SQL Server In-Memory OLTP Internals for SQL Server 2016

Technical White Paper

Writer: Kalen Delaney

Technical Reviewers: Sunil Agarwal and Jos de Bruijn

Published: June 2016

Applies to: SQL Server 2016 RTM

Summary:

In-memory OLTP, frequently referred to by its codename "Hekaton", was introduced in SQL Server 2014. This powerful technology allows you to take advantage of large amounts of memory and many dozens of cores to increase performance for OLTP operations by up to 30 to 40 times! SQL Server 2016 is continuing the investment in In-memory OLTP by removing many of the limitations found in SQL Server 2014, and enhancing internal processing algorithms so that In-memory OLTP can provide even greater improvements. This paper describes the implementation of SQL Server 2016's In-memory OLTP technology as of SQL Server 2016 RTM. Using In-memory OLTP, tables can be declared as 'memory optimized' to enable In-Memory OLTP's capabilities. Memory-optimized tables are fully transactional and can be accessed using Transact-SQL. Transact-SQL stored procedures, triggers and scalar UDFs can be compiled to machine code for further performance improvements on memory-optimized tables. The engine is designed for high concurrency with no blocking.

Copyright

1.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 This assumption is no longer valid as memory prices have dropped enormously over the last 30 years.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 describes the technical implementation of the In-memory OLTP database engine feature.

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 due to writes to disk needed by traditional database systems). Other features of In-Memory OLTP 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 reason for creating In-Memory OLTP comes mainly from this fact that main memory sizes are growing at a rapid rate and becoming less expensive. 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 and 5TB with working sets that are significantly smaller. As of Q2 2016, even a four socket server could hold 3TB of DRAM using 32GB SIMMS and 6TB of DRAM using 64GB DIMMS. Looking further ahead, it's entirely possible that in a few years you'll be able to build distributed DRAM based systems with capacities of 1-10 Petabytes at a cost of less than $5/GB. It is also only a question of time before non-volatile RAM becomes viable, as it already is in development in various form factors.

If most or all of an application's data is able to be entirely memory resident, the costing rules, particularly for estimating I/O costs, that the SQL Server optimizer has used since the very first version become almost completely obsolete, because the rules assume all pages accessed can potentially require a physical read from disk. If there is no need to ever read from disk, the optimizer doesn't need to consider I/O cost at all. 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 become disproportionately large. In-Memory OLTP addresses all these issues. In-Memory OLTP removes the issues of waiting for locks to be released, using a new type of multi-version optimistic concurrency control. It reduces the delays of waiting for log writes by generating far less log data and needing fewer log writes.

Terminology

SQL Server 2016's *In-Memory OLTP* feature refers to a suite of technologies for working with memory-optimized tables. The alternative to memory-optimized tables will be referred to as *disk-based tables*, which SQL Server has always provided. 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 modules* refer to object types supported by In-Memory OLTP that can be compiled to machine code and have the potential to increase performance even further than just using memory-optimized tables. Supported object types are stored procedures, triggers scalar user-defined functions and inline multi-statement user-defined functions. The alternative is *interpreted Transact-SQL* modules, which is what SQL Server has always used. Natively compiled modules 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 Transact-SQL that references memory-optimized tables

What's Special About In-Memory OLTP?

In-Memory OLTP is integrated with the SQL Server relational engine, and can be accessed transparently using the same interfaces. In fact, users may be unaware that they are working with memory-optimized tables rather than disk-based tables. However, the internal behavior and capabilities of In-memory OLTP are very different. Figure 1 gives an overview of the SQL Server engine with the In-Memory OLTP components.

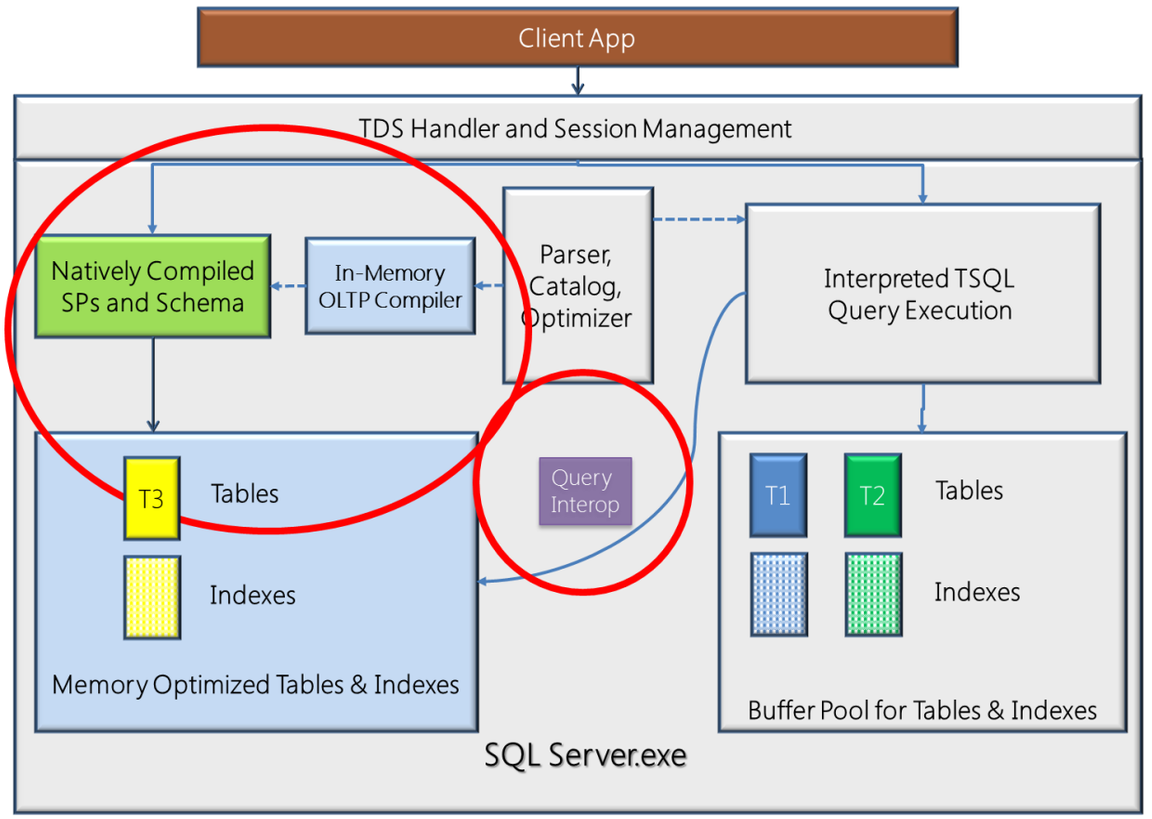

Figure 1 The SQL Server engine including the In-Memory OLTP component

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

Memory-optimized tables

One of the most important differences 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. Memory-optimized tables can be either durable or non-durable. The default is for these tables to be durable, and these durable tables also meet all the other transactional requirements; they are atomic, isolated, and consistent. A set of checkpoint files (data and delta file pairs), which are only used for recovery purposes, is created using operating system files residing in memory-optimized filegroups that keep track of the changes to the data in the durable tables. These 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, and nothing is written to the checkpoint files for these tables. The data is only available in memory while SQL Server is running. In the event of a SQL Server shutdown or an AlwaysOn Availabilty Group failover, the data in these tables is lost. The schema will be recreated when the database they belong to is recovered, but there will be no data. These tables could be useful, for example, as staging tables in ETL scenarios or for storing Web server session state. Although the data is not durable, operations on these tables meet all the other transactional requirements. (That is, they are atomic, isolated, and consistent.) We'll see the syntax for creating both durable and non-durable tables in the section on Creating Tables.

> **Note:** Non-durable memory-optimized table may seem similar to global temporary tables (indicated with ## at the beginning of the name.) and can sometimes be used for similar purposes. However, global temp tables are regular disk-based tables, stored on pages in the *tempdb* database, and read from and written to disk as needed. The only thing special about the global temp tables is that they are dropped when no longer needed. Also, like any other object in *tempdb*, neither the schema nor the data is recovered when SQL Server is restarted. A non-durable memory-optimized table can be a part of any database that allows memory-optimized tables, but its data only resides in memory. Memory-optimized tables also use completely different structures for keeping track of the data than are used for disk-based tables, whether temporary or permanent.

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 for memory-optimized tables are stored using special Bw-trees. The range index with Bw-tree can be used to quickly find qualifying rows in a range predicate just like traditional a B-tree but it is designed with optimistic concurrency control with no locking or latching.

Every memory-optimized table must have at least one index, and if it is a durable table (specified with the option SCHEMA_AND_DATA) it must have a declared primary key, which could then be supported by the required index.

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 rebuilt as the data is streamed into memory.

Concurrency improvements

When accessing memory-optimized tables, SQL Server implements an 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. In addition to being lock-free, In-memory OLTP does not use any latches or even spinlocks. Any synchronization between threads

that is absolutely required to maintain consistency (for example, when updating an index pointer) is handled at the lowest level: a single CPU instruction (interlocked-compare and exchange), which is supported by all modern 64-bit processors. (For more details about this instruction, please take a look at this article: https://en.wikipedia.org/wiki/Compare-and-swap).

Note that this does not mean that there is no possibility of waiting when using memory-optimized tables. There may be other wait types encountered, such as waiting for a log write to complete at the end of a transaction, which can happen for writes to disk-based tables as well. 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. (Note that waiting for log writes to complete can be ameliorated using SQL Server's "Delayed Durability" option, which you can read about here: https://msdn.microsoft.com/library/dn449490.aspx)

Natively Compiled Modules

The best execution performance is obtained when using natively compiled modules with memory-optimized tables. As of SQL Server 2016, these modules include stored procedures, triggers and user-defined scalar functions. Inline table valued functions can also be declared with native compilation, and although they are not in themselves actually compiled, they can then be used in a natively compiled module. In addition, there are only a few limitations on the Transact-SQL language constructs that are allowed inside a natively compiled stored module, compared to the feature set available with interpreted Transact-SQL.

Performance

The special data structures for rows and indexes, the elimination of locks and latches, and the ability to create natively compiled stored procedures allow for incredible performance when working with memory-optimized tables. In the Microsoft lab, one particular customer achieved 1 million batch requests/sec, with 4KB of data read and written with each batch, without maxing out the CPU.

Although that workload used SCHEMA_ONLY tables, durable tables can also get impressive results. Lab results repeatedly showed a sustained ingestion of ingestion of 10M rows/second, with an average of 100 bytes per row. A representative order processing workload showed 260K transactions per second, with 1GB/sec of log generation.

Both the SCHEMA_ONLY and SCHEMA_AND_DATA tables were created on 4-socket servers with a total of 72 physical cores.

Using In-Memory OLTP

The In-Memory OLTP engine was introduced in SQL Server 2014. Installation of In-Memory OLTP is part of the SQL Server setup application, as it is just a part of the database engine service. The In-Memory OLTP components can only be installed with a 64-bit edition of SQL Server, and not available at all with a 32-bit edition.

Creating Databases

Any database that will contain memory-optimized tables needs to have one MEMORY_OPTIMIZED_DATA filegroup. This filegroup is used for storing the data and delta file pairs 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:

```
USE Master;
GO
CREATE DATABASE IMDB
    ON
    PRIMARY  (NAME = IMDB_data, FILENAME = 'C:\HKData\IMDB_data.mdf'),
    FILEGROUP IMDB_mod_FG CONTAINS MEMORY_OPTIMIZED_DATA
                      (NAME = IMDB_mod, FILENAME = 'C:\HKData\IMDB_mod');
```

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, if you already have the *AdventureWorks2016* database, you can add a filegroup for memory optimized data as shown:

```
ALTER DATABASE AdventureWorks2016
            ADD FILEGROUP AW_mod_FG CONTAINS MEMORY_OPTIMIZED_DATA;
GO
ALTER DATABASE AdventureWorks2016
            ADD FILE (NAME='AW_mod', FILENAME='C:\HKData\AW_mod')
    TO FILEGROUP AW_mod_FG;
GO
```

Note that although the syntax indicates that we are specifying a file, using the FILENAME argument, there is no file suffix specified because we are actually specifying a path for a folder that will contain the checkpoint files. We will discuss the checkpoint files in detail in a later section. Many of my examples will use *IMDB* (or some variation thereof) as database name, to indicated it is an '**i**n-**m**emory **d**ata**b**ase'). Also, many of my filegroups for storing the memory-optimized data will use the suffix MOD, for **m**emory-**o**ptimized **d**ata. Once a database has a filegroup with the property CONTAINS MEMORY_OPTIMIZED_DATA, and that filegroup has at least one file in it, you can create memory-optimized tables and other in-memory OLTP objects in that database. The following query will show you the names of all the databases in an instance of SQL Server that meets those requirements:

```
EXEC sp_MSforeachdb 'USE ? IF EXISTS (SELECT 1 FROM sys.filegroups FG
               JOIN sys.database_files F
                           ON FG.data_space_id = F.data_space_id
     WHERE FG.type = ''FX'' AND F.type = 2)
           PRINT ''?'' + '' can contain memory-optimized tables.'' ';
GO
```

With SQL Server 2016, the 'filestream' filegroup is only used as a container and the file-management within the container is done by in-memory OLTP engine. This allows for Transparent Data Encryption as well as for faster garbage collection of unneeded files. Note, this is different compared to SQL Server 2014, in which the files in the 'filestream' filegroup are managed by SQL Server's FILESTREAM mechanism.

Most of the properties that can be enabled for any SQL Server 2016 database can be enabled for a database that supports memory-optimized tables. As of SQL Server 2016, if you the enable Transparent Data Encryption (TDE) for a database, the checkpoint files in the filegroup with the CONTAINS MEMORY_OPTIMIZED_DATA property will also be encrypted using the same encryption key as your other files (.mdf, .ndf and .ldf) in the database.

Azure SQL Database

Memory-optimized tables can be created in an Azure SQL Database using a PREMIUM tier. All such databases are already capable of supporting memory-optimized tables, so you have no worry about provisioning. You do not need to specific a filegroup with the CONTAINS MEMORY_OPTIMIZED_DATA property, and you don't need to bind the database to a resource pool. In fact, for Azure SQL Database, you can't do these things!

One issue to be aware of however, occurs if you have created SCHEMA_ONLY tables. As will be discussed in the next section, the data in these tables is non-durable and will be lost on a system failure or shutdown. If the failure happens in the middle of a transaction, your application may be able to determine that the transaction was not committed, so the data does not exist, but if the failure occurs outside of a transaction, Azure SQL Database will not notify you of the failure. It will automatically recover, but any non-durable memory-optimized tables will then be completely empty.

Additional Database Requirements and Recommendations

Another subtler database requirement is that the processor needs to support the instruction **cmpxchg16b**, which all modern 64-bit processors support, but is **not** supported by all virtual machine hosts. For example, VirtualBox does not support this instruction by default.

If you are using a VM host application and SQL Server displays an error caused by an older processor, you can check to see if the application has a configuration option to allow **cmpxchg16b**. If not, consider using Hyper-V as your virtual machine host, which supports **cmpxchg16b** without needing to modify a configuration option. It is also recommended that you verify that any database in which you plan to create memory-optimized tables is using compatibility level 130. This will allow certain new SQL Server 2016 capabilities such a s automatic update of statistics and parallel plans. These capabilities will be discussed later in this document. Finally consider setting the database option MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT to ON. Details can be found in the section on isolation levels and in-memory OLTP. You might want to take a look at this useful script which will verify that your SQL Server and database have all the appropriate settings for working with memory-optimized tables and objects.

https://github.com/Microsoft/sql-server-samples/blob/master/samples/features/in-memory/t-sql-scripts/enable-in-memory-oltp.sql

This script verifies both SQL Server databases and Azure SQL DB. It ensures a memory optimized filegroup is created, and creates one for you in the default data directly if one does not yet exist. It then sets all the recommended settings.

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
- String types: `char(n)`, `varchar(n)`, `nchar(n)`, `nvarchar(n)`, `sysname`, `varchar(MAX)`, `nvarchar(MAX)`
- Binary types: `binary(n)`, `varbinary(n)`, `varbinary(MAX)`
- `Uniqueidentifier`
-

- Note that the legacy LOB data types (*text*, *ntext* and *image*) are not allowed; also there can be no columns of type *XML* or *CLR*. As of SQL Server 2016, the MAX datatypes are allowed, and these can be referred to as LOB columns. In addition, row lengths can exceed 8060 bytes using only regular variable length columns, with the largest variable length columns stored off-row, similar to row-overflow data for disk-based tables. The decision as to which columns will be stored off-row is made at table creation time based on the maximum allowable length, not on the actual length of any data values inserted. Note that if the table has LOB or off-row columns, you cannot build a columnstore index on it.

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. If a memory-optimized table defined with DURABILITY = SCHEMA_ONLY, then 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. 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 value for the bucket count will be mentioned when discussing details of hash index storage. Note that the index supporting the primary key is the only index on a memory-optimized table that can be unique. The UNIQUE keyword is not supported for indexes on memory-optimized tables.

Single-column indexes may be created in line with the column definition in the CREATE TABLE statement, as shown below.

```
CREATE TABLE dbo.people
(
  [Name] varchar(32) not null PRIMARY KEY NONCLUSTERED,
  [City] varchar(32) null,
  [State_Province] 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, as in the example below. This example adds a range index to definition above. Notice the difference in the specification for the two types of indexes is that one uses the keyword HASH, and the other doesn't. Both types of indexes are specified as NONCLUSTERED, but if the word HASH is not used, the index is a range index.

```
CREATE TABLE dbo.people2
(
  [Name] varchar(32) not null PRIMARY KEY NONCLUSTERED,
  [City] varchar(32) null,
  [State_Province] varchar(32) null,
  [LastModified] datetime not null,

  INDEX T1_ndx_c2c3 NONCLUSTERED ([City],[State_Province])

) 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.

For SQL Server 2016, there are only a few limitations when creating memory-optimized tables, in addition to the data type limitations already listed. Some of these limitations are likely to be removed in Azure SQL DB and in a later release of SQL Server.

To see the changes in SQL Server 2016, take a look at this page in the documentation: https://msdn.microsoft.com/library/bb510411.aspx.

- DML triggers must be created as natively compiled modules
- FOREIGN KEY constraints must reference a PRIMARY KEY constraint (not a UNIQUE constraint) and the referenced table must also be a memory-optimized table.
- IDENTITY columns can only be defined with SEED and INCREMENT of 1
- A maximum of 8 indexes can be created on a single table, including the index supporting the PRIMARY KEY

> **Note** Prior to SQL Server 2016, for an index to be created on a character column in a memory-optimized table, the column would have to have been created with a binary (BIN) collation. The collation could have been specified in the CREATE TABLE statement, or the database could have been created to use a BIN collation for all character columns. As of SQL Server 2016, this restriction no longer applies. The CREATE TABLE statements above create indexes on the *Name* column which does not specify a BIN collation.

The catalog view *sys.tables* contains metadata about each of your memory-optimized tables. The column *is_memory_optimized* is a Boolean value, with a 1 indicating a memory-optimized table. The *durability* column is translated by the *durability_desc* column, which has values SCHEMA_ONLY and

SCHEMA_AND_DATA. The following query will show you your memory-optimized tables and their durability:

```
SELECT name, durability_desc FROM sys.tables
WHERE is_memory_optimized = 1;
```

Altering Tables and Indexes

As of SQL Server 2016, ALTER TABLE can be used to add and drop columns, indexes, and constraints as well as modify column definitions and change the number of hash buckets in a hash index. Each ALTER TABLE will require a complete rebuild of the table, even if you're just changing the number of buckets in a hash index, so you'll need to make sure you have sufficient memory available before running this command. As the table is being rebuilt, each row will be reinserted into the new table and the table will be unavailable while the ALTER operation is being performed.

If multiple changes need to be made to a single table, changes of same type can be combined into a single ALTER TABLE command. For example, you can ADD multiple columns, indexes and constraints in a single ALTER TABLE and you can DROP multiple columns, indexes and constraints in a single ALTER TABLE, but you cannot have an ADD and a DROP in the same ALTER TABLE command. It is recommended that you combine changes wherever possible, so that you can keep the number of table rebuilds to a minimum.

Most ALTER TABLE operations can be executed in parallel by the SQL Server engine, and the operation is log-optimized, which means that only metadata changes are written to the transaction log. However, there are some ALTER TABLE operations that can only run single-threaded and will write every row to the log as the ALTER TABLE is being performed. The following operations are the ones that run single-threaded and will log every row:

- ADD/ALTER a column to use a LOB type: nvarchar(max), varchar(max), or varbinary(max).
- ADD/DROP a COLUMNSTORE index.
- ADD/ALTER an off-row column and ADD/ALTER/DROP operations that cause an in-row column to be moved off-row, or an off-row column to be moved in-row. (Note that ALTER operations that increase the length of a column that is already stored off-row are log-optimized.)

Here are some code examples illustrating the SQL Server 2016 ALTER TABLE operations on a memory-optimized table:

```
USE IMDB
GO

DROP TABLE IF EXISTS dbo.OrderDetails;
GO

-- create a simple table
CREATE TABLE dbo.OrderDetails
(
  OrderID int NOT NULL,
  ProductID int NOT NULL,
  UnitPrice money NOT NULL,
  Quantity smallint NOT NULL,
  Discount real NULL

  INDEX IX_OrderID NONCLUSTERED HASH (OrderID) WITH ( BUCKET_COUNT = 1048576),
  INDEX IX_ProductID NONCLUSTERED HASH (ProductID) WITH ( BUCKET_COUNT = 131072),
  CONSTRAINT PK_Order_Details PRIMARY KEY
            NONCLUSTERED HASH (OrderID,  ProductID) WITH ( BUCKET_COUNT = 1048576)
```

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

GO

-- index operations

-- change hash index bucket count
ALTER TABLE dbo.OrderDetails
  ALTER INDEX IX_OrderID
            REBUILD WITH (BUCKET_COUNT=2097152);
GO

-- add index
ALTER TABLE dbo.OrderDetails
  ADD INDEX IX_UnitPrice NONCLUSTERED (UnitPrice);
GO

-- drop index
ALTER TABLE dbo.OrderDetails
  DROP INDEX IX_UnitPrice;
GO

-- add multiple indexes
ALTER TABLE dbo.OrderDetails
  ADD INDEX IX_UnitPrice NONCLUSTERED (UnitPrice),
   INDEX IX_Discount NONCLUSTERED (Quantity);
GO

-- Add a new column and an index

ALTER TABLE dbo.OrderDetails
  ADD ModifiedDate datetime,
      INDEX IX_ModifiedDate NONCLUSTERED (ModifiedDate);
GO

-- Drop a column
ALTER TABLE dbo.OrderDetails
  DROP COLUMN Discount;
GO
```

Table Types and Table Variables

In addition to creating memory-optimized tables, SQL Server 2016 allows you to create memory-optimized table types. The biggest benefit of a memory-optimized table type is the ability to use it when declaring a table variable as shown here:

```
USE IMDB;
GO
CREATE TYPE SalesOrderDetailType_inmem
AS TABLE
(
  OrderQty smallint NOT NULL,
  ProductID int NOT NULL,
  SpecialOfferID int NOT NULL,
  LocalID int NOT NULL,

  INDEX IX_ProductID NONCLUSTERED HASH (ProductID) WITH (BUCKET_COUNT = 131072),
  INDEX IX_SpecialOfferID NONCLUSTERED (SpecialOfferID)
)
WITH (MEMORY_OPTIMIZED = ON );
GO
DECLARE @SalesDetail SalesOrderDetailType_inmem;
GO
```

Memory-optimized table variables can give you the following advantages when compared to disk-based table variables:

- The variables are only stored in memory. Data access is more efficient because tables defined using memory-optimized table types use the same data structures used for memory-optimized tables. The efficiency is increased further if a memory-optimized table variable is used in a natively compiled module.

- Table variables are not stored in *tempdb* and do not use any resources in *tempdb*.
- Memory-optimized table variables avoid contention on the database PFS and SGAM pages, which happens frequently in high-throughput workloads that use table-valued parameters.

You might want to take a look a following blog post, which explains how you have can start leveraging memory-optimized table variables in your application with very minimal changes:

https://blogs.msdn.microsoft.com/sqlserverstorageengine/2016/03/21/improving-temp-table-and-table-variable-performance-using-memory-optimization/

Row and Index Storage

In-Memory OLTP memory-optimized tables and their indexes are stored very differently than disk-based tables. Memory-optimized tables are not stored on pages like disk-based tables, nor is space allocated from extents. This is due to the design principle of optimizing for byte-addressable memory instead of block-addressable disk.

Memory Consumers

For SQL Server 2016, each table has space allocated from its own memory heap, called a varheap, which is a type of memory consumer. Varheaps are variable size heap data structures, where 'variable' means that they can grow and shrink. They are actually implemented as collections of fix-sized allocations from pages of various sizes, with 64K being the most common. For example, there can be a page dedicated to 32-byte memory allocations and another page used just for 128-byte allocations. Some pages could be used for 8192-byte allocations and when a certain number of bytes needs to be allocated for a memory-optimized table, it is taken from a page dedicated to sizes equal to or greater than the needed number of bytes. So a request for 100 bytes would use 128-byte allocation from a page dedicated to allocations of that size.

The varheaps are used for both table rows and Bw-tree indexes. (Hash indexes are the only structure used with memory-optimized tables that uses a different memory consumer.) In addition, each LOB, column (specified with the MAX qualifier in the datatype definition) has its own varheap. As mentioned earlier, SQL Server 2016 also supports large columns similar to the row-overflow columns for disk-based tables. For memory-optimized tables, SQL Server will decide when the table is created which of your variable length columns will be stored in the table's varheap and which will be stored as overflow data and have their own varheap. You can think of LOB and row-overflow columns as being stored in their own internal tables.

You can examine the metadata for each varheap (and the other memory consumers) in a DMV called *sys.dm_db_xtp_memory_consumers*. Each memory-optimized table has a row in this view for each varheap and for each hash index. (We will see one more type of memory consumer, called HkCS Allocator, in the section on columnstore indexes on memory-optimized tables.) If we join this DMV with the catalog view called *sys.memory_optimized_tables_internal_attributes* we can see which varheap belongs to a specific column. Each user table has a unique *object_id* and *xtp_object_id* value which is used by all the indexes. Each additional varheap, for the row_overflow and LOB columns will have its own *xtp_object_id*. Note that if an object is altered, its *xtp_object_id* will change, but its *object_id* will not. The code below creates a

table with both hash and range indexes which has both types of large columns. The column in the table called *description_medium* (column 7) will be row_overflow and *description_full* (column 8) will be LOB. The code then SELECTs from a join of three views to show the memory consumer information.

```
USE IMDB;
GO
DROP TABLE IF EXISTS dbo.OrderDetails;
GO
CREATE TABLE dbo.OrderDetails
(
  OrderID int NOT NULL,
  ProductID int NOT NULL,
  UnitPrice money NOT NULL,
  Quantity smallint NOT NULL,
  Discount real NOT NULL,
  Description varchar(2000),
  Description_medium varchar(8000),
  Description_full varchar(max),

  INDEX IX_OrderID NONCLUSTERED (OrderID),
  INDEX IX_ProductID NONCLUSTERED (     ProductID),
  CONSTRAINT PK_Order_Details PRIMARY KEY
        NONCLUSTERED HASH (OrderID,      ProductID)
        WITH ( BUCKET_COUNT = 1048576)
) WITH ( MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA )

GO

SELECT object_name(c.object_id) AS table_name,
         a.xtp_object_id, a.type_desc,
            minor_id, memory_consumer_id as consumer_id,
            memory_consumer_type_desc as consumer_type_desc,
            memory_consumer_desc as consumer_desc,
            allocated_bytes as bytes
  FROM sys.memory_optimized_tables_internal_attributes a
    JOIN sys.dm_db_xtp_memory_consumers c
     ON a.object_id = c.object_id and a.xtp_object_id = c.xtp_object_id
       LEFT JOIN sys.indexes i
              ON c.object_id = i.object_id
        AND c.index_id = i.index_id
  WHERE c.object_id = object_id('dbo.OrderDetails');
GO
```

The SELECT gives me the results shown in Figure 2.

table_name	xtp_object_id	type_desc	minor_id	consumer_id	index_id	consumer_type_desc	consumer_desc	bytes
OrderDetails	-2147483615	USER_TABLE	0	219	4	VARHEAP	Range index heap	131072
OrderDetails	-2147483615	USER_TABLE	0	218	3	VARHEAP	Range index heap	131072
OrderDetails	-2147483615	USER_TABLE	0	217	2	HASH	Hash index	8388608
OrderDetails	-2147483615	USER_TABLE	0	216	NULL	VARHEAP	Table heap	0
OrderDetails	-2147483613	INTERNAL OFF-ROW DATA TABLE	8	215	2	VARHEAP	Range index heap	131072
OrderDetails	-2147483613	INTERNAL OFF-ROW DATA TABLE	8	214	NULL	VARHEAP	LOB Page Allocator	0
OrderDetails	-2147483613	INTERNAL OFF-ROW DATA TABLE	8	213	NULL	VARHEAP	Table heap	0
OrderDetails	-2147483614	INTERNAL OFF-ROW DATA TABLE	7	212	2	VARHEAP	Range index heap	131072
OrderDetails	-2147483614	INTERNAL OFF-ROW DATA TABLE	7	211	NULL	VARHEAP	Table heap	0

Figure 2: Various memory consumers after creating a memory-optimized table

In this output, the *minor_id* indicates the column number in the table. We can see three different *xtp_object_id* values, one for the table, one for the LOB column (*minor_id* = 8) and one for the row_overflow column (*minor_id* = 7). The first four rows for the table itself (with *type_desc* = 'USER_TABLE') show three rows for the indexes (two range indexes and one hash index). The range index and the table are stored in varheap structures and the hash index uses a hash structure. The row_overflow column has two varheaps, and the LOB column has three. For the row_overflow column the actual columns are stored in the Table heap structure for the column, and the Range index heap allows SQL Server to find the columns it needs. For the LOB column, the LOB data is stored in the LOB Page Allocator and the Table

heap for the LOB column just contains pointers. Like for row_overflow, the Range index heap allows fast access to various parts of the LOB data.

Rows

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 3 shows this structure, as well as expanding on the content of the header area.

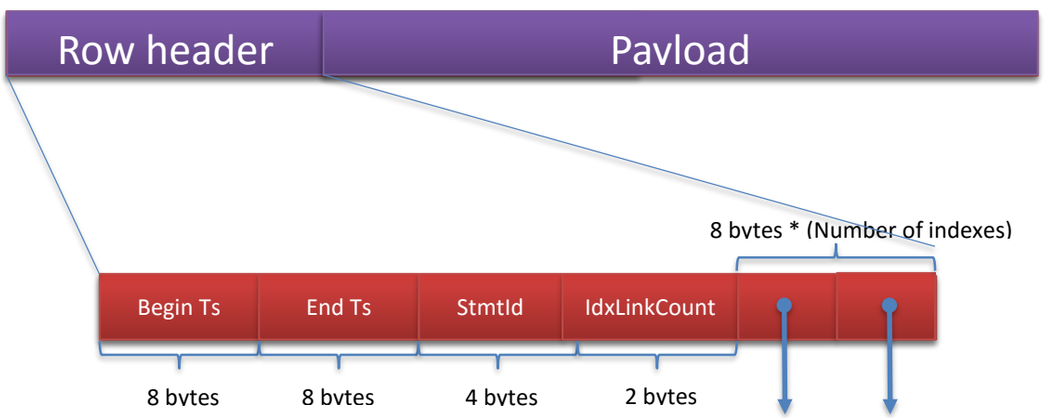

Figure 3 The structure of a row in a memory-optimized table

Row header

The header contains two 8-byte fields holding In-Memory OLTP timestamps: a Begin-Ts and an End-Ts. Every database that supports memory-optimized tables manages two internal counters that are used to generate these timestamps.

- The Transaction-ID counter is a global, unique value that is reset when the SQL Server instance is restarted. It is incremented every time a new transaction starts.
- The Global Transaction Timestamp is also global and unique, but is not reset on a restart. This value is incremented each time a transaction ends and begins validation processing. The new value is then the timestamp for the current transaction. The Global Transaction Timestamp value is initialized during recovery with the highest transaction timestamp found among the recovered records. (We'll see more about recovery later in this paper.)

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 the row. A special value (referred to as 'infinity') is used as the End-Ts value for rows that have not been deleted. However, when a row is first inserted, before the insert transaction is completed, the transaction's timestamp is not known, so the global Transaction_ID value is used for Begin-Ts until the transaction commits. At this time, the row is not visible to any other transactions. Similarly, for a delete operation, the transaction timestamp is not known, so the End-Ts value

for the deleted rows uses the global Transaction_ID value, which is replaced once the real Global Transaction Timestamp is assigned. Until the transaction commits, the row will still be visible, as it will not yet have been actually deleted. As we'll see when discussing data operations, once the transaction commits, 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. For example, an UPDATE statement will not update rows written by itself. The StmtId is used to enforce Halloween Protection.

> **Note**: Describing Halloween Protection in detail is beyond the scope of this paper, but there are many articles and blog posts you can find online about "The Halloween Problem" and "Halloween Protection". It is worth noting that the Halloween problem is handled differently for memory-optimized tables than for disk-based tables. For memory-optimized tables, it is so dealt with at the lowest level, the row header. For disk-based tables, the problem is dealt with through the query plan, for example by using spools.

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.

Payload area

The payload is the data 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 (i.e. the leaf node has all the columns from the table.) 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 because it knows the payload format used when inserting rows into a table, it can also generate the appropriate commands for all row operations.

Indexes On Memory-Optimized Tables

All memory-optimized tables must have at least one index. In SQL Server 2014, only the indexes would connect the rows in a table together. As mentioned earlier, data rows are not stored on pages, so there is no collection of pages or extents, no partitions or allocation units, that can be referenced to get all the pages for a table. There is some concept of *index* pages for one of the types of indexes, but they are stored differently than indexes for disk-based tables. In SQL Server 2016, since each table has its own varheap, all the rows for a particular table could be accessed by referencing the relevant memory consumer.

In-Memory OLTP indexes, and changes made to them during data manipulation, are never written to disk. Only the data rows, and changes to the data, are written to the transaction log. All indexes on memory-optimized tables are created based on the index definitions during database recovery. We'll cover details in the Checkpoint and Recovery section below.

Hash Indexes

A hash index consists of an array of pointers, and each element of the array is called a hash bucket. 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 4 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 used 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 in the record is NULL.

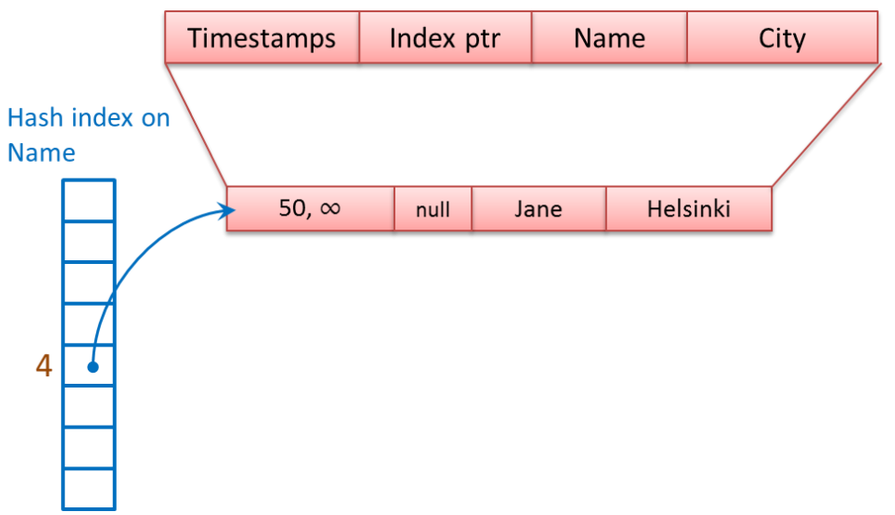

Figure 4 A hash index with a single row

In Figure 5, a row with a *name* value of Greg has been added to the table. Since Greg also has 4 characters, 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.

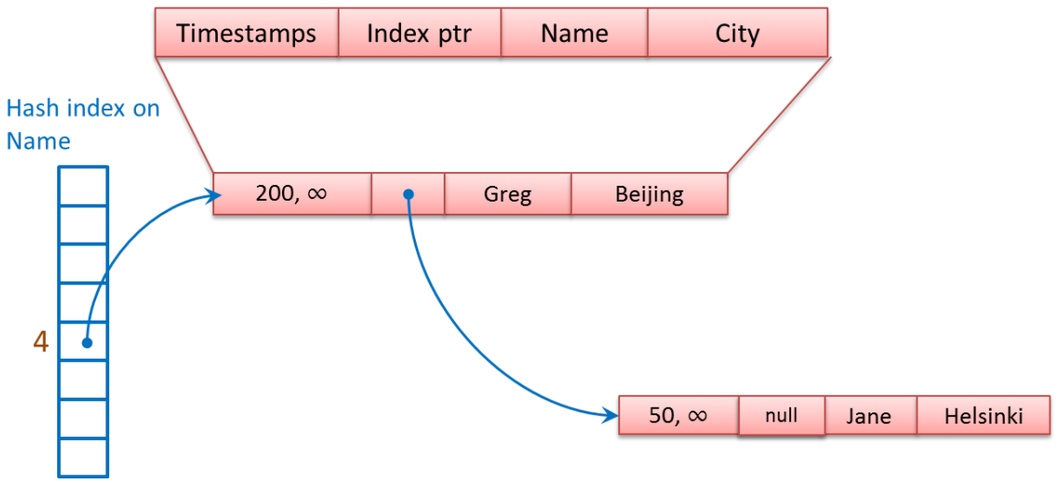

Figure 5 A hash index with two rows

A second hash index included in the table definition on the *City* column creates a second pointer field. 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 6 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.

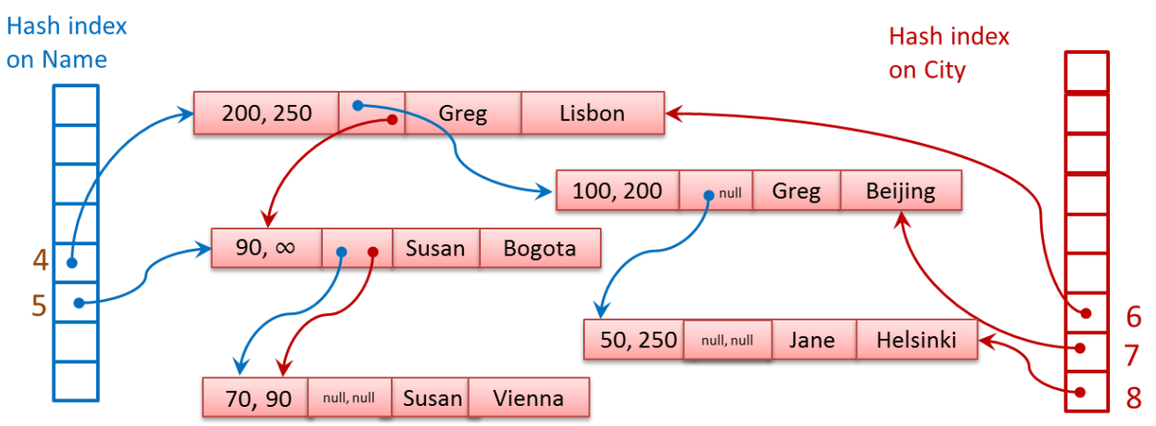

Figure 6 Two hash indexes on the same table

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 that is one to two times 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 8 bytes of memory. The number you supply is rounded up to the next power of two, so a value of 1,000,000 will be rounded up to 1,048,576 buckets, or 8 MB of memory space. 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. However, it is generally better to have too many buckets than too few.

When deciding to build a hash index, keep in mind that the hash function actually used is based on ALL the key columns. This means that if you have a hash index on the columns: *lastname, firstname* in an *employees* table, a row with the values "Harrison" and "Josh" will probably hash to a different bucket than a row with the values "Harrison" and "Joan". A query that just supplies a *lastname* value, or one with an inexact *firstname* value (such as "Jo%") will not be able to use the index at all.

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 implemented using a new data structure called a Bw-tree, originally envisioned and described by Microsoft Research in 2011. A Bw-tree is a lock- and latch-free variation of a B-tree.

The general structure of a Bw-tree is similar to SQL Server's regular B-trees, except that the index pages are not a fixed size, and once they are built they are unchangeable. Like a regular B-tree page, each index page contains a set of ordered key values, and for each value there is a corresponding pointer. At the upper levels of the index, on what are called the *internal pages*, the pointers point to an index page at the next level of the tree, and at the leaf level, the pointers point to a data row. Just like for In-Memory OLTP hash indexes, multiple data rows can be linked together. In the case of range indexes, rows that have the same value for the index key will be linked.

One big difference between Bw-trees and SQL Server's B-trees is that a page pointer is a logical page ID (PID), instead of a physical page number. The PID indicates a position in a mapping table, which maps each PID to a physical memory address. Index pages are never updated; instead, they are replaced with a new page and the mapping table is updated so that the same PID indicates a new physical memory address.

Figure 7 shows the general structure of Bw-tree, plus the Page Mapping Table.

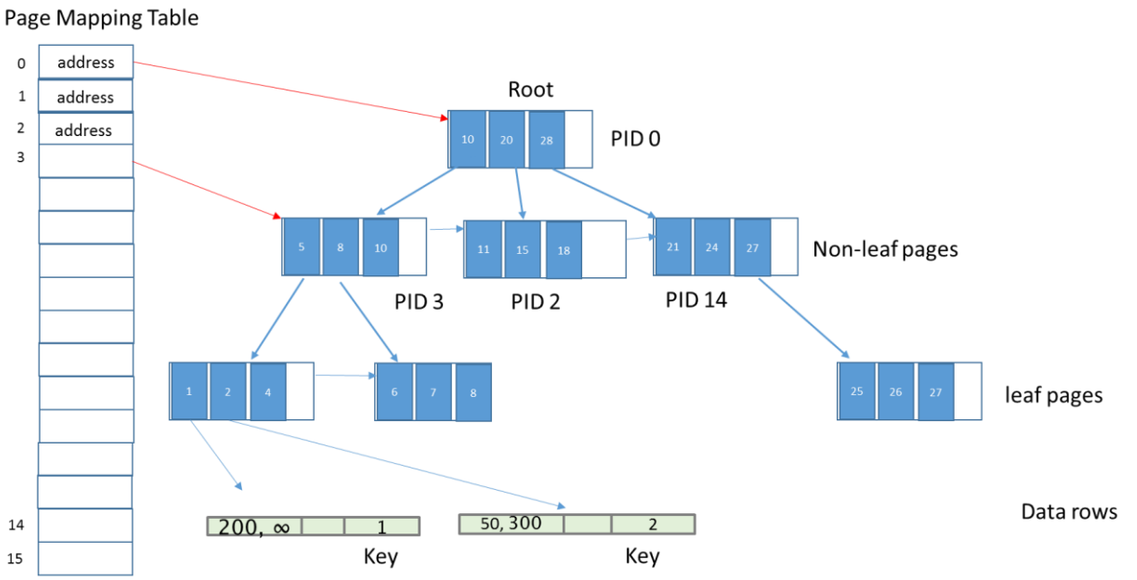

Figure 7 The general structure of a Bw-Tree

Not all the PID values are indicated in Figure 7, and the Mapping Table does not show all the PID values that are in use. The index pages are showing key values for this index. Each index row in the internal index pages contains a key value (shown) and a PID of a page at the next level down. The key value is the highest value possible on the page referenced. (Note this is different than a regular B-tree index, for which the index rows stores the **minimum** value on the page at the next level down.)

The leaf level index pages also contain key values, but instead of a PID, they contain an actual memory address of a data row, which could be the first in a chain of data rows, all with the same key value. (You can note another difference compared to regular B-tree indexes in that the leaf pages will not contain duplicates in the Bw-Tree. If a key value occurs multiple times in the data, there will a chain of rows pointed to by the single entry in the leaf.)

Another big difference between Bw-trees and SQL Server's B-trees is that at the leaf level, data changes are kept track of using a set of delta values. The leaf pages themselves are not replaced for every change. Each update to a page, which can be an insert or delete of a key value on that page, produces a page containing a delta record indicating the change that was made. An update is represented by two new delta records, one for the delete of the original value, and one for the insert of the new value. When each delta record is added, the mapping table is updated with the physical address of the page containing the newly added delta record. Figure 8 illustrates this behavior. The mapping table is showing only a single page with logical address P. The physical address in the mapping table originally was the memory address of the corresponding leaf level index page, shown as Page P. After a new row with index key value 50 (which we'll assume did not already occur in the table's data) is added to the table, In-Memory OLTP adds the delta record to Page P, indicating the insert of the new key, and the physical address of page P is updated to indicate the address of the first delta record page. Assume then that the only row with index key value 48 is deleted from the table. In-Memory OLTP must then remove the index row with key 48, so another delta record is created, and the physical address for page P is updated once again.

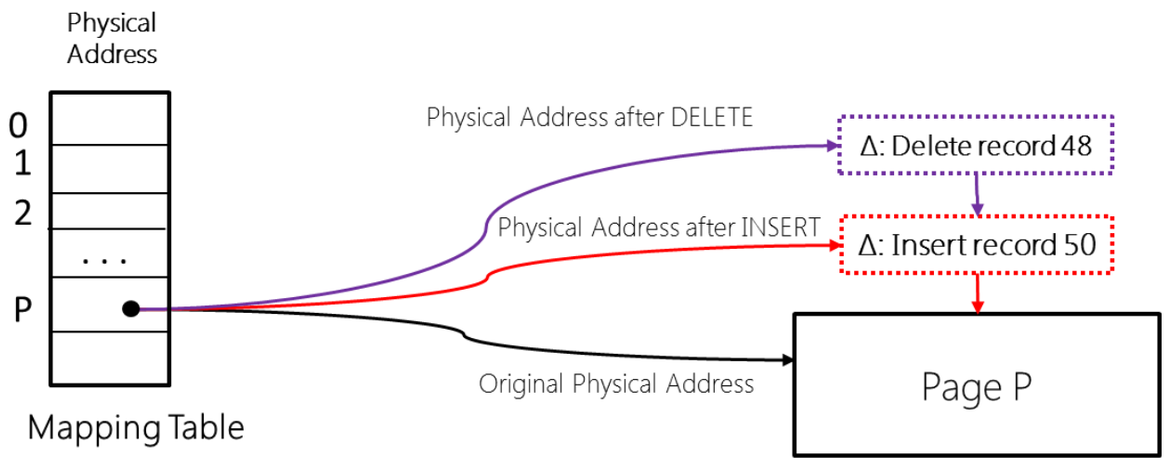

Figure 8 Delta records linked to a leaf level index page

When searching through a range index, SQL Server must combine the delta records with the base page, making the search operation a bit more expensive. However, not having to completely replace the leaf page for every change gives us a performance savings. As we'll see in the later section, *Consolidating Delta Records*, eventually SQL Server will combine the original page and chain of delta pages into a new base page.

Index page structures

In-Memory OLTP range index pages are not a fixed size as they are for indexes on disk-based tables, although the maximum index page size is still 8 KB.

Range index pages for memory-optimized tables all have a header area which contains the following information:

- PID - the pointer into the mapping table
- Page Type - leaf, internal, delta or special
- Right PID - the PID of the page to the right of the current page
- Height – the vertical distance from the current page to the leaf
- Page statistics – the count of delta records plus the count of records on the page
- Max Key – the upper limit of values on the page

In addition, both leaf and internal pages contains two or three fixed length arrays:

- Values – this is really a pointer array. Each entry in the array is 8 bytes long. For internal pages the entry contains PID of a page at the next level and for a leaf page, the entry contains the memory address for the first row in a chain of rows having equal key values. (Note that technically, the PID could be stored in 4 bytes, but to allow the same values structure to be used for all index pages, the array allows 8 bytes per entry.)
- Offsets – this array exists only for pages of indexes with variable length keys. Each entry is 2 bytes and contains the offset where the corresponding key starts in the key array on the page.

- Keys – this is the array of key values. If the current page is an internal page, the key represents the first value on the page referenced by the PID. If the current page is a leaf page, the key is the value in the chain of rows.

The smallest pages are typically the delta pages, which have a header which contains most of the same information as in an internal or leaf page. However, delta page headers don't have the arrays described for leaf or internal pages. A delta page only contains an operation code (insert or delete) and a value, which is the memory address of the first row in a chain of records. Finally, the delta page will also contain the key value for the current delta operation. In effect you can think of a delta page as being a mini-index page holding a single element whereas the regular index pages store an array of N elements.

Bw-tree internal reorganization operations

There are three different operations that SQL Server might need to perform while managing the structure of a Bw-tree: consolidation, split and merge. For all of these operations, no changes are made to existing index pages. Changes may be made to the mapping table to update the physical address corresponding to a PID value. If an index page needs to add a new row (or have a row removed) a whole new page is created and the PID values are updated in the Mapping Table.

Consolidating delta records

A long chain of delta records can eventually degrade search performance, if SQL Server has to consider the changes in the delta records along with the contents of the index pages when it's searching through an index. If In-Memory OLTP attempts to add a new delta record to a chain that already has 16 elements, the changes in the delta records will be consolidated into the referenced index page, and the page will then be rebuilt, including the changes indicated by the new delta record that triggered the consolidation. The newly rebuilt page will have the same PID but a new memory address. The old pages (index page plus delta pages) will be marked for garbage collection.

Splitting of a full index page

An index page in Bw-Tree grows on an as-needed basis, starting from storing a single row to storing a maximum of 8K bytes. Once the index page grows to 8K bytes, a new insert of a single row will cause the index page to split. For an internal page, this means when there is no more room to add another key value and pointer, and for a leaf page, it means that the row would be too big to fit on the page once all the delta records are incorporated.

The statistics information in the page header for a leaf page keeps track of how much space would be required to consolidate the delta records, and that information is adjusted as each new delta record is added. The easiest way to visualize how a page split occurs is to walk through an example. Figure 9 shows a representation of the original structure, where P_s is the page to be split into pages P_1 and P_2, and the P_p is its parent page, with a row that points to P_s. Keep in mind that a split can happen at any level of an index, so it not specified whether P_s is a leaf page or an internal page. It could be either.

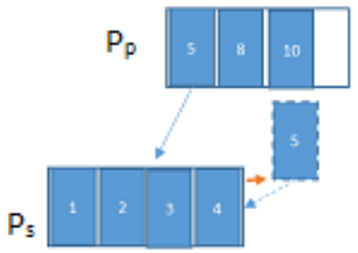

Figure 9: Attempting to insert a new row into a full index page.

Assume we have executed an `INSERT` statement that inserts a row with key value of 5 into this table, so that 5 now needs to be added to the range index. The first entry in Page P_p is a 5, which means 5 is the maximum value that could occur on the page to which P_p points, which is P_s. Page P_s doesn't currently have a value 5, but page P_s is where the 5 belongs. However, the Page P_s is full so that it is unable to add the key value 5 to the page, so it has to split.

The split operation occurs in two atomic steps, as described in the next two sections.

Step 1: Allocate new pages, split the rows

Step1 allocates two new pages, P_1 and P_2, and splits the rows from page P_s onto these pages, including the newly inserted row. A new slot in the Page Mapping table stores the physical address of page P_2. These pages, P_1 and P_2 are not yet accessible to any concurrent operations and those will see the original page P_s. In addition, the 'logical' pointer from $P_1$to P_2 is set. Figure 10 shows the changes, where P_1 contains key values 1 thru 3 and P_2 contains key values 4 and 5.

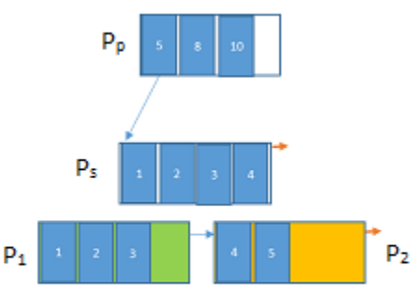

Figure 10: Splitting a full index page into two pages.

In the same atomic operation as splitting the page, SQL Server updates the Page Mapping Table to change the pointer to point to P_1 instead of P_s. After this operation, Page P_p points directly to Page P_1; there is no pointer to page P_s, as shown in Figure 11.

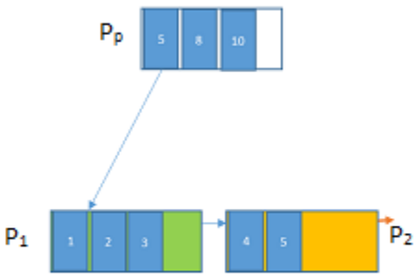

Figure 11: The pointer from the parent points to the first new child page.

Step 2: Create a new Pointer

Eventually, all pages should be directly accessible from the higher level but for a brief period, after Step 1, the parent page P_p points to P_1 but there is no direct pointer from P_p to page P_2, although P_2 contains the highest value that exists on P_2, which is 5. P_2 can be reached only via page P_1.

To create a pointer from P_p to page P_2, SQL Server allocates a new parent page P_{pp}, copies into it all the rows from page P_p, adds a new row to point to page P_1, which holds the maximum key value of the rows on P_1 which is 3, as shown in Figure 12.

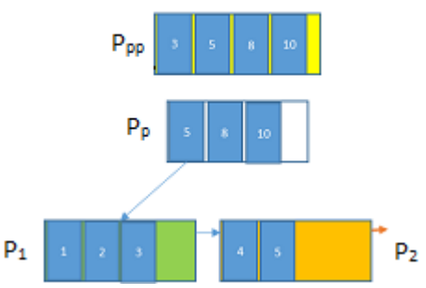

Figure 12: A new parent page is created.

In the same atomic operation as creating the new pointer, SQL Server then updates the Page Mapping Table to change the pointer from P_p to P_{pp}, as shown in Figure 13.

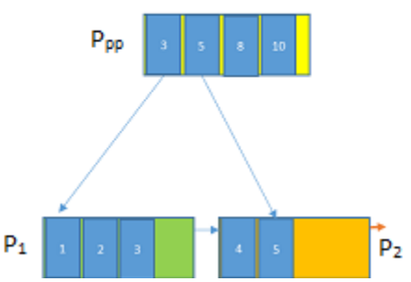

Figure 13: After the split is complete.

Merging of adjacent index pages

When a `DELETE` operation leaves an index page P with less than 10% of the maximum page size (currently 8K), or with a single row on it, SQL Server will merge page P with its neighboring page. When a row is deleted from page P, SQL Server adds a new delta record for the delete, as usual, and then checks to see if the remaining space after deleting the row will be less than 10% of maximum page size. If it will be then Page P qualifies for a Merge operation.

Again, to illustrate how this works, we'll walk through a simple example, which assumes we'll be merging a page P with its left neighbor, Page P_{ln}, that is, one with smaller values.

Figure 14 shows a representation of the original structure where page P_p, the parent page, contains a row that points to page P. Page P_{ln} has a maximum key value of 8, meaning that the row in Page P_p that points to page P_{ln} contains the value 8. We will delete from page P the row with key value 10, leaving only one row remaining, with the key value 9.

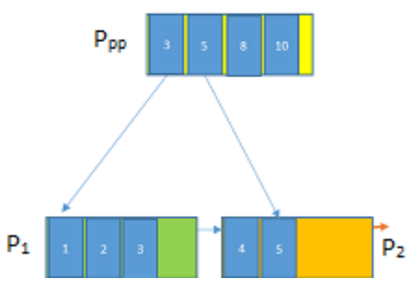

Figure 14: Index pages prior to deleting row 10.

The merge operation occurs in three atomic steps, as described over the following sections.

Step 1: Create New Delta pages for delete

SQL Server creates a delta page, DP_{10}, representing key value 10 and its pointer is set to point to Page P. Additionally, SQL Server creates a special 'merge-delta page', DPm, and links it to point to DP_{10}. At this stage, neither DP_{10} nor DP_m are visible to any concurrent transactions.

In the same atomic step, SQL Server updates the pointer to page P in the Page Mapping Table is to point to DP_m. After this step, the entry for key value 10 in parent page P_p now points to DP_m. You can see this in Figure 15.

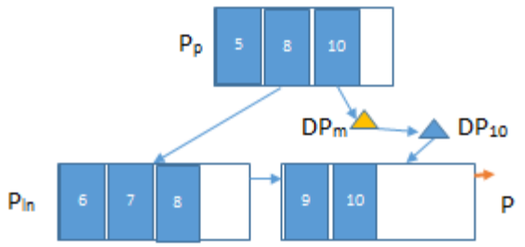

Figure 15: The delta page and the merge-delta page are added to indicate a deletion.

Step 2: Create a new non-leaf page with correct index entries

In step 2, SQL Server removes the row with key value 8 in page P_p (since 8 will no longer be the high value on any page) and updates the entry for key value 10 (DP_{10}) to point to page P_{ln}. To do this, it allocates a new non-leaf page, P_{p2}, and copies to it all the rows from P_p except for the row representing key value 8.

Once this is done, in the same atomic step, SQL Server updates the page mapping table entry pointing to page P_p to point to page P_{p2}. Page P_p is no longer reachable. This is shown in Figure 16.

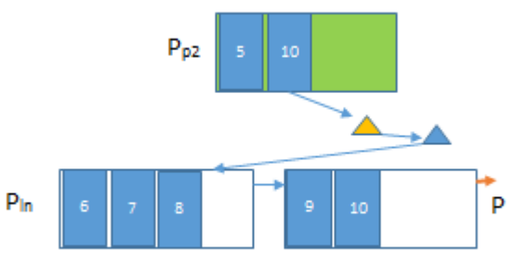

Figure 16: Pointers are adjusted to get ready for the merge.

Step 3: Merge pages, remove deltas

In the final step, SQL Server merges the leaf pages P and P_{ln} and removes the delta pages. To do this, it allocates a new page, P_{new}, merges the rows from P and P_{ln}, and includes the delta page changes in the new P_{new}. Finally, in the same atomic operation, SQL Server updates the page mapping table entry currently pointing to page P_{ln} so that it now points to page P_{new}. At this point, the new page, as shown in Figure 17, is available to any concurrent transactions.

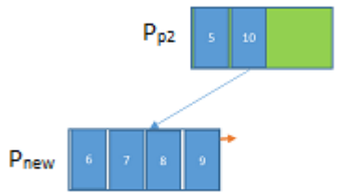

Figure 17: After the merge is completed.

Columnstore Indexes

In SQL Server 2012, the version in which columnstore indexes were first introduced, and SQL Server 2014, in which updateable clustered columnstore indexes were added, these indexes were intended for analytics, including reports and analysis. In-memory OLTP, on the other hand, as the name 'OLTP' implies, was intended for operational data that was very quickly growing and changing. However, in many systems the line is blurred as to what data is operational and what is for analytics, and the same data may need to be available for both purposes. Instead of bending each solution (columnstore indexes and memory-optimized tables) to do what they are not designed to do, that is, to force memory-optimized tables to be better with analytic queries and columnstore indexes to be better with OLTP data modification – and eventually end up with half-baked solutions, SQL Server 2016 provides a solution that leverages both technologies in their own strength and hide the seams and storage details from the users.

Columnstore Index Basic Architecture

Because SQL Server 2016 provides support for building clustered columnstore indexes on memory-optimized tables, this paper will provide a high-level overview of columnstore index architecture. It is beyond the scope of this document to discuss all the details of columnstore indexes, but a basic understanding of columnstore index storage will be useful to understand the special considerations for columnstore indexes on memory-optimized tables. For more complete details, please take a look at the documentation: https://msdn.microsoft.com/library/gg492088.aspx.

Columnstore indexes, which are typically wide composite indexes, are not organized as rows, but as columns. Rows are grouped into rowgroups of about one million rows each. (The actual maximum is 2^{20} or 1,048,576 rows per group.) SQL Server will attempt to put the full 2^{20} values in each rowgroup, leaving a final rowgroup with whatever leftover rows there are. For example, if there are exactly 10 million rows in a table, the columnstore index could have 9 rowgroups of 1,048,576 rows and one of 562,816 rows. However, because the index is usually built in parallel, with each thread processing its own subset of rows, there may be multiple rowgroups with fewer than the full 1,048,576 values.

Within each rowgroup, SQL Server applies its Vertipaq compression technology which encodes the values and then rearranges the rows within the rowgroup to give the best compression results. Each index column in each rowgroup is stored separately, in a structure called a segment. Each segment is stored as a Large Object (LOB) value, and stored in the LOB allocation unit. Segments are the basic unit of

manipulation and can be read and written separately. Figure 18 illustrates the encoding and conversion of a set of values for multiple index columns into several segments.

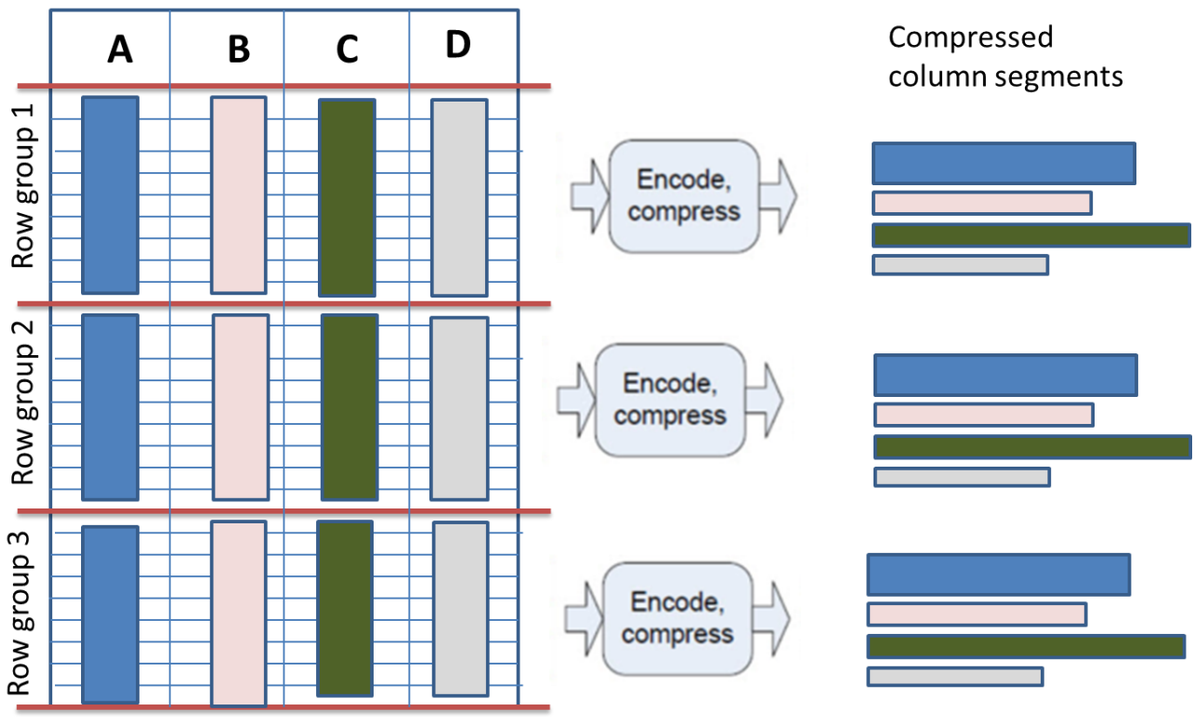

Figure 18. Transforming columnstore index columns into segments

The table in Figure 18 has been divided into three rowgroups where all four columns from the table are defined as part of the columnstore index. We end up with 12 compressed column segments, three segments for each of the four columns.

To support updating a clustered columnstore index, two additional structures are introduced. There is a separate internal table, the 'deleted rows table', or DRT, frequently referred to as the 'deleted bitmap', which stores the Row IDs of all rows that have been deleted. The new rows added to the table are initially stored in a heap called the Delta Store. When a sufficient number of rows have been added (usually 2^{20} or about a million rows), SQL Server will convert those rows into a new compressed rowgroup. Columnstore indexes on memory-optimized tables

The compressed row groups for a clustered columnstore index on a memory-optimized table are stored separately from the rows accessible from the memory-optimized nonclustered indexes described above (the hash and range indexes) and are basically a copy of the data. Note that this is different than a clustered columnstore index on a disk-based table, for which a clustered columnstore index is the only copy of the data that exists. Because the columnstore index is a copy of the data, there will be additional memory required. In fact, you can consider a clustered columnstore index on a memory-optimized table as a nonclustered columnstore index with all the table's columns included as part of the index. However, because the data is so efficiently compressed, the overhead is minimal. In many cases, clustered columnstore index data can be compressed down to only 10% of the original data size, so the overhead is only that 10% needed for the columnstore index.

All columnstore index segments are all fully resident in memory at all times. For recovery purposes, each rowgroup of the clustered columnstore index is stored in a separate file in the memory-optimized filegroup, with a type of LARGE DATA. Within the file for a particular rowgroup, all the segment are stored together, but SQL Server maintains a pointer to each segment and can access a segment individually when fetching the subset of columns referenced in a query. These files are discussed below in the section on CHECKPOINT FILES. As new rows are added to a memory-optimized table with a columnstore index, they are not immediately added to the compressed rowgroups of the columnstore index; instead, the new rows are only available as regular rows accessible through any of the other memory-optimized table's indexes. Figure 19 shows these new rows as maintained separately from the rest of the table. You can think of these rows, labelled as the "Delta Rowgroup" as similar to the Delta Store in a disk-based table with a clustered columnstore index, but it's not exactly the same because these rows are part of the memory-optimized table and NOT technically part of the columnstore index. This structure is actually a virtual delta rowgroup.

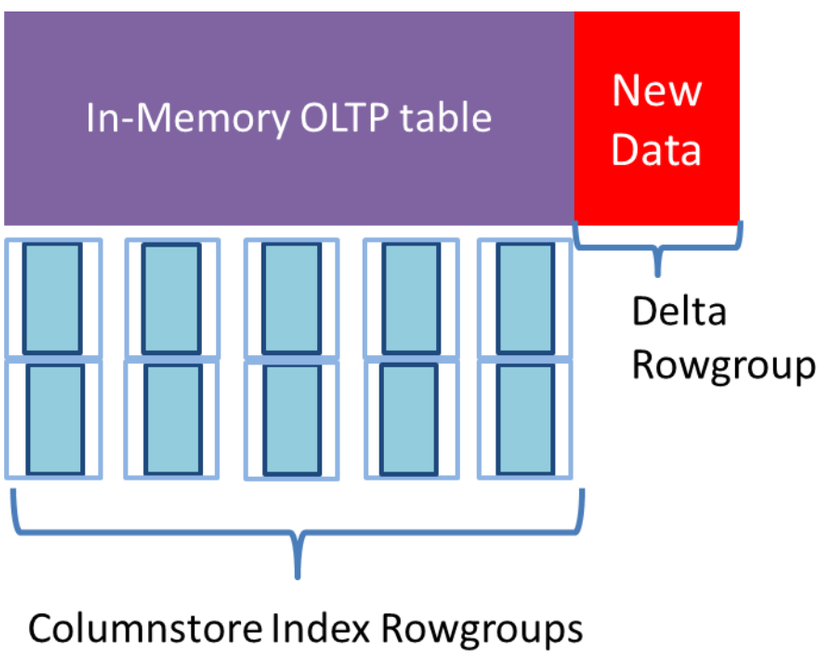

Figure 19: A columnstore index on a memory-optimized table.

Columnstore indexes on memory-optimized tables can only be chosen by the optimizer for queries running in Interop mode. These queries using a columnstore index can run in parallel and take full advantage of batch mode execution for higher performance. The optimizer will never choose a columnstore index for tables accessed in natively compiled procedure, and no queries in natively compiled procedures will run in parallel. If a SQL Server 2016 memory-optimized table has a clustered columnstore index, there will be two varheaps, which can be seen in the list of memory consumers described above, and will also be shown in the example below. One of the varheaps will be for the compressed rowgroups, and the new rows will be allocated from a separate varheap This separate storage allows SQL Server to quickly identify the rows that have not yet been compressed into segments of the columnstore index, that is, the rows that are part of the virtual delta rowgroup.

A background thread wakes up every 2 minutes (by default) and examines the rows that have been added to delta rowgroup. Note that these rows include newly inserted rows, and rows that have been updated, as UPDATE is always performed by deleting the old row and inserting a new one. If the count of such rows exceeds 1 million, the thread performs the following two operations: when

1. The rows are copied into one or more rowgroups, from which each of the segments will compressed and encoded to become part of the clustered columnstore index.
2. The rows will be moved from the special memory allocator to the regular memory storage with the rest of the rows from the table.

SQL Server does not actually count the rows; it uses a best guess estimate. In no case can the number of rows in a columnstore index rowgroup exceed 1,048,576. If there are more 100,000 leftover rows after the rowgroup is compressed, another rowgroup will be created. If there are less than 100,000 rows leftover, those rows will remain in the OPEN filegroup.

If you would like to delay the compression of newly added rows, because update operations might be performed on them, or they might even be quickly removed due to your application logic, you can configure a waiting period. When a memory-optimized table with a clustered columnstore index is created, you can add an option called COMPRESSION_DELAY that specifies how many minutes a row must exist in the delta rowgroup before it is considered for compression. And only when the delta rowgroup accumulates a million rows that have been there for more than this number of minutes will those rows be compressed into the regular columnstore index rowgroups.

After rows have been converted to compressed rowgroups, all delete operations are indicated by marking the row as deleted in the Deleted Rows Table, the same as for a clustered columnstore index on a disk-based table. Lookups can be very inefficient once you have a large number of deleted rows. There is no way to do any kind of reorganization on a columnstore index, other than dropping and rebuilding the index. However, once 90% of the rows in a rowgroup have been deleted, the remaining 10% are automatically reinserted into the uncompressed varheap, in the "Delta Rowgroup" of the memory-optimized table. The storage used for the rowgroup will then be available for garbage collection.

> Note: As mentioned above, if a memory-optimized table has any LOB or off-row columns, a columnstore index cannot be built on it. This means that the absolute maximum row size for the table is 8060 bytes. In addition, once a memory-optimized table has a columnstore index, no ALTER operations can be performed on the table. The workaround would be to drop the columnstore index, perform the ALTER, and then create the index.

The following script creates a memory-optimized table with two indexes (one supports the primary key), one range index and a columnstore index, and then runs the same metadata query we've used before to examine the memory consumers. This script also sets the COMPRESSION_DELAY value for the columnstore index compression to 60 minutes, so SQL Server will only evaluation new rows for compression after they have been in the table for an hour.

```
USE master
GO
SET NOCOUNT ON
GO
DROP DATABASE IF EXISTS IMDB;
GO
CREATE DATABASE IMDB
GO
----- Enable database for memory optimized tables
-- add memory_optimized_data filegroup
ALTER DATABASE IMDB
    ADD FILEGROUP IMDB_mod_FG CONTAINS MEMORY_OPTIMIZED_DATA
GO

-- add container to the filegroup
ALTER DATABASE IMDB
  ADD FILE (NAME='IMDB_mod', FILENAME='c:\HKData\IMDB_mod')
  TO FILEGROUP IMDB_mod_FG
GO
--- create the table
USE IMDB
GO
DROP TABLE IF EXISTS dbo.OrderDetailsBig;
GO
-- create a simple table
CREATE TABLE dbo.OrderDetailsBig
(
  OrderID int NOT NULL,
  ProductID int NOT NULL,
  UnitPrice money NOT NULL,
  Quantity smallint NOT NULL,
  Discount real NOT NULL

  INDEX IX_OrderID NONCLUSTERED HASH (OrderID) WITH (BUCKET_COUNT = 20000000),
  INDEX IX_ProductID NONCLUSTERED (ProductID),
  CONSTRAINT PK_Order_Details PRIMARY KEY
           NONCLUSTERED (OrderID,      ProductID),
  INDEX clcsi_OrderDetailsBig CLUSTERED COLUMNSTORE WITH (COMPRESSION_DELAY = 60)
) WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
GO
-- Examine the memory consumers for this table
SELECT object_name(c.object_id) AS table_name, a.xtp_object_id, a.type_desc,
             minor_id, memory_consumer_id as consumer_id,
             memory_consumer_type_desc as consumer_type_desc,
             memory_consumer_desc as consumer_desc,
             convert(numeric(10,2), allocated_bytes/1024./1024) as allocated_MB,
                      convert(numeric(10,2), used_bytes/1024./1024) as used_MB
  FROM sys.memory_optimized_tables_internal_attributes a
    JOIN sys.dm_db_xtp_memory_consumers c
          ON a.object_id = c.object_id and a.xtp_object_id = c.xtp_object_id
            LEFT JOIN sys.indexes i
                ON c.object_id = i.object_id
             AND c.index_id = i.index_id;
GO
```

My results are shown in Figure 20.

table_name	xtp_object_id	type_desc	minor_id	consumer_id	consumer_type_desc	consumer_desc	allocated_MB	used_MB
OrderDetailsBig	-2147483648	USER_TABLE	0	207	HKCS_COMPRESSED	HkCS Allocator	16.00	16.00
OrderDetailsBig	-2147483648	USER_TABLE	0	206	VARHEAP	Range index heap	8959.10	6998.17
OrderDetailsBig	-2147483648	USER_TABLE	0	200	HASH	Hash index	262117.79	262117.79
OrderDetailsBig	-2147483648	USER_TABLE	0	199	HASH	Hash index	262117.79	262117.79
OrderDetailsBig	-2147483648	USER_TABLE	0	198	VARHEAP	Table heap	0.00	0.00
OrderDetailsBig	-2147483648	USER_TABLE	0	197	VARHEAP	Table heap	43835.62	43535.02
OrderDetailsBig	-2147483644	ROW_GROUPS_INFO_TABLE	0	194	HASH	Hash index	1023.90	1023.90
OrderDetailsBig	-2147483644	ROW_GROUPS_INFO_TABLE	0	193	VARHEAP	Table heap	63.99	0.09
OrderDetailsBig	-2147483645	SEGMENTS_TABLE	0	192	HASH	Hash index	32764.72	32764.72
OrderDetailsBig	-2147483645	SEGMENTS_TABLE	0	191	HASH	Hash index	32764.72	32764.72
OrderDetailsBig	-2147483645	SEGMENTS_TABLE	0	190	VARHEAP	Table heap	0.00	0.00
OrderDetailsBig	-2147483646	DICTIONARIES_TABLE	0	189	HASH	Hash index	32764.72	32764.72
OrderDetailsBig	-2147483646	DICTIONARIES_TABLE	0	188	VARHEAP	Table heap	0.00	0.00
OrderDetailsBig	-2147483647	DELETED_ROWS_TABLE	0	187	VARHEAP	Range index heap	127.99	0.15
OrderDetailsBig	-2147483647	DELETED_ROWS_TABLE	0	186	VARHEAP	Table heap	0.00	0.00

Figure 20: Memory consumers for memory-optimized table with columnstore index

The output in Figure 20 shows six rows for the table itself. There is one memory consumer for the compressed rowgroups (the HKCS_COMPRESSED consumer), one for the range index, two for the hash indexes and two for the table rowstore. One of the rowstore varheaps is for most of the table rows, and the second one is for the delta rowgroup, containing the newly added rows that are not yet also part of the compressed rowgroups. There are also four internal tables for any table with a columnstore index; note that they all have different *xtp_object_id* values. Each of these internal tables has at least one index to help SQL Server access the information in the internal table efficiently. The four internal tables, which you can see in Figure 20, are ROW_GROUPS_INFO_TABLE (plus hash index), SEGMENTS_TABLE (plus two hash indexes), DICTIONARIES_TABLE (plus hash index) and DELETED_ROWS_TABLE (plus range index). (The details of what these internal tables are used for is not specifically an In-memory OLTP topic, so it is out of scope for this paper.)

Besides looking at the DMVs for memory consumers, another DMV that you might want to examine is *sys.dm_db_column_store_row_group_physical_stats*. This view not only tells you how many rows are in each COMPRESSED and OPEN rowgroup in a columnstore index, but for rowgroups with less than the maximum number of rows, it tells you why there isn't the maximum number. If you would like to see this information for yourself, you can run this script below, which inserts 10,000,000 rows into the table created above, and examines several values in *sys.dm_db_column_store_row_group_physical_stats*.

```
USE IMDB
GO
SET NOCOUNT ON
GO
BEGIN TRAN
DECLARE @i int = 0
WHILE (@i < 10000000)
BEGIN
    INSERT INTO dbo.OrderDetailsBig VALUES (@i, @i % 1000000, @i % 57, @i % 10, 0.5)
    SET @i = @i + 1
       IF (@i % 264 = 0)
       BEGIN
              COMMIT TRAN;
              BEGIN TRAN;
       END
END
COMMIT TRAN;
GO

--
SELECT row_group_id, state_desc, total_rows, trim_reason_desc
FROM sys.dm_db_column_store_row_group_physical_stats
WHERE object_id = object_id('dbo.OrderDetailsBig')
ORDER BY row_group_id;
GO
```

My results are shown in Figure 21. Note that the actual numbers you get may vary.

row_group_id	state_desc	total_rows	trim_reason_desc
-1	OPEN	0	NULL
1	COMPRESSED	1048576	NO_TRIM
2	COMPRESSED	1048576	NO_TRIM
3	COMPRESSED	1048576	NO_TRIM
4	COMPRESSED	324351	SPILLOVER
5	COMPRESSED	430649	SPILLOVER
6	COMPRESSED	1048576	NO_TRIM
7	COMPRESSED	928262	STATS_MISMATCH
8	COMPRESSED	956730	STATS_MISMATCH
9	COMPRESSED	313504	SPILLOVER
10	COMPRESSED	1048576	NO_TRIM
11	COMPRESSED	1048576	NO_TRIM
12	COMPRESSED	384784	SPILLOVER
13	COMPRESSED	370264	SPILLOVER

Figure 21: Columnstore index rowgroup metadata showing why a rowgroup had fewer than the maximum rows

In the output in Figure 21 you can see three different values in the *trim_reason_desc* column, which is an explanation of the reason why a COMPRESSED rowgroup has fewer than 1,048.576 rows. Obviously, for a rowgroup with the maximum number of rows, we don't need any explanation, so the value is NO_TRIM. The OPEN rowgroup is not compressed, so its value is always NULL. The value STATS_MISMATCH indicates the estimate of the number of rows was too low, and there actually weren't the full number when the compression was performed. The fourth value, SPILLOVER, is used for rowgroups that contain the leftover rows after a full rowgroup is created.

Index Metadata

We have already seen some of the metadata, for the memory consumers. In addition, index metadata is available in several catalog views and DMVs. The *sys.indexes* view contains one row for each index on each memory-optimized table. The type column will have one of the following values for the indexes on a memory-optimized table:

Type 2: Nonclustered (range)
Type 5: Clustered columnstore
Type 7: Nonclustered hash

The following query will show you all your indexes on your memory-optimized tables:

```
SELECT t.name AS table_name, i.name AS index_name, index_id, i.type, i.type_desc
FROM sys.tables t JOIN sys.indexes i
  ON t.object_id = i.object_id
WHERE is_memory_optimized = 1;
GO
```

In addition, the view *sys.hash_indexes*, which contains all the columns from *sys.indexes*, but only the rows where *type* = 7, has one additional column: *bucket_count*.

Storage space used by your memory-optimized tables and their indexes is shown in the DMV *sys.dm_db_xtp_table_memory_stats*. The following query lists each memory-optimized table and the space allocated used, both for the data and for the indexes.

```
SELECT object_name(object_id) AS 'Object name', *
FROM sys.dm_db_xtp_table_memory_stats;
GO
```

You might also want to inspect the following dynamic management objects to help manage your memory-optimized tables:

dm_db_xtp_index_stats – The is view reports on the number of rows in each index, as well as the number of times each index was accessed and the number of rows that are no longer valid, based on the oldest timestamp in the database.

dm_db_xtp_hash_index_stats – This view can help you manage and tune the bucket counts of your hash indexes, as it reports on the number of empty buckets, and the average and maximum chain length.

dm_db_xtp_nonclustered_index_stats – This view can help you manage your range indexes. It includes information about the operation on the Bw-tree including page splits and merges.

Other DMVs will be described later, as they become relevant to the topics discussed.

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. Earlier this value was referred to as a Global Transaction Timestamp. The timestamps are generated by a monotonically increasing counter which increases every time a transaction commits. When a transaction starts, it assumes the highest timestamp in the database at as its start time, 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 we saw back 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. Back 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.

The notion of version visibility is fundamental to proper concurrency control with 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.

Isolation Levels Allowed with Memory-Optimized Tables

Data operations on memory-optimized tables always use optimistic multi version concurrency control (MVCC). Optimistic data access does not use locking or latching to provide transaction isolation. We'll look at the details of how this lock and latch-free behavior is managed, as well as details on the reasons for the allowed transaction isolation levels in a later section. In this section, we'll only be discussing the details of transaction isolation level necessary to understand the basics of data access and modification operations.

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

- SNAPSHOT
- REPEATABLE READ
- SERIALIZABLE

The transaction isolation level must be specified as part of the ATOMIC block of a natively compiled stored procedure. When accessing memory-optimized tables from interpreted Transact-SQL, the isolation level should be specified using table-level hints or a database option called `MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT`. This option will be discussed later, after we have looked at isolation levels for accessing memory-optimized tables.

The isolation level READ COMMITTED is supported for memory optimized tables with autocommit (single statement) transactions. It is not supported with explicit or implicit user transactions. (Implicit transactions are those invoked under the session option IMPLICIT_TRANSACTIONS. In this mode, behavior is the same as for an explicit transaction, but no BEGIN TRANSACTION statement is required. Any DML statement will start a transaction, and the transaction must be explicitly either committed or rolled back. Only the BEGIN TRANSACTION is implicit.) Isolation level READ_COMMITTED_SNAPSHOT is supported for memory-optimized tables with autocommit transactions 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 previously described, 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 with transaction ID 100 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 100 with an extra flag bit indicating that the value is a transaction ID. Any other transaction that now attempts to access the row finds that the end timestamp contains a transaction ID (100) which indicates that the row may have been deleted. It then locates TX1 in the transaction map and checks 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 INSERT a complete new row. This begins by constructing the new row <Jane, Perth> with begin timestamp 100 containing a flag bit indicating that it is a transaction ID, 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. Then <Jane, Perth> is inserted by linking it into both indexes. Next <Jane, Helsinki> is deleted just as described for the DELETE operation in the preceding paragraph. Any other transaction that attempts to update or delete <Jane, Helsinki> will notice that the end timestamp does not contain infinity but a transaction ID, conclude that there is write-write conflict, and immediately abort.

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 at the end of 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 22. This write set forms the content of the log for the transaction. Transactions normally generate only a single log record that contains its ID and commit timestamp 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 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 type flag on the row's end timestamp field.
- 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 type flag on the row's begin timestamp field

The actual unlinking and deletion of old row versions is handled by the garbage collection system, which will be discussed below.

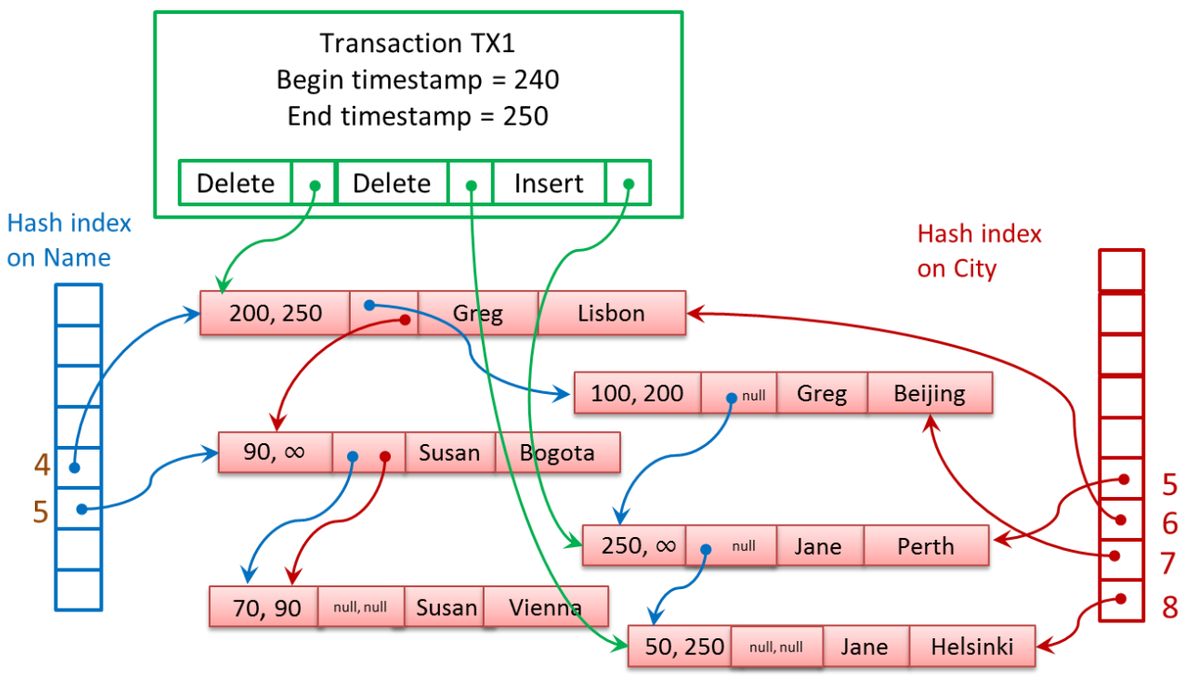

Figure 22 Transactional Modifications on a table

Reading

Now let's look at the read transactions, TX2 and TX3, which will be processed concurrently 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;
GO
```

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 valid 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 and transaction TX3 will roll back. We'll see more about validation in the next section.

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 Transact-SQL

When using the interop capability, you will have access to virtually the full Transact-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 ad hoc 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 Transact-SQL should also be used when you need to access both memory-optimized tables and disk-based tables.

The only Transact-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
- All locking hints: TABLOCK, XLOCK, PAGLOCK, etc. (NOLOCK is supported, but is quietly ignored.)
- Isolation level hints READUNCOMMITTED, READCOMMITTED and READCOMMITTEDLOCK

- Other table hints: IGNORE_CONSTRAINTS, IGNORE_TRIGGERS, NOWAIT, READPAST, SPATIAL_WINDOW_MAX_CELLS

T-SQL in Natively Compiled Procedures

Natively compiled stored procedures allow you to execute Transact-SQL in the fastest way, which includes accessing data in memory-optimized tables. There are however, additional 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 Transact-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. Many of the restrictions on Transact-SQL in natively compiled procedures in SQL Server 2014 have been removed in SQL Server 2016, and more will be removed in subsequent versions. Some of the constructs that are available in natively compiled procedures in SQL Server 2016 that were not available in SQL Server 2014 are the following:

- LEFT and RIGHT OUTER JOIN
- SELECT DISTINCT
- OR and NOT operators
- Subqueries in all clauses of a SELECT statement
- Nested stored procedure calls
- UNION and UNION ALL
- All built-in math functions
- Some security functions, including @@spid
- Scalar user-defined functions
- Inline table-valued functions
- EXECUTE AS CALLER

For the full list of supported features in natively compiled procedures please refer to the documentation: https://msdn.microsoft.com/library/dn452279.aspx

For the list of features that are not supported in in-memory OLTP, please refer to the following page: https://msdn.microsoft.com/library/dn246937.aspx

Garbage Collection of Rows in Memory

Because In-Memory OLTP is a multi-versioning system, your DELETE and UPDATE operations (as well as aborted INSERT operations) will generate row versions that will eventually become stale, which means they will no longer be visible to any transaction. These unneeded versions will slow down scans of index structures and create unused memory that needs to be reclaimed.

The garbage collection process for stale versions in your memory-optimized tables is analogous to the version store cleanup that SQL Server performs for disk-based tables using one of the snapshot-based isolation levels. Unlike disk-based tables where row versions are kept in *tempdb*, the row versions for memory-optimized tables are maintained in the in-memory table structures themselves.

To determine which rows can be safely deleted, the system keeps track of the timestamp of the oldest active transaction running in the system, and uses this value to determine which rows are still potentially needed. Any rows that are not valid as of this point in time (that is, their end-timestamp is earlier than this time) are considered *stale*. Stale rows can be removed and their memory can be released back to the system.

The garbage collection system is designed to be non-blocking, cooperative, efficient, responsive and scalable. Of particular interest is the 'cooperative' attribute. Although there is a dedicated system thread for the garbage collection process, user threads actually do most of the work. If a user thread is scanning an index (and all index access on memory-optimized tables is considered to be scanning) and it comes across a stale row version, it will unlink that version from the current chain and adjust the pointers. It will also decrement the reference count in the row header area. In addition, when a user thread completes a transaction, it then adds information about the transaction to a queue of transactions to be processed by the garbage collection process. Finally, it picks up one or more work items from a queue created by the garbage collection thread, and frees the memory used by the rows making up the work item.

The garbage collection thread goes through a queue of completed transactions about once a minute, but the system can adjust the frequency internally based on the number of completed transactions waiting to be processed. From each transaction, it determines which rows are stale, and builds work items made up of a set of rows that are ready for removal. These work items are distributed across multiple queues, one for each CPU used by SQL Server. Normally, the work of actually removing the rows from memory is left to the user threads which process these work items from the queues, but if there is little user activity, the garbage collection thread itself can remove rows to reclaim system memory.

The DVM *sys.dm_db_xtp_index_stats* has a row for each index on each memory-optimized table, and the column *rows_expired* indicates how many rows have been detected as being stale during scans of that index. There is also a column called *rows_expired_removed* that indicates how many rows have been unlinked from that index. As mentioned above, once rows have been unlinked from all indexes on a table, it can be removed by the garbage collection thread. So you will not see the *rows_expired_removed* value going up until the *rows_expired* counters have been incremented for every index on a memory-optimized table.

The following query allows you to observe these values. It joins the *sys.dm_db_xtp_index_stats* DMV with the *sys.indexes* catalog view to be able to return the name of the index.

```
SELECT name AS 'index_name', s.index_id, scans_started, rows_returned,
       rows_expired, rows_expired_removed
FROM sys.dm_db_xtp_index_stats s JOIN sys.indexes i
     ON s.object_id=i.object_id and s.index_id=i.index_id
WHERE object_id('<memory-optimized table name>') = s.object_id;
GO
```

Depending on your volume of data changes and the rate and which new versions are generated, SQL Server can be using a substantial amount of memory for old row versions and you'll need to make sure that your system has enough memory available.

Transaction Processing

As mentioned above, all access of data in memory-optimized tables is done using completely optimistic concurrency control, but multiple transaction isolation levels are still allowed. However, what isolation levels are allowed in what situations might seem a little confusing and non-intuitive. The isolation levels we are concerned about are the ones involving a ***cross-container transaction,*** which means any interpreted query that references memory-optimized tables whether executed from an explicit or implicit transaction or in auto-commit mode. The isolation levels that can be used with your memory-optimized tables in a cross-container transaction depend on what isolation level the transaction has defined for the SQL Server transaction. Most of the restrictions have to do with the fact that operations on disk-based tables and operations on memory-optimized tables each have their own transaction sequence number, even if they are accessed in the same Transact-SQL transaction. You can think of this behavior as having two sub-transactions within the larger transaction: one sub-transaction is for the disk-based tables and one is for the memory-optimized tables.

A DMV that can be useful for monitoring transactions in progress is *sys.dm_db_xtp_transactions*. You can think of this view as allowing you to peek into the global transaction table that was mentioned earlier. We'll be looking at this view in more detail later in this paper, but we can take a quick look at it now. I started two simple transactions doing inserts into a memory-optimized table, and then ran the following query:

```
SELECT xtp_transaction_id, transaction_id, session_id,
       begin_tsn, end_tsn, state_desc
FROM sys.dm_db_xtp_transactions;
WHERE transaction_id > 0;
GO
```

My output shows two transactions:

	xtp_transaction_id	transaction_id	session_id	begin_tsn	end_tsn	state_desc
1	20049	1296390	70	8005	0	ACTIVE
2	20048	1296385	66	8005	0	ACTIVE

The *xtp_transaction_id* in the first column is the In-Memory OLTP Transaction-ID, and you can see that my two transactions have consecutive values. These are very different values than the *transaction_id* in the second column, which is the id for the 'regular' transaction, that the In-memory OLTP transaction is a part of. The *xtp_transaction_id* is the value used as the end-timestamp for records this transaction deletes and the begin-timestamp for rows this transaction inserts, before this transaction commits and gets timestamp of its own. We can also see that both of these transactions have the same value for *begin_tsn*, which is the current timestamp (for the last committed transaction) at the time this transaction started. There is no value for the end-timestamp because these transactions are still in progress.

When a transaction is submitted to the In-Memory OLTP engine for processing, it goes through the following steps:

1. Query processing

- When the first statement accessing a memory-optimized table is executed, or when a natively compiled module starts execution, SQL Server obtains a transaction-id for the Transact-SQL part of the transaction and a transaction-ID for the In-Memory OLTP portion. If any query tries to update a row that has already been updated by an active transaction, an 'update conflict' error is generated. Most other isolation level errors are not caught until the validation phase. Depending on the isolation level, the In-Memory OLTP engine keeps track of a read-set and write-set, which are sets of pointers to the rows that have been read or written, respectively. Also depending on the isolation level, it will keep track of a scan-set, which is information about the predicate used to access a set of records. If the transaction commits, an end-timestamp is obtained, but the transaction is not really committed until after the validation phase.
-

2. Validation

- The validation phase verifies that the consistency properties for the requested isolation level have been met. We'll see more details about validation in a later section, after I have talked about isolation levels. After the isolation levels behavior is validated, In-Memory OLTP may need to wait for any commit dependencies, which will also be described shortly. If the transaction passes the validation phase, after any commit dependencies are gone it is considered really committed. If the any of the modified tables were created with SCHEMA_AND_DATA, the changes will be logged. In-Memory OLTP will read the write-set for the transaction to determine what operations will be logged. There may be waiting for commit dependencies, which are usually very brief, and there may be waiting for the write to the transaction log. Because logging for memory-optimized tables is much more efficient that logging for disk-based tables (as we'll see in in the section on Logging) these waits can also be very short.
-

3. Post-processing

- The post-processing phase is usually the shortest. If the transaction committed, the begin-timestamp in all the inserted records is replaced by the actual timestamp of this transaction and the end-timestamp of all the deleted records is replaced by the actual timestamp. If the transaction failed or was explicitly rolled back, inserted rows will be marked as garbage and deleted rows will have their end-timestamp changed back to infinity.

Isolation Levels

First, let me give you a little background on isolation levels in general. This will not be a complete discussion of isolation levels, which is beyond the scope of this book. Isolation levels can be defined in terms of the consistency properties that are guaranteed for your transactions. The most important properties are the following:

1. Read Stability. If a transaction T reads some version V1 of a record during its processing, to achieve Read Stability we must guarantee that V1 is still the version visible to T as of the end of the transaction; that is, V1 has not been replaced by another committed version V2. Read Stability be enforced either by acquiring a shared lock on V1 to prevent changes or by validating that V1 has not been updated before the transaction is committed.
2. Phantom Avoidance. To achieve Phantom Avoidance we must be able to guarantee that a transaction T's scans would not return additional new versions added between the time T starts and the time T commits. Phantom Avoidance can be enforced in two ways: by locking the

scanned part of an index or table or by rescanning before the transaction is committed to check for new versions.

Once we understand these properties, we can define the transaction isolation levels based on these properties. The first one listed (SNAPSHOT) does not mention these properties, but the second two do.

- SNAPSHOT

  SHAPSHOT isolation level specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction. The transaction can only recognize data modifications that were committed before the start of the transaction. Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction. The statements in a transaction get a snapshot of the committed data as it existed at the start of the transaction. In other words, a transaction running in SNAPSHOT isolation will always see the most recent committed data as of the start of the transaction.

- REPEATABLE READ

  REPEATABLE READ isolation level includes the guarantees given by SNAPSHOT isolation level. In addition, REPEATABLE READ guarantees Read Stability. For any row that is read by the transaction, at the time the transaction commits the row has not been changed by any other transaction. Every read operation in the transaction is repeatable up to the end of the transaction.

- SERIALIZABLE

  SERIALIAZABLE isolation level includes the guarantees given by the REPEATABLE READ isolation level. In addition, SERIALIZABLE guarantees Phantom Avoidance. The operations in the transaction have not missed any rows. No phantom rows have appeared between time of the snapshot and the end of the transaction. Phantom rows match the filter condition of a SELECT/UPDATE/DELETE. A transaction is serializable if we can guarantee that it would see exactly the same data if all its reads were repeated at the end of the transaction.

The simplest and most widely used MVCC (multi version concurrency control) method is SNAPSHOT isolation but SNAPSHOT isolation does not guarantee serializability because reads and writes logically occur at different times, reads at the beginning of the transaction and writes at the end.

Access to disk-based tables also supports READ COMMITTED isolation, which simply guarantees that the transaction will not read any dirty (uncommitted) data. Access to memory-optimized tables needs to use one of the three isolation levels mentioned above. Table 1 lists which isolation levels can be used together in a cross-container transaction. You should also consider that once you have accessed a table in a cross-container transaction using an isolation level HINT, you should continue to use that same hint for all subsequent access of the table. Using different isolation levels for the same table (whether a disk-based table or memory-optimized table) will usually lead to failure of the transaction.

Disk-based tables	Memory-optimized tables	Recommendations

READ COMMITTED	SNAPSHOT	This is the baseline combination and should be used for most situations using READ COMMITTED for disk-based tables.
READ COMMITTED	REPEATABLE READ / SERIALIZABLE	This combination can be used during data migration and for memory-optimized table access in interop mode (not in a natively compiled procedure).
REPEATABLE READ / SERIALIZABLE	SNAPSHOT	The access for memory-optimized tables is only INSERT operations. This combination can also be useful during migration and if no concurrent write operations are being performed on the memory-optimized tables.
SNAPSHOT	-	No memory-optimized table access allowed (see note 1)
REPEATABLE READ / SERIALIZABLE	REPEATABLE READ / SERIALIZABLE	This combination is not allowed (see note 2)

Table 1: Compatible isolation levels in cross-container transactions

Note: For SHAPSHOT Isolation, all operations need to see the versions of the data that existed as of the beginning of the transaction. For SNAPSHOT transactions, the beginning of the transaction is considered to be when the first table is accessed. In a cross-container transaction, however, since the sub-transactions can each start at a different time, another transaction may have changed data between the start times of the two sub-transactions. The cross-container transaction then will have no one point in time that the snapshot is based on.

Note 2: The reason both the sub-transactions (the one on the disk-based tables and the one on the memory-optimized tables) can't use REPEATABLE READ or SERIALZABLE is because the two systems implement the isolation levels in different ways. Imagine you are running the two cross-container transactions in Table 2. RHk# indicates a row in a memory-optimized table, and RSql# indicates a row in a disk-based table. Tx1 would read the row from the memory-optimized table first and no locks would be held, so that Tx2 could complete and change the two rows. When Tx1 resumed, when it read the row from the disk-based table, it would now have a set of values for the two rows that could never have existed if the transaction were run in isolation (i.e. if the transaction were truly serializable.) So this combination is not allowed.

Time	Tx1 (SERIALIZBLE)	Tx2 (any isolation level)
1	BEGIN SQL/In-Memory sub-transactions	
2	Read RHk1	
3		BEGIN SQL/In-Memory sub-transactions
4		Read RSql1 and update to RSql2
5		Read RHk1 and update to RHk2
6		COMMIT
7	Read RSql2	

Table 2 Two concurrent cross-container transactions

For more details on Isolation Levels, please see the following references:

http://en.wikipedia.org/wiki/Isolation_(database_systems)

http://research.microsoft.com/apps/pubs/default.aspx?id=69541

Since SNAPSHOT isolation is probably the most used isolation level with memory-optimized tables, and is the recommended isolation level in most cases, a new database property is available to automatically upgrade the isolation to SHAPSHOT for all operations on memory-optimized tables, if the T-SQL transaction is running in a lower isolation level. Lower levels are READ COMMITTED, which is SQL Server's default, and READ UNCOMMITTED, which is not recommended. The code below shows how you can set this property for the *AdventureWorks2016* database.

```
ALTER DATABASE AdventureWorks2016
SET MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT ON;
```

You can verify whether this option has been set in two ways, as shown below, either by inspecting the *sys.databases* catalog view or by querying the DATABASEPROPERTYEX function.

```
SELECT is_memory_optimized_elevate_to_snapshot_on FROM sys.databases
WHERE name = ' AdventureWorks2016 ';
GO
SELECT DATABASEPROPERTYEX(' AdventureWorks2016 ',
          'IsMemoryOptimizedElevateToSnapshotEnabled');
GO
```

Keep in mind that when this option is set to OFF, almost all of your queries accessing memory-optimized tables will need to use hints to specify the transaction isolation level.

Validation and Post-processing

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.

To help ensure the validation process can be performed efficiently, SQL Server may use the read-set and scan-set, mentioned earlier in this section, to help verify that no inappropriate changes were made. Whether or not a read-set or scan-set must be maintained depends on the transactions' isolation level, as shown in Table 3.

Isolation Level	Read-set	Scan-set
SNAPSHOT	NO	NO
REPEATABLE READ	YES	NO
SERIALIZABLE	YES	YES

Table 3: Changes monitored in the allowed isolation levels

In addition, the read-set and write-set might also be needed if any of the tables modified in the transaction have defined constraints that must be validated.

If memory-optimized tables are accessed in SNAPSHOT isolation, the following validation error is 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, the following error will be generated and the transaction will be aborted.

```
Error 41325: The current transaction failed to commit due to a serializable validation fa
ilure
```

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, the following error will be generated and the transaction will be aborted.

```
Error 41305: The current transaction failed to commit due to a repeatable read
validation failure.
```

The transaction's read-set is used to determine if any of the rows read previously have a new version by the end of the transaction.

Table 4 shows an example of a repeatable read isolation failure.

Time	Transaction Tx1 (REPEATABLE READ)	Transaction Tx2 (any isolation level)
1	BEGIN TRAN	
2	SELECT City FROM Person WHERE Name = 'Jill'	BEGIN TRAN
3		UPDATE Person SET City = 'Madrid' WHERE Name = 'Jill'
4	--- other operations	
5		COMMIT TRAN
6	COMMIT TRAN During validation, error 41305 is generated and Tx1 is rolled back	

Table 4: Transactions resulting in a REPEATABLE READ isolation failure

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 a valid row that meets the specified filter conditions (due to deletions by other transactions), or encounters phantoms rows inserted by other transactions that meet the specified filter conditions, 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 (shown above) is generated and the transaction will be aborted.

The transaction's Scan-set is used to determine if any additional rows now meet the predicate's condition.

Table 5 shows an example of a serializable isolation failure.

Time	Transaction Tx1 (SERIALIZABLE)	Transaction Tx2 (any isolation level)
1	BEGIN TRAN	
2	SELECT Name FROM Person WHERE City = 'Perth'	BEGIN TRAN
3		INSERT INTO Person VALUES ('Charlie', 'Perth')
4	--- other operations	
5		COMMIT TRAN
6	COMMIT TRAN During validation, error 41325 is	

	generated and Tx1 is rolled back

Table 5: Transactions resulting in a SERIALIZABLE isolation failure

Another isolation level violation that can occur is a write-write conflict. However, as mentioned earlier, this error is caught during regular processing and not during the validation phase.

```
Error 41302: The current transaction attempted to update a record in table X that has
been updated since this transaction started.  The transaction was aborted.
```

Validation of Foreign Keys

SQL Server 2016 introduced support for foreign key constraints on memory optimized tables. Validation of the foreign key relationships may introduce unexpected isolation level errors even if you are running in the recommended SNAPSHOT isolation. If you update or delete data from a table which has foreign keys referencing it, the constraint validation has to happen under a higher isolation level in order to ensure that no rows are changed such that the constraint is violated. With disk-based tables there is no problem, because SQL Server will acquire row locks on the tables involved in foreign key constraints, and any concurrent transactions trying to delete or update those rows will be blocked. However, memory optimized tables are lock-free and hence validations at the right isolation level are required to ensure correctness. For more details about this behavior, take a look at this post:

https://blogs.msdn.microsoft.com/sqlcat/2016/03/24/considerations-around-validation-errors-41305-and-41325-on-memory-optimized-tables-with-foreign-keys/

Commit Dependencies

During regular processing, a transaction can read rows written by other transactions that are in the validation or post-processing phases, but have not yet committed. The rows are visible because the logical end time of those other transactions has been assigned at the start of the validation phase. (If the other transactions are not entered their validation phase yet, they are not committed and a concurrent transaction will never be able to see them.)

If a transaction Tx1 reads such uncommitted rows from Tx2, Tx1 will take a commit dependency on Tx2 and increment an internal counter that keeps track of the number of commit dependencies that Tx1 has. In addition, Tx2 will add a pointer to Tx1 to a list of dependent transactions that Tx2 maintains.

Waiting for commit dependencies to clear has two main implications:

- A transaction cannot commit until the transactions it depends on have committed. In other words, it cannot enter the commit phase, until all dependencies have cleared and the internal counter has been decremented to 0.
- In addition, result sets are not returned to the client until all dependencies have cleared. This prevents the client from retrieving uncommitted data.

If any of the dependent transactions fails to commit, there is a commit dependency failure. This means the transaction will fail to commit with the following error:

> Error 41301: A previous transaction that the current transaction took a dependency on has aborted, and the current transaction can no longer commit.

SQL Server 2016 allows a maximum of 8 commit dependencies on a single transaction, and when that number is exceeded, the transaction that tries to take a dependency fails as shown:

> Error 41839: Transaction exceeded the maximum number of commit dependencies and the last statement was aborted. Retry the statement.

The errorlog will show an informational message when this error occurs:

> [INFO] At least 8 incoming dependencies taken on a single transaction.

A new message will be generated each time the number of dependencies attempted on the original transaction hits another power of two, so you would see an information message after 16 attempts, 32 attempts, 64 attempts, etc.

Commit processing

Once all the transactions we are dependent on have been verified and commited, the transaction is logged. Logging will be covered in more detail in a later section. After the logging is completed, the transaction is marked as committed in the global transaction table.

Post-processing

The final phase is the post-processingThe main operations are to update the timestamps of each of the rows inserted or deleted by this transaction.

- For a DELETE operation, set the row's end timestamp to the end timestamp of the transaction and clear the type flag on the row's end timestamp field to indicate the end-timestamp is really a timestamp, and not a transaction-ID .
- For an INSERT operation, set the row's begin timestamp to the end timestamp of the transaction and clear the type flag on the row's begin timestamp field.

The actual unlinking and deletion of old row versions is handled by the garbage collection system, which was discussed earlier.

Finally, SQL Server goes through the linked list of transactions that were dependent on this one, and reduces their dependency counters by 1.

To summarize, let's take a look at the steps taken when processing a transaction Tx1 involving one or more memory-optimized tables, after the all of the Transact-SQL is executed and the COMMIT TRAN statement is encountered:

1. Validate the changes made by Tx1
2. Wait for any commit dependencies to reduce the dependency count to 0.

3. Log the changes.
4. Mark the transaction as committed in the global transaction table.
5. Update the begin-timestamp of inserted rows and the end-timestamp of deleted rows (post-processing).
6. Clear dependencies of transactions that are dependent on Tx1.

The final step of removing any unneeded or inaccessible rows is not always done immediately and may be handled by completely separate threads performing garbage collection.

Concurrency

As mentioned earlier, when accessing memory-optimized tables, SQL Server uses completely optimistic multi-version concurrency control. This does not mean there is never any waiting when working with memory-optimized tables in a multi-user system, but there is never any waiting for locks. The waiting that does occur is usually of very short duration, such as when SQL Server is waiting for dependencies to be resolved during validation, and also when waiting for log writes to complete.

There is some similarity between disk-based tables and memory-optimized tables when performing concurrent data operations. Both do conflict detection when attempting to update to make sure that updates are not lost. But processing modifications to disk-based tables is pessimistic in the sense that if transaction Tx2 attempts to update data that transaction Tx1 has already updated, the system will consider that transaction Tx1 may possibly fail to commit. Tx2 will then wait for Tx1 to complete, and will generate a conflict error only if Tx1 was successfully committed. If Tx1 is rolled back, Tx2 can proceed with no conflict. Because there is an underlying assumption that Tx1 could fail, this is a pessimistic approach. When operating on memory-optimized tables, the assumption is that Tx1 will commit. So the In-Memory OLTP engine will generate the conflict error immediately, without waiting to find out if Tx1 does indeed commit.

In general, you can think of the general strategy in processing operations on memory-optimized tables is that a transaction Tx1 cannot cause other transaction Tx2 to wait, just because Tx2 wants to perform such action such as changing a row's values. It is Tx1 that wants to perform the operation, or wants to guarantee a certain isolation level, that covers the cost. This could take the form of getting an update conflict error or waiting for validation to take place which might result in a validation failure.

Locks

Operations on disk-based tables follow the requested transaction isolation level semantics by using locks to make sure data is not changed by transaction Tx2 while transaction Tx1 needs the data to remain unchanged. In a traditional relational database system in which pages need to be read from disk before they can be processed, the cost of acquiring and managing locks can just be a fraction of the total wait time. Waiting for disk reads, and managing the pages in the buffer pool can be at least as much overhead. But with memory-optimized tables, where there is no cost for reading pages from disk, overhead for lock waits could be a major concern. SQL Server In-Memory OLTP was designed from the beginning to be a totally lock-free system. Existing rows are never modified, so there is no need to lock them. As we've seen, all updates are performed as a two-step process using row versions. The current version of the row is marked as deleted and a new version of the row is created.

Latches

Latches are lightweight synchronization primitives that are used by the SQL Server engine to guarantee consistency of data structures used to manage disk-based tables, including; index and data pages as well as internal structures such as non-leaf pages in a B-Tree. Even though latches are quite a bit lighter weight than locks, there can still be substantial overhead and wait time involved in using latches. A latch must be acquired every time a page is read from disk, to make sure no other process writes the page while it is being read. A latch is acquired on the memory buffer that a page from disk is being read into, to make sure no other process uses that buffer. In addition, SQL Server acquires latches on internal metadata, such as the internal table that keeps track of locks being acquired and released. Since SQL Server In-Memory OLTP doesn't do any reading from disk during data processing, doesn't store data in buffers and doesn't apply any locks, there is no reason that latches are required for operations on memory-optimized tables and one more possible source of waiting and contention is removed.

Spinlocks

Spinlocks are lightweight synchronization primitives which are used to protect access to data structures. Spinlocks are not unique to SQL Server. They are generally used when it is expected that access to a given data structure will need to be held for a very short period of time. When a thread attempting to acquire a spinlock is unable to obtain access it executes in a loop periodically checking to determine if the resource is available instead of immediately yielding. After some period of time a thread waiting on a spinlock will yield before it is able to acquire the resource in order to allow other threads running on the same CPU to execute.

The mechanics and internals of locks, latches and spinlocks is a huge topic, and is really not in the scope of this paper. However, it might be useful just to see a quick summary of the difference between locks and latches, and this is shown in Table 6.

Structure	Purpose	Controlled by	Performance cost
Latch	Guarantee consistency of in-memory structures.	SQL Server engine only.	Performance cost is low. To allow for maximum concurrency and provide maximum performance, latches are held only for the duration of the physical operation on the in-memory structure, unlike locks which are held for the duration of the logical transaction.

Spinlock	Guarantee consistency of in-memory structures.	SQL Server engine only.	Performance cost is low but the CPU is used by the thread while waiting for the spinlock. Used when the expected wait time is less than the cost of a context switch.
Lock	Guarantee consistency of transactions, based on transaction isolation level.	Can be controlled by user.	Performance cost is high relative to latches as locks must be held for the duration of the transaction.

Table 6: Comparing locks, latches and spinlocks

There is a lot of information available about troubleshooting issues with locks, including in my books SQL Server Concurrency: Locking, Blocking and Row Versions and SQL Server 2012 Internals. For more details regarding latching and spinlocks, you can refer to the following documents from Microsoft:

Diagnosing and Resolving Latch Contention on SQL Server

Diagnosing and Resolving Spinlock Contention on SQL Server

Waiting

Although there are no locks, latches or spinlocks used with memory-optimized tables, that does not mean there is never any waiting. As discussed above, if there are commit dependencies affecting any rows returned to the client, SQL Server will wait for dependencies to clear before the rows can be returned. In addition, the processing the actual transaction COMMIT requires interaction with the log manager, and there can be waiting for the log records to be flushed to disk, as well as waiting for secondary replicas to be synchronized. Note that the waiting on the flush to disk is avoided when using delayed durability, and interaction with log manager is completely avoided when using SCHEMA_ONLY tables.

Checkpoint and Recovery

SQL Server must ensure transaction durability for memory-optimized tables, so that changes can be recovered after a restart or a restore. In-Memory OLTP achieves this by having both the checkpoint process and the transaction logging process write to durable storage. Though not covered in this paper, In-Memory OLTP is also integrated with the AlwaysOn Availability Group feature that maintains highly available replicas supporting failover.

The information written to disk consists of checkpoint streams and transaction log streams.

- *Log streams* contain the changes made by committed transactions logged as insertion and deletion of row versions.
- *Checkpoint streams* come in three varieties:
    - *data streams* contain all versions inserted during between two specific timestamp values.

- *delta streams* are associated with a particular data stream and contain a list of integers indicating which row versions in its corresponding data stream have been deleted.
- *large data streams* contain data from the columnstore index compressed rowgroups for memory-optimized tables and for LOB column in memory-optimized tables.

The combined contents of the transaction log and the checkpoint streams are sufficient to recover the in-memory state of memory-optimized tables to a transactionally consistent point in time. Before we go into more detail of how the log and the checkpoint files are generated and used, here are a few crucial points to keep in mind:

- Log streams are stored in the regular SQL Server transaction log.
- Checkpoint streams are stored in SQL Server filestream files which in essence are sequential files fully managed by SQL Server. (Filestream storage was introduced in SQL Server 2008 and In-Memory OLTP checkpoint files take advantage of that technology. For more details about filestream storage and management, see this whitepaper: http://msdn.microsoft.com/library/hh461480.aspx). One big change in SQL 2016 is that in-memory OLTP uses the FileStream filegroup only as a container. File creation and garbage collection is now fully managed by in-memory OLTP engine, one result of which is much more efficient garbage collection of checkpoint files that are no longer needed.
- The transaction log contains enough information about committed transactions to redo the transaction. The changes are recorded as inserts and deletes of row versions marked with the table they belong to. The transaction log stream for the changes to memory-optimized tