SQL Server In-Memory OLTP Internals Overview for CTP1

SQL Server Technical Article

Writer: Kalen Delaney

Technical Reviewers: Kevin Liu, Jos de Bruijn, Kevin Farlee, Mike Zwilling, Sunil Agarwal, Craig Freedman, Mike Weiner, Cristian Diaconu, Pooja Harjani, Darmadi Komo, Dandy Weyn

Published: June 3 2013

Applies to: SQL Server 2014 CTP1

Summary: In-Memory OLTP (formally known as code name “Hekaton”) is a new database engine component, fully integrated into SQL Server. It is optimized for OLTP workloads accessing memory resident data. In-Memory OLTP allows OLTP workloads to achieve remarkable improvements in performance and reduction in processing time. Tables can be declared as ‘memory optimized’ to take advantage of In-Memory OLTP’s capabilities. In-Memory OLTP tables are fully transactional and can be accessed using Transact-SQL. Transact-SQL stored procedures can be compiled into machine code for further performance improvements if all the tables referenced are In-Memory OLTP tables. The engine is designed for high concurrency and blocking is minimal.

Copyright

Contents

Introduction

SQL Server was originally designed at a time when it could be assumed that main memory was very expensive, so data needed to reside on disk except when it was actually needed for processing. This assumption is no longer valid as memory prices have dropped enormously over the last 30 years. At the same time, multi-core servers have become affordable so that today one can buy a server with 32 cores and 1TB of memory for under $50K. Since many, if not most, of the OLTP databases in production can fit entirely in 1TB, we need to re-evaluate the benefit of storing data on disk and incurring the I/O expense when the data needs to be read into memory to be processed. In addition, OLTP databases also incur expenses when this data is updated and needs to be written back out to disk. Memory-optimized tables are stored completely differently than disk-based tables and these new data structures allow the data to be accessed and processed much more efficiently.

Because of this trend to much more available memory and many more cores, the SQL Server team at Microsoft began building a database engine optimized for large main memories and many-core CPUs. This paper gives a technical overview of this new database engine In-Memory OLTP.

Design Considerations and Purpose

The move to produce a true main-memory database has been driven by three basic needs: 1) fitting most or all of data required by a workload into main-memory, 2) lower latency time for data operations, and 3) specialized database engines that target specific types of workloads need to be tuned just for those workloads. Moore's law has impacted the cost of memory allowing for main memories to be large enough to satisfy (1) and to partially satisfy (2). (Larger memories reduce latency for reads, but don't affect the latency required for writes to disk needed by traditional database systems). Other features of SQL Server In-Memory OLTP, which will be discussed in this document, allow for greatly improved latency for data modification operations. The need for specialized database engines is driven by the recognition that systems designed for a particular class of workload can frequently out-perform more general purpose systems by a factor of 10 or more. Most specialized systems, including those for CEP (complex event processing), DW/BI and OLTP, optimize data structures and algorithms by focusing on in-memory structures.

Microsoft's impetus for creating In-Memory OLTP comes mainly from this fact that main memory sizes are growing at a rapid rate and becoming cheaper. In addition, because of the near universality of 64 bit architectures and multicore processors, it is not unreasonable to think that most, if not all, OLTP databases or the entire performance sensitive working dataset could reside entirely in memory. Many of the largest financial, online retail and airline reservation systems fall between 500GB to 5TB with working sets that are significantly smaller. As of 2012, even a two socket server could hold 2TB of DRAM using 32GB DIMMS (such as IBM x3680 X5). Looking further ahead, it's entirely possible that within a few years you'll be able to build distributed DRAM based systems with capacities of 1-10 Petabytes at a cost less than $5/GB. It is also only a question of time before non-volatile RAM becomes viable.

If most or all of an application's data is able to be entirely memory resident, the costing rules that the SQL Server optimizer has used since the very first version become almost completely meaningless, because the rules assume all pages accessed can potentially require a physical read from disk. If there is no reading from disk required, the optimizer can use a different costing algorithm. In addition, if there is no wait time required for disk reads, other wait statistics, such as waiting for locks to be released, waiting for latches to be available, or waiting for log writes to complete, can disproportionately large. In-Memory OLTP addresses all these issues. In-Memory OLTP removes the issues of waiting for locks to be released using a completely new type of multi-version optimistic concurrency control. It ameliorates the delays of waiting for log writes by generating far less log data and needing fewer log writes. The details of exactly how concurrency is managed with no locking or latching, as well as more details of what gets logged when modifying memory-optimized tables will be covered in a subsequent whitepaper.

Terminology

In this paper, I will be using the term *In-Memory OLTP* to refer to a suite of technologies for working with memory-optimized tables in SQL Server. The alternative to memory-optimized tables, which SQL Server has always provided, will be referred to as *disk-based tables*. Terms to be used include:

- *Memory-optimized tables* refer to tables using the new data structures added as part of In-Memory OLTP, and will be described in detail in this paper.
- *Disk-based tables* refer to the alternative to memory-optimized tables, and use the data structures that SQL Server has always used, with pages of 8K that need to be read from and written to disk as a unit.
- *Natively compiled stored procedures* refer to an object type supported by In-Memory OLTP that is compiled to machine code and has the potential to increase performance even further than just using memory-optimized tables. The alternative is *interpreted TSQL* stored procedures, which is what SQL Server has always used. Natively compiled stored procedures can only reference memory-optimized tables.
- *Cross-container transactions* refer to transactions that reference both memory-optimized tables and disk-based tables.
- *Interop* refers to interpreted TSQL that references memory-optimized tables

Overview of Functionality

During most of your data processing operations with In-Memory OLTP, you may be completely unaware that you are working with memory-optimized tables rather than disk-based tables. However, behind the scenes SQL Server is working with your data very differently if it is stored in memory-optimized tables. In this section, we'll look at an overview of how In-Memory OLTP operations and data are handled differently than disk-based operations and data in SQL Server. We'll also briefly mention some of the memory optimized database solutions from competitors and point out how SQL Server In-Memory OLTP is different from them.

What's Special About In-Memory OLTP

Although In-Memory OLTP is integrated with the SQL Server relational engine, and can be accessed using the same interfaces transparently, its internal behavior and capabilities are very different. Figure 1 gives an overview of the SQL Server engine with the In-Memory OLTP components.

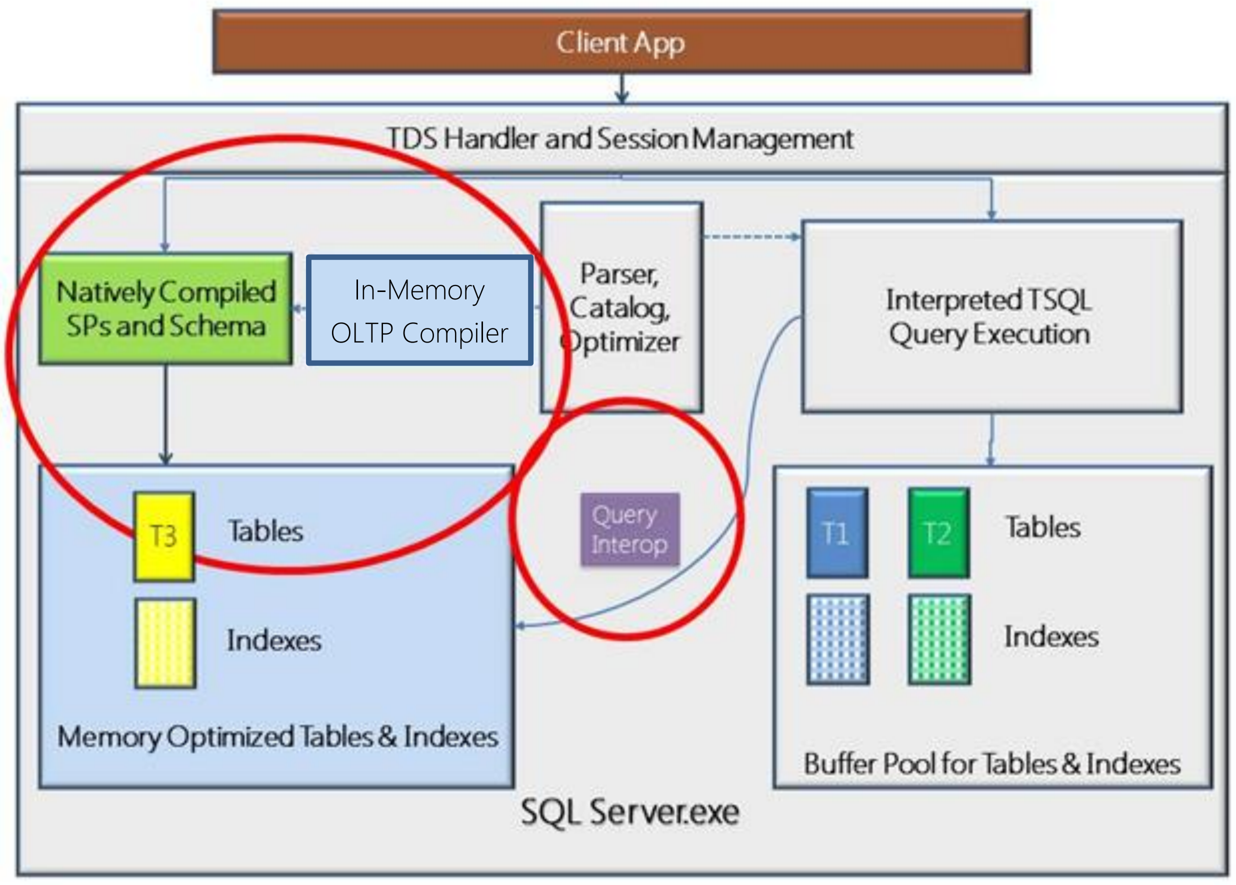
Client App
TDS Handler and Session Management
Natively Compiled SPs and Schema
In-Memory OLTP Compiler
Parser, Catalog, Optimizer
Interpreted TSQL Query Execution
Query Interop
T3
Tables
Indexes
Memory Optimized Tables & Indexes
T1
T2
Tables
Indexes
Buffer Pool for Tables & Indexes
SQL Server.exe

Figure 1The SQL Server engine including the In-Memory OLTP component

Notice that the client application connects to the TDS Handler the same way whether it will be working with memory-optimized tables or disk-based tables, whether it will be calling natively compiled stored procedures or interpreted TSQL.   You can see that interpreted TSQL can access memory-optimized tables using the interop capabilities, but that natively compiled stored procedures can only access memory-optimized tables.

Memory-optimized tables

The most important difference between memory-optimized tables and disk-based tables is that pages do not need to be read into cache from disk when the memory-optimized tables are accessed. All the data is stored in memory, all the time. A set of checkpoint files which are only used for recovery purposes, is created on filestream filegroups that keep track of the changes to the data, and the checkpoint files are append-only.

Operations on memory-optimized tables use the same transaction log that is used for operations on disk-based tables, and as always, the transaction log is stored on disk. In case of a system crash or server shutdown, the rows of data in the memory-optimized tables can be recreated from the checkpoint files and the transaction log.

In-Memory OLTP does provide the option to create a table that is non-durable and not logged using an option called SCHEMA_ONLY. As the option indicates, the table schema will be durable, even though the data is not. These tables do not require any IO operations during transaction processing, but the data is only available in memory while SQL Server is running. In the event of a SQL Server shutdown or an AlwaysOn failover, the data is lost. The tables will be recreated when the database they belong to is recovered, but there will be no data in the tables. These tables could be useful, for example, as staging tables in ETL scenarios or for storing Web server session state. Although the data are not durable, operations on these tables meet all the other transactional requirements; i.e. they are atomic, isolated, and consistent. We'll see the syntax for creating a non-durable table in the section on Creating Tables.

Indexes on memory-optimized tables

Indexes on memory-optimized tables are not stored as traditional B-trees. Memory-optimized tables support hash indexes, stored as hash tables with linked lists connecting all the rows that hash to the same value and range indexes, which are stored using special BW-trees. Hash indexes will be described below. Range indexes and BW-trees are not available in CTP 1, so they will be described in a later paper.

Every memory-optimized table must have at least one index, because it is the indexes that combine all the rows into a single table. Memory-optimized tables are never stored as unorganized sets of rows, like a disk-based table heap is stored.

Indexes are never stored on disk, and are not reflected in the on-disk checkpoint files and operations on indexes are never logged. The indexes are maintained automatically during all modification operations on memory-optimized tables, just like b-tree indexes on disk-based tables, but in case of a SQL Server restart, the indexes on the memory-optimized tables are completely rebuilt as the data is streamed into memory.

Concurrency improvements

When accessing memory-optimized tables, SQL Server uses completely optimistic multi-version concurrency control. Although SQL Server has previously been described as supporting optimistic concurrency control with the snapshot-based isolation levels introduced in SQL Server 2005, these so-called optimistic methods do acquire locks during data modification operations. For memory-optimized tables, there are no locks acquired, and thus no waiting because of blocking.

Note that this does not mean that there is no possibility of waiting when using memory-optimized tables. There are other wait types, such as waiting for a log write to complete. However, logging when making changes to memory-optimized tables is much more efficient than logging for disk-based tables, so the wait times will be much shorter. And there never will be any waits for reading data from disk, and no waits for locks on data rows.

Feature Set

The best execution performance is obtained when using natively compiled stored procedures with memory-optimized tables. However, there are limitations on the TSQL language constructs that are allowed inside a natively compiled stored procedure compared to the rich feature set available with interpreted code in the TSQL language. In addition, natively compiled stored procedures can only access memory-optimized tables and cannot reference disk-based tables.

Is In-Memory OLTP just an improved DBCC PINTABLE?

