provided by SQL Server
  + Can be used for queries that combine disk based and memory –optimized tables.
* Natively Compiled Stored Procedures
  + Increased Performance
  + Procedures are converted into native C code and compiled when created as opposed to at runtime
  + Can only be used to access memory-optimized tables
  + C code is compiled into DLL which is then loaded into memory.

**Natively Compiled Stored Procedures**

**Native compilation** is the process by which a stored SQL program is **compiled** into **native** code that does not need to be interpreted at runtime.

If you do not use native compilation, each SQL program unit is compiled into an intermediate form, machine-readable code (MCode). The MCode is stored in the database dictionary and interpreted at run time. With SQL native compilation, the SQL program is compiled into machine native code that bypasses all the runtime interpretation, giving faster runtime performance.

In-Memory OLTP introduces the concept of native compilation. SQL Server can natively compile stored procedures that access memory-optimized tables. SQL Server is also able to natively compile memory-optimized tables. Native compilation allows faster data access and more efficient query execution than interpreted (traditional) Transact-SQL. Native compilation of tables and stored procedures produce DLLs.

CREATE PROCEDURE dbo.native\_sp with native\_compilation

* Stored Procedures Written in T-SQL
* Compiled into Native ( C ) Code when created
* Stored as DLL and loaded into memory
* Used to access memory optimized tables
* Greater Speed and efficiency
* Must be schema bound to the objects they reference means reference tables cannot be dropped as long as the procedure exists
* Requires an Execution context – Execute as CALLER not supported
* Traditional Disk based Stored Procedures
* Compiled when first run
* Errors in compilation may not revealed at creation time
* T-SQL code must be interpreted

**Using natively compiled stored procedures**

* In-Memory OLTP
* Implanting or converting database tables to memory optimized tables.
* Increased speed and efficiency are of the utmost important
* Offer significant performance improvement over disk based stored procedures.
* Transitioning to In-memory OLTP can be time consuming.

**Restrictions:**

T-SQL constructs not supported by natively compiled stored procedures

* EXISTS
* MERGE
* CASE Statements
* OUTER JOINS
* TempDB cannot be used
* Alter procedure
* Cursors
* CTE

**Atomic Blocks in Native Stored Procedures**

If there is no active transaction on a session, **BEGIN ATOMIC** will start a new transaction. If no exception is thrown outside the scope of the block, the transaction will be committed at the end of the block. If the block throws an exception (that is, the exception is not caught and handled within the block), the transaction will be rolled back. For transactions that span a single atomic block (a single natively compiled stored procedure), you do not need to write explicit **BEGIN TRANSACTION** and **COMMIT** or **ROLLBACK** statements.

Natively compiled stored procedures support the **TRY**, **CATCH**, and **THROW** constructs for error handling. **RAISERROR** is not supported.

* Supported by SQL server at top level of natively compiled
* Stored Procedures
* Scalar User defined stored procedures.
* Executed with in Transaction
* All block statements succeed or rollback to save point
* Atomic Block
* Exactly one block of statements

**Atomic Blocks**

* Fixed Session settings
* Transaction isolation level
* SNAPSHOT, REPEATABLE READ, SERIALIZATION
* Optional Session Settings
* Date Format
* Date First
* Delayed Durability

**Best Practices for Natively compiled Stored Procedures**

* Use When Application Performance was critical
* Frequently used/Execution
* Very fast results are required
* Can only access memory optimized tables.
* Performance gains becomes more noticeable as the number of rows and amount of logic processed increases
* Stored procedures that only return or affect small number of rows may not show any significant improvement in performance.
* Use in more complex statements that performs:
* Aggregations
* Nested Loops
* Multi statement DML operations
* Complex Expression
* Conditional Statements and Procedural logic.