DBCC PINTABLE was a feature available in older versions of SQL Server that would not remove any data pages from a "pinned" table from memory, once those pages were read from disk. The pages did need to be read in initially, so there was always a cost for page reads the first time such a table was accessed. These pinned tables were no different than any other disk-based tables. They required the same amount of locking, latching and logging and they used the same index structures, which also required locking and logging. In-Memory OLTP memory-optimized tables are completely different than SQL Server disk-based tables, they use different data and index structures, no locking is used and logging changes to these memory-optimized tables is much more efficient that logging changes to disk-based tables.

Offerings from competitors

For processing OLTP data, there are two types of specialized engines. The first are main-memory databases. Oracle has TimesTen, IBM has SolidDB and there are many others that primarily target the embedded DB space. The second are applications caches or key-value stores (e.g. Velocity – App Fabric Cache and Gigaspaces) that leverage app and middle-tier memory to offload work from the database system. These caches continue to get more sophisticated and acquire database capabilities like transactions, range indexing and query capabilities (Gigaspaces already has these for example). At the same time database systems are acquiring cache capabilities like high-performance hash indexes and scale across a cluster of machines (VoltDB is an example). The In-Memory OLTP engine is meant to offer the best of both of these types of engines. One way to think of In-Memory OLTP is that it has the performance of a cache and the capability of a database. It supports storing your tables and indexes completely in memory, so you can create an entire database to be a complete in-memory system. It also offers high performance indexes and logging as well as other features to drastically improve query execution performance.

SQL Server In-Memory OLTP offers the following features that few (or none) of the competitions' products provide:

- Integration between memory-optimized tables and disk-based tables so that the transition to a completely memory resident database can be made gradually
- Natively compiled stored procedures to improve execution time for basic data manipulation operations by orders of magnitude
- Hash indexes specifically optimized for main memory access
- No storage of data on pages, removing the need for page latches.
- True multi-version optimistic concurrency control with no locking or latching for any operations

Getting Started with In-Memory OLTP

Using In-Memory OLTP

In-Memory OLTP is a part of SQL Server, starting with SQL Server 2014. CTPs will be available starting in June 2013. Installation of In-Memory OLTP will be done using the normal SQL Server setup application.

The In-Memory OLTP components will always be installed with a 64-bit edition of SQL Server 2014, and not available at all with a 32-bit edition.

Creating Databases

Any database that will contain memory-optimized tables needs be created with at least one MEMORY_OPTIMIZED_DATA filegroup. These filegroups are used for storing the checkpoint and delta files needed by SQL Server to recover the memory-optimized tables, and although the syntax for creating them is almost the same as for creating a regular filestream filegroup, it must also specify the option CONTAINS MEMORY_OPTIMIZED_DATA. Here is an example of a CREATE DATABASE statement for a database that can support memory-optimized tables:

```
CREATE DATABASE HKDB
    ON
    PRIMARY(NAME = [HKDB_data],
                     FILENAME = 'Q:\data\HKDB_data.mdf', size=500MB),
    FILEGROUP [SampleDB_mod_fg] CONTAINS MEMORY_OPTIMIZED_DATA
                     (NAME = [HKDB_mod_dir],
                     FILENAME = 'R:\data\HKDB_mod_dir'),
                     (NAME = [HKDB_mod_dir],
                     FILENAME = 'S:\data\HKDB_mod_dir')

  LOG ON (name = [SampleDB_log], Filename='L:\log\HKDB_log.ldf', size=500MB)
  COLLATE Latin1_General_100_BIN2;
```

Note that the above code example creates files on three different drives (Q:, R: and S:) so if you would like to run this code, you might need to edit the path names to match your system. The names of the files on R: and S: are identical, so if you create all these files on the same drive you'll need to differentiate the two file names.

Also notice a binary collation was specified. At this time, any indexes on memory-optimized tables can only be on columns using a Windows (non-SQL) BIN2 collation and natively compiled procedures only support comparisons, sorting, and grouping on those same collations. It can be specified (as done in the CREATE DATABASE statement above) with a default binary collation for the entire database, or you can specify the collation for any character data in the CREATE TABLE statement. (You can also specify collation in a query, for any comparison, sorting or grouping operation.)

It is also possible to add a MEMORY_OPTIMIZED_DATA filegroup to an existing database, and then files can be added to that filegroup. For example:

```
ALTER DATABASE AdventureWorks2012
       ADD FILEGROUP hk_mod CONTAINS MEMORY_OPTIMIZED_DATA;
GO
ALTER DATABASE AdventureWorks2012
       ADD FILE (NAME='hk_mod', FILENAME='c:\data\hk_mod')
    TO FILEGROUP hk_mod;
GO
```

Creating Tables

The syntax for creating memory-optimized tables is almost identical to the syntax for creating disk-based tables, with a few restrictions, as well as a few required extensions. Specifying that the table is a memory-optimized table is done using the MEMORY_OPTIMIZED = ON clause. A memory-optimized table can only have columns of these supported datatypes:

- bit
- All integer types: `tinyint`, `smallint`, `int`, `bigint`
- All money types: `money`, `smallmoney`
- All floating types: `float`, `real`
- date/time types: `datetime`, `smalldatetime`, `datetime2`, `date`, `time`
- `numeric` and `decimal` types
- All non-LOB string types: `char(n)`, `varchar(n)`, `nchar(n)`, `nvarchar(n)`, `sysname`
- Non-LOB binary types: `binary(n)`, `varbinary(n)`
- `Uniqueidentifier`

Note that none of the LOB data types are allowed; there can be no columns of type XML, CLR or the *max* data types, and all rows will be limited to 8060 bytes with no off-row data. In fact, the 8060 byte limit is enforced at table-creation time, so unlike a disk-based table, a memory-optimized tables with two varchar(5000) columns could not be created.

A memory-optimized table can be defined with one of two DURABILITY values: SCHEMA_AND_DATA or SCHEMA_ONLY with the former being the default. A memory-optimized table defined with DURABILITY=SCHEMA_ONLY, which means that changes to the table's data are not logged and the data in the table is not persisted on disk. However, the schema is persisted as part of the database metadata, so the empty table will be available after the database is recovered during a SQL Server restart.

As mentioned earlier, a memory-optimized table must always have at least one index but this requirement could be satisfied with the index created automatically to support a primary key constraint. All tables except for those created with the SCHEMA_ONLY option must have a declared primary key. At least one index must be declared to support a PRIMARY KEY constraint. The following example shows a PRIMARY KEY index created as a HASH index, for which a bucket count must also be specified. A few guidelines for choosing a bucket count will be mentioned when discussing details of hash index storage.

Single-column indexes may be created in line with the column definition in the CREATE TABLE statement (highlighted below). The BUCKET_COUNT attribute will be discussed in the second on Hash Indexes.

```
CREATE TABLE T1
(
  [Name] varchar(32) not null PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT = 1024),
  [City] varchar(32) null,
  [LastModified] datetime not null,

) WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Alternatively, composite indexes may be created after all the columns have been defined (highlighted below). The example below adds a NONCLUSTERED (Range) index to definition above. (Remember that NONCLUSTERED indexes are not available in CTP1.)

```
CREATE TABLE T2
(
  [Name] varchar(32) not null PRIMARY KEY NONCLUSTERED HASH WITH (BUCKET_COUNT = 1024),
  [City] varchar(32) null,
  [LastModified] datetime not null,

  INDEX T1_ndx_c2c3 NONCLUSTERED ([c2],[c3])

) WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

When a memory-optimized table is created, the In-Memory OLTP engine will generate and compile DML routines just for accessing that table, and load the routines as DLLs. SQL Server itself does not perform the actual data manipulation (record cracking) on memory-optimized tables, instead it calls the appropriate DLL for the required operation when a memory-optimized table is accessed.

There are only a few limitations when creating memory-optimized tables, in addition to the data type limitations already listed.

- No DML triggers
- No FOREIGN KEY or CHECK constraints
- No IDENTITY columns
- No UNIQUE indexes other than for the PRIMARY KEY
- A maximum of 8 indexes, including the index supporting the PRIMARY KEY

In addition, no schema changes are allowed once a table is created. Instead of using ALTER TABLE, you will need to drop and recreate the table. In addition, there are no specific index DDL commands (i.e. CREATE INDEX, ALTER INDEX, DROP INDEX). Indexes are always created at as part of the table creation.

Storage Overview

In-Memory OLTP memory-optimized tables and their indexes are stored very differently than disk-based tables.

Rows

Memory-optimized tables are not stored on pages like disk-based tables, nor is space allocated from extents, and this is due to the design principle of optimizing for byte-addressable memory instead of block-addressable disk. Rows are allocated from structures called heaps, which are different than the type of heaps SQL Server has supported for disk-based tables. Rows for a single table are not necessarily stored near other rows from the same table and the only way SQL Server knows what rows belong to the same table is because they are all connected using the tables' indexes. This is why memory-optimized tables have the requirement that there must be at least one index created on them. It is the index that provides structure for the tables.

The rows themselves have a structure very different than the row structures used for disk-based tables. Each row consists of a header and a payload containing the row attributes. Figure 2 shows this structure, as well as expanding on the content of the header area.

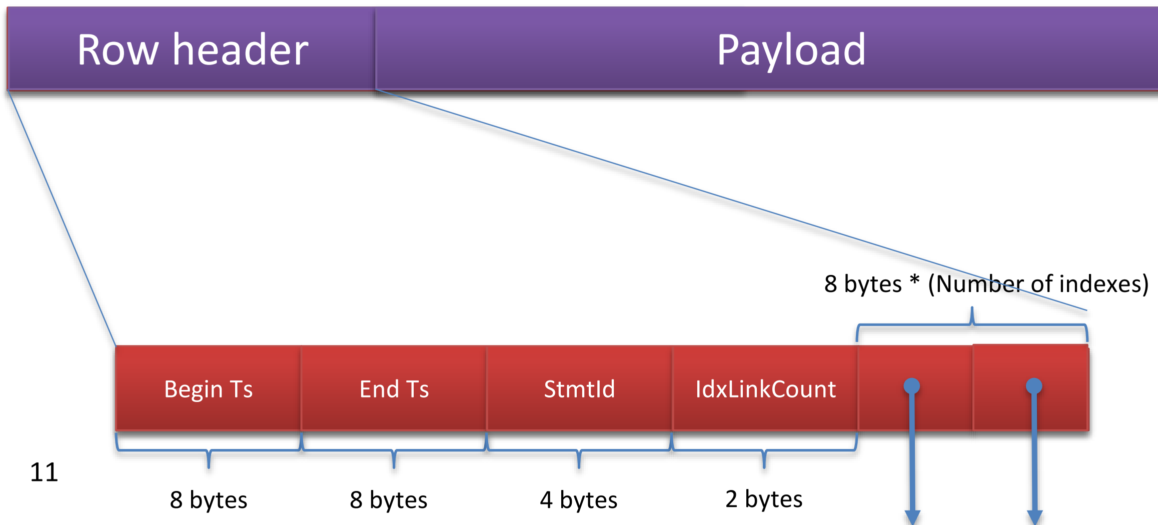

Figure 2 The structure of a row in a memory-optimized table

Row header

The header contains two 8-byte fields holding In-Memory OLTP timestamps. Every database capable of storing memory-optimized tables maintains an internal timestamp, or transaction counter, that is incremented every time a transaction referencing memory-optimized tables commits. The value of Begin-Ts is the timestamp of the transaction that inserted the row, and the End-Ts value is the timestamp for the transaction that deleted this row. A special value (referred to as 'infinity') is used as the End-Ts value for rows that have not been deleted. As we'll see when discussing data operations, the Begin-Ts and End-Ts values determine which other transactions will be able to see this row.

The header also contains a four-byte statement ID value. Every statement within a transaction has a unique *StmtId* value, and when a row is created it stores the *StmtId* for the statement that created the row. If the same row is then accessed again by the same statement, it can be skipped.

Finally, the header contains a two-byte value (*idxLinkCount*) which is really a reference count indicating how many indexes reference this row. Following the *idxLinkCount* value is a set of index pointers, which will be described in the next section. The number of pointers is equal to the number of indexes. The reference value of 1 that a row starts with is needed so the row can be referenced by the garbage collection (GC) mechanism even if the row is no longer connected to any indexes. The GC is considered the 'owner' of the initial reference, and it is the only process that can set the value to 0.

As mentioned, there is a pointer for each index on the table, and it is these pointers that connect the rows together. There are no other structures for combining rows into a table other than to link them together with the index pointers. This creates the requirement that all memory-optimized tables must have at least one index on them. Also, since the number of pointers is part of the row structure, and rows are never modified, all indexes must be defined at the time your memory-optimized table is created.

Payload area

The payload is the row itself, containing the key columns plus all the other columns in the row. (So this means that all indexes on a memory-optimized table are actually covering indexes.) The payload format can vary depending on the table. As mentioned earlier in the section on creating tables, the In-Memory OLTP compiler generates the DLLs for table operations, and as long as it knows the payload format used when inserting rows into a table, it can also generate the appropriate commands for all row operations.

Hash Indexes

A hash index consists of an array of pointers, and each element of the array is called a hash bucket. Up to 8 indexes can be declared for a memory-optimized table. The index key column in each row has a hash function applied to it, and the result of the function determines which bucket is used for that row. All key values that hash to the same value (have the same result from the hash function) are accessed from the same pointer in the hash index and are linked together in a chain. When a row is added to the table, the hash function is applied to the index key value in the row. If there is duplication of key values, the duplicates will always generate the same function result and thus will always be in the same chain.

Figure 3 shows one row in a hash index on a *name* column. For this example, assume there is a very simple hash function that results in a value equal to the length of the string in the index key column. The first value of 'Jane' will then hash to 4, which is the first bucket in the hash index so far. (Note that the real hash function is much more random and unpredictable, but I am using the length example to make it easier to illustrate.) You can see the pointer from the 4 entry in the hash table to the row with Jane. That row doesn't point to any other rows, so the index pointer is NULL.

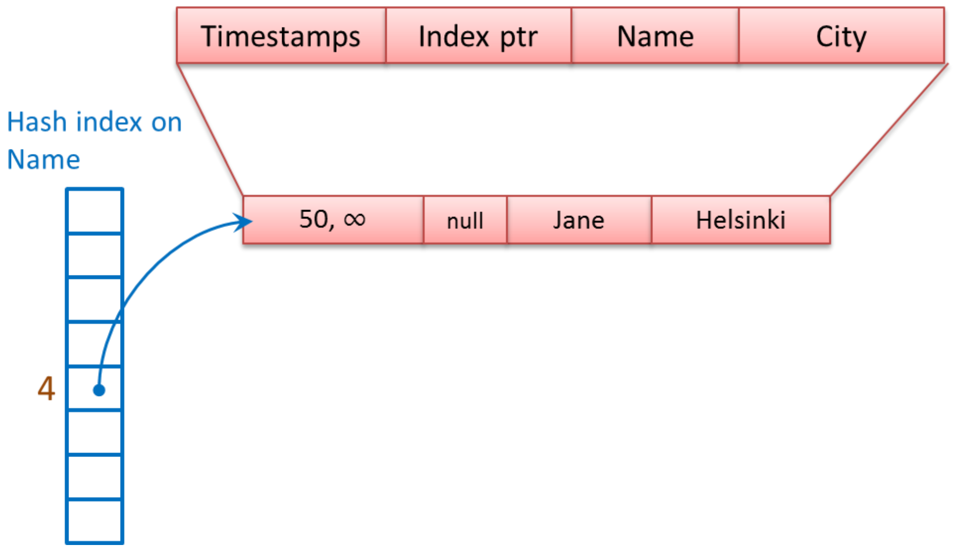

Figure 3 A hash index with a single row

In Figure 4, a row with a *name* value of Greg has been added to the table. Since we'll assume that Greg also maps to 4, it hashes to the same bucket as Jane, and the row is linked into the same chain as the row for Jane. The Greg row has a pointer to the Jane row.

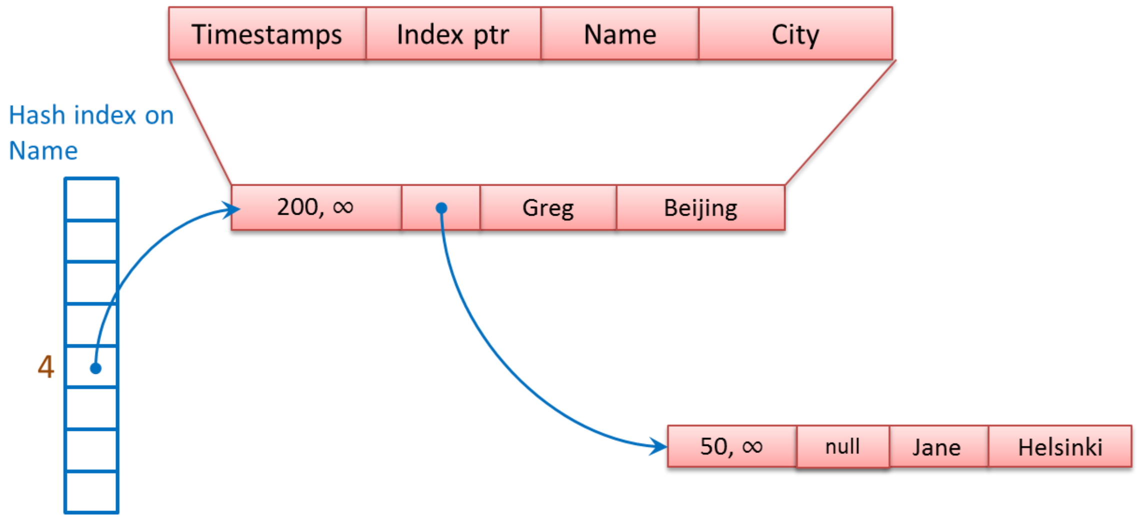

Figure 4 A hash index with two rows

Adding a second hash index to the table on the *City* column creates a second array of pointers. Each row in the table now has two pointers pointing to it, and the ability to point to two more rows, one for each index. The first pointer in each row points to the next value in the chain for the *Name* index; the second pointer points to the next value in the chain for the *City* index. Figure 5 shows the same hash index on *Name*, this time with three rows that hash to 4, and two rows that hash to 5, which uses the second bucket in the *Name* index. The second index on the *City* column uses three buckets. The bucket for 6 has three values in the chain, the bucket for 7 has one value in the chain, and the bucket for 8 also has one value.

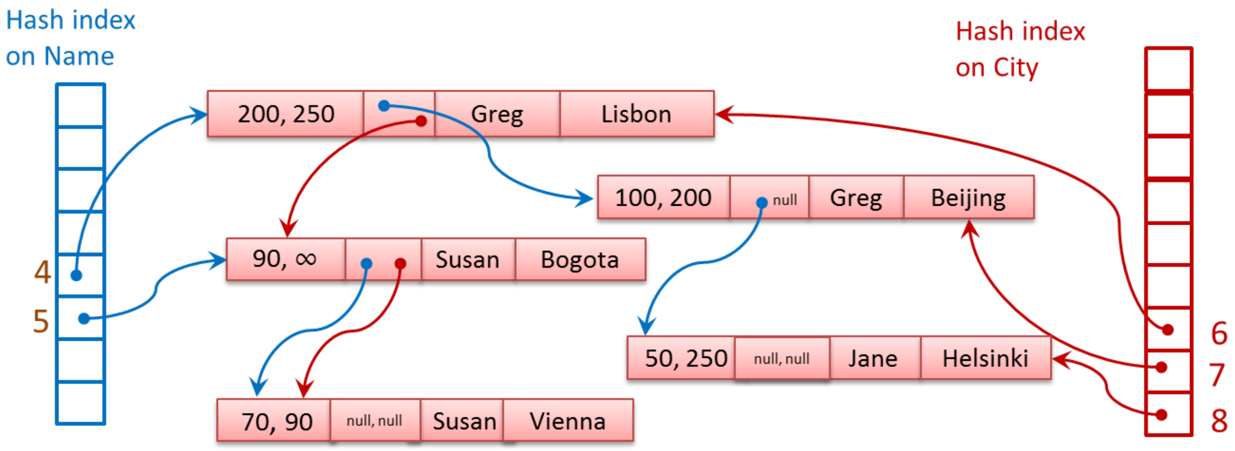

Figure 5 Two hash indexes on the same table

When a hash index is created, you must specify a number of buckets, as shown in the CREATE TABLE example above. It is recommended that you choose a number of buckets equal to or greater than the expected cardinality (the number of unique values) of the index key column so that there will be a greater likelihood that each bucket will only have rows with a single value in its chain. Be careful not to choose a number that is too big however, because each bucket uses memory. The number you supply is rounded up to the next power of two, so a value of 50,000 will be rounded up to 65,536. Having extra buckets will not improve performance but will simply waste memory and possible reduce the performance of scans which will have to check each bucket for rows.

Range Indexes

If you have no idea of the number of buckets you'll need for a particular column, or if you know you'll be searching your data based on a range of values, you should consider creating a range index instead of a hash index. Range indexes are not available in CTP 1, so range indexes will not be discussed further in this paper.

Data Operations

SQL Server In-Memory OLTP determines what row versions are visible to what transactions by maintaining an internal Transaction ID that serves the purpose of a timestamp, and will be referred to as a timestamp in this discussion. The timestamps are generated by a monotonically increasing counter which increases every time a transaction commits. A transaction's start time is the highest timestamp in the database at the time the transaction starts, and when the transaction commits, it generates a new timestamp which then uniquely identifies that transaction. Timestamps are used to specify the following:

- **Commit/End Time**: every transaction that modifies data commits at a distinct point in time called the commit or end timestamp of the transaction. The commit time effectively identifies a transaction's location in the serialization history.

- **Valid Time** for a version of a record: As shown in Figure 2, all records in the database contain two timestamps –the begin timestamp (Begin-Ts) and the end timestamp (End-Ts). The begin timestamp denotes the commit time of the transaction that created the version and the end timestamp denotes the commit timestamp of the transaction that deleted the version (and perhaps replaced it with a new version). The *valid time* for a record version denotes the range of timestamps where the version is visible to other transactions. In Figure 5, Susan's record is updated at time "90" from Vienna to Bogota as an example.

- **Logical Read Time**: the read time can be any value between the transaction's begin time and the current time. Only versions whose valid time overlaps the logical read time are visible to the read. For all isolation levels other than read-committed, the logical read time of a transaction corresponds to the start of the transaction. For read-committed it corresponds to the start of a statement within the transaction.

The notion of version visibility is fundamental to proper concurrency control in In-Memory OLTP. A transaction executing with logical read time RT must only see versions whose begin timestamp is less than RT and whose end timestamp is greater than RT.

Transaction Isolation Levels

The following isolation levels are supported for transactions accessing memory-optimized tables.

- SNAPSHOT
- REPEATABLE READ
- SERIALIZABLE

The transaction isolation level can be specified as part of the Atomic block of a natively compiled stored procedure. Alternatively, when accessing memory-optimized tables from interpreted Transact-SQL, the isolation level can be specified using table-level hints.

Specification of the transaction isolation level is required with natively compiled stored procedures. Specification of the isolation level in table hints is required when accessing memory-optimized tables from user transactions in interpreted Transact-SQL.

The isolation level READ COMMITTED is supported for memory optimized tables with autocommit transactions. It is not supported with explicit or implicit user transactions. Isolation level

READ_COMMITTED_SNAPSHOT is supported for memory-optimized tables with autocommit transaction and only if the query does not access any disk-based tables. In addition, transactions that are started using interpreted Transact-SQL with SNAPSHOT isolation cannot access memory-optimized tables. Transactions that are started using interpreted Transact-SQL with either REPEATABLE READ or SERIALIZABLE isolation must access memory-optimized tables using SNAPSHOT isolation.

Given the in-memory structures for rows described in Figures 3, 4 and 5, let's now look at how DML operations are performed by walking through an example. We will indicate rows by listing the contents in order, in angle brackets. Assume we have a transaction TX1 running at SERIALIZABLE isolation level that starts at timestamp 240 and performs two operations:

- DELETE the row <Greg , Lisbon>
- UPDATE <Jane, Helsinki> to <Jane, Perth>

Concurrently, two other transactions will read the rows. TX2 is an auto-commit, single statement SELECT that runs at timestamp 243. TX3 is an explicit transaction that reads a row and then updates another row based on the value it read in the SELECT; it has a timestamp of 246.

First we'll look at the data modification transaction. The transaction begins by obtaining a *begin timestamp* that indicates when it began relative to the serialization order of the database. In our example, that timestamp is 240.

While it is operating, transaction TX1 will only be able to access records that have a begin timestamp less than or equal to 240 and an end timestamp greater than 240.

Deleting

Transaction TX1 first locates <Greg, Lisbon> via one of the indexes. To delete the row, the end timestamp on the row is set to 240 with an extra flag bit indicating that transaction TX1 is active. Any other transaction that now attempts to access the row will need to determine if transaction TX1 is still active to determine if the deletion of <Greg , Lisbon> has been completed or not.

Updating and Inserting

Next the update of <Jane, Helsinki> is performed by breaking the operation into two separate operations: DELETE the entire original row, and then INSERT a complete new row. This begins by constructing the new row <Jane, Perth> with begin timestamp = 240 containing a flag bit indicating that Transaction TX1 is active, and then setting the end timestamp to ∞ (infinity). Any other transaction that attempts to access the row will need to determine if transaction TX1 is still active to decide whether it can see <Jane, Perth> or not. Any write-write conflict will cause the latter transaction to fail immediately. Next <Jane, Helsinki> is deleted just as described for the DELETE operation in the preceding paragraph. Then <Jane, Perth> is inserted by linking it into both indexes.

At this point transaction TX1 has completed its operations but not yet committed. Commit processing begins by obtaining an end timestamp for the transaction. This end timestamp, assume 250 for this example, identifies the point in the serialization order of the database where this transaction's updates have logically all occurred. In obtaining this end timestamp, the transaction enters a state called *validation* where it performs checks to ensure it that there are no violations of the current isolation level. If the validation fails, the transaction is aborted. More details about validation are covered shortly. SQL Server will also write to the transaction log during the validation phase.

Transactions track all of their changes in a *write set* that is basically a list of delete/insert operations with pointers to the version associated with each operation. The write set for this transaction, and the changed rows, are shown in the green box in Figure 6. This write set forms the content of the log for the transaction. Transactions may generate only a single log record that contains its timestamps and the versions of all records it deleted or inserted. There will not be separate log records for each row affected as there are for disk-based tables. However, there is an upper limit on the size of a log record, and if a transaction on memory-optimized tables needs exceeds the limit, there can be multiple log records generated. Once the log record has been hardened to storage the state of the transaction is changed to *committed* and post-processing is started.

Post-processing involves iterating over the write set and processing each entry as follows:

- For a DELETE operation, set the row's end timestamp to the end timestamp of the transaction (in this case 250) and clear the active flag on the row's end timestamp
- For an INSERT operation, set the affected row's begin timestamp to the end timestamp of the transaction (in this case 250) and clear the active flag on the row's begin timestamp

The actual unlinking and deletion of old row versions is handled by the garbage collection system, which will be discussed further in a later paper.

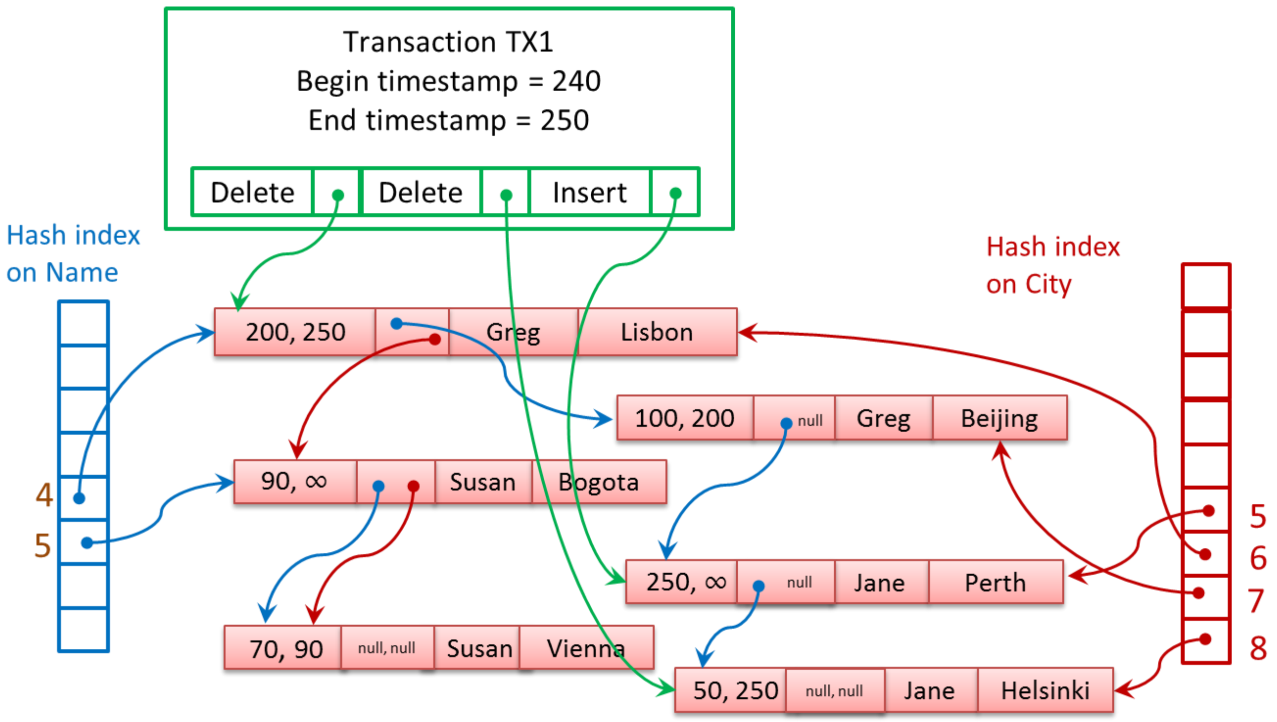

Figure 6 Transactional Modifications on a table

Reading

Now let's look at the read transactions, TX2 and TX3, which will be processed currently with TX1. Remember that TX1 is deleting the row <Greg , Lisbon> and updating <Jane, Helsinki> to <Jane, Perth> .

TX2 is an autocommit transaction that reads the entire table:

```
SELECT Name, City
FROM T1
```

TX2's session is running in the default isolation level READ COMMITTED, but as described above, because no hints are specified, and T1 is memory-optimized table, the data will be accessed using SNAPSHOT isolation. Because TX2 runs at timestamp 243, it will be able to read rows that existed at that time. It will not be able to access <Greg, Beijing> because that row no longer is valid at timestamp 243. The row <Greg, Lisbon> will be deleted as of timestamp 250, but it is value between timestamps 200 and 250, so transaction TX2 can read it. TX2 will also read the <Susan, Bogota> row and the <Jane, Helsinki> row.

TX3 is an explicit transaction that starts at timestamp 246. It will read one row and update another based on the value read.

```
DECLARE @City nvarchar(32);
BEGIN TRAN TX3
   SELECT @City = City
   FROM T1 WITH (REPEATABLEREAD)
   WHERE Name = 'Jane';

   UPDATE T1 WITH (REPEATABLEREAD)
   SET City = @City
   WHERE Name = 'Susan';
COMMIT TRAN  -- commits at timestamp 255
```

In TX3, the SELECT will read the row <Jane, Helsinki> because that row still is accessible as of timestamp 243. It will then update the <Susan, Bogota> row to <Susan, Helsinki>. However, if transaction TX3 tries to commit after TX1 has committed, SQL Server will detect that the <Jane, Helsinki> row has been updated by another transaction. This is a violation of the requested REPEATABLE READ isolation, so the commit will fail transaction TX3 will roll back. We'll see more about validation in the next section.

Validation

Prior to the final commit of transactions involving memory-optimized tables, SQL Server performs a validation step. Because no locks are acquired during data modifications, it is possible that the data changes could result in invalid data based on the requested isolation level. So this phase of the commit processing makes sure that there is no invalid data.

The following list shows you some of the possible violations that can be encountered in each of the possible isolation levels. More possible violations, as well as commit dependencies, will be discussed in the next paper when isolation levels and concurrency control will be described in greater detail.

If memory-optimized tables are accessed in SNAPSHOT isolation, the following validation errors are possible when a COMMIT is attempted:

- If the current transaction inserted a row with the same primary key value as a row that was inserted by another transaction that committed before the current transaction, error 41325 ("The current transaction failed to commit due to a repeatable read validation failure on table X.") will be generated and the transaction will be aborted.

If memory-optimized tables are accessed in REPEATABLE READ isolation, the following additional validation error is possible when a COMMIT is attempted:

- If the current transaction has read any row that was then updated by another transaction that committed before the current transaction, error 41305 ("The current transaction failed to commit due to a repeatable read validation failure on table X.") will be generated and the transaction will be aborted.

If memory-optimized tables are accessed in SERIALIZABLE isolation, the following additional validation errors are possible when a COMMIT is attempted:

- If the current transaction fails to read any valid rows or encounters phantoms that have occurred based on the filter condition of a SELECT, UPDATE or DELETE, the commit will fail. The transaction needs to be executed as if there are no concurrent transactions. All actions logically happen at a single serialization point. If any of these guarantees are violated, error 41325 is generated and the transaction will be aborted.

T-SQL Support

Memory-optimized tables can be accessed in two different ways: either through interop, using interpreted Transact-SQL, or through natively compiled stored procedures.

Interpreted T-SQL

When using the interop capability, you will have access to virtually the full T-SQL surface area when working with your memory-optimized tables, but you should not expect the same performance as when you access memory-optimized tables using natively compiled stored procedures. Interop is the appropriate choice when running adhoc queries, or to use while migrating your applications to In-Memory OLTP, as a step in the migration process, before migrating the most performance critical procedures. Interpreted T-SQL should also be used when you need to access both memory-optimized tables and disk-based tables.

The only T-SQL features not supported when accessing memory-optimized tables using interop are the following:

- TRUNCATE TABLE
- MERGE (when a memory-optimized table is the target)
- Dynamic and keyset cursors (these are automatically degraded to static cursors)
- Cross-database queries
- Cross-database transactions
- Linked servers
- Locking hints: TABLOCK, XLOCK, PAGLOCK, NOLOCK, etc.
- Isolation level hints READUNCOMMITTED, READCOMMITTED and READCOMMITTEDLOCK
- Memory-optimized table types and table variables are not supported in CTP1

T-SQL in Natively Compiled Procedures

Natively compiled stored procedures allow you to execute T-SQL in the fastest way, which includes accessing data in memory-optimized tables. There are however, many more limitations on the Transact-SQL that is allowed in these procedures. There are also limitations on the data types and collations that can be accessed and processed in natively compiled procedures. Please refer to the documentation for the full list of supported T-SQL statements, data types and operators that are allowed. In addition, disk-based tables are not allowed to be accessed at all inside natively compiled stored procedures.

The reason for the restrictions is due to the fact that internally, a separate function must be created for each operation on each table. The interface will be expanded in subsequent versions.

Features Not Supported

Many SQL Server features are supported for In-Memory OLTP and databases containing memory-optimized tables, but not all. For example, AlwaysOn, log shipping, and database backup and restore are fully supported. However, database mirroring and replication are not supported. You can use SQL Server Management Studio to work with memory-optimized tables and SSIS is also supported.

For the full list of supported and unsupported features, please refer to the SQL Server In-Memory OLTP documentation.

Metadata

Several existing metadata objects have been enhanced to provide information about memory-optimized tables and procedures.

There is one function that has been enhanced:

- *OBJECTPROPERTY* – now includes a property *TableIsMemoryOptimized*

The following system views have been enhanced:

- *sys.data_spaces* – now has a *type* value of FX and a *type_desc* value of MEMORY_OPTIMIZED_DATA_FILEGROUP
- *sys.tables* – has three new columns:
  - *durability* (0 or 1)
  - *durability_desc* (SCHEMA_AND_DATA and SCHEMA_ONLY)
  - *is_memory_optimized* (0 or 1)
- *sys.table_types* – now has a column *is_memory_optimized*
- *sys.indexes* – now has a possible *type* value of 7 and a corresponding *type_desc* value of NONCLUSTERED HASH
- *sys.data_spaces* -- now has a possible *type* value of FX and a corresponding *type_desc* value of MEMORY_OPTIMIZED_DATA_FILEGROUP
- *sys.sql_modules* and *sys.all_sql_modules* – now contain a column *uses_native_compilation*

In addition, there are several new metadata objects that provide information specifically for memory-optimized tables.

A new catalog view has been added to CTP1 to support hash indexes: *sys.hash_indexes*. This view is based on *sys.indexes* so has the same columns as that view, with one extra column added. The *bucket_count* column shows a count of the number of hash buckets specified for the index and the value cannot be changed without dropping and recreating the index.

The following SQL Server Dynamic Management Views are new for In-Memory OLTP. The ones that start with *sys.dm_db_xtp_** give information about individual In-Memory OLTP -enabled databases, where the ones that start with *sys.dm_xtp_** provide instance-wide information. You can read about the details of these objects in the documentation. A subsequent whitepaper will go into more detail on the DMVs relevant to the topics we will be discussing in that paper, for example, checking and checkpoint files, transactions and garbage collection.

- sys.dm_db_xtp_checkpoint
- sys.dm_db_xtp_checkpoint_files
- sys.dm_xtp_consumer_memory_usage
- sys.dm_db_xtp_gc_cycles_stats
- sys.dm_xtp_gc_stats
- sys.dm_xtp_memory_stats
- sys.dm_xtp_system_memory_consumers
- sys.dm_xtp_threads
- sys.dm_xtp_transaction_stats
- sys.dm_db_xtp_index_stats
- sys.dm_db_xtp_memory_consumers
- sys.dm_db_xtp_object_stats
- sys.dm_db_xtp_transactions
- sys.dm_db_xtp_table_memory_stats

Using AMR for Best Practice Recommendations

After installing SQL Server In-Memory OLTP, the AMR (Analysis, Migration and Reporting) tool can provide recommendations as to what tables and procedures you might want to consider migrating to In-Memory OLTP.

This tool is built on management Data Warehouse using data collector, and comes in the form of new report types, available when right-clicking on the MDW database, and choosing Reports | Management Data Warehouse. You will then have the option to choose Transaction Performance Analysis Overview.

One of the reports generated will contain recommendations on which tables might provide the biggest performance gain when converted to memory-optimized tables. The report will also describe how much effort would be required to carry out the conversion based on how many unsupported features the table concurrently uses. For example, it will point out unsupported data types and constraints used in the table definition.

Another report will contain recommendations on which procedures might benefit from being converted to natively compiled procedures for use with memory-optimized tables.

The fact that In-Memory OLTP memory-optimized tables can be used with interpreted TSQL as well as with natively compiled stored procedures, and that In-Memory OLTP memory-optimized tables can be used in the same queries as disk-based tables means that migrating to an In-Memory OLTP environment can be done gradually and iteratively. Based on recommendations from the Management Data Warehouse reports, you can start converting tables to be memory-optimized tables one at a time, starting with ones that would benefit most from the memory optimized storage. As you start seeing the benefit of the conversion to memory-optimized tables, you can continue to convert more of more of your tables, but access them using your normal Transact-SQL interface, with very few application changes.

Once your tables have been converted, , you can then start planning a rewrite of the code into natively compiled stored procedures, again starting with the ones that the Management Data Warehouse reports indicate would provide the most benefit.

Summary

SQL Server In-Memory OLTP provides the ability to create and work with tables that are memory-optimized and extremely efficient to manage, providing performance optimization for OLTP workloads. They are accessed with true multi-version optimistic concurrency control requiring no locks or latches during processing. All In-Memory OLTP memory-optimized tables must have at least one index, and all access is via indexes. In-Memory OLTP memory-optimized tables can be referenced in the same transactions as disk-based tables, with only a few restrictions. No access of disk-based tables is allowed inside natively compiled stored procedures, which is the fastest way to access your memory-optimized tables and perform business logic and computations.

For more information:

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

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 due to having good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

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

As mentioned in this paper, there will be a follow-up whitepaper available for In-Memory OLTP CTP2, containing more technical details on the following topics;

1. Range indexes
2. Concurrency management
3. Checkpoints and recovery

If you have specific questions in these areas, or any of the areas discussed in the current paper, that you would like to see addressed in the subsequent paper, please submit them through the feedback link.

Send feedback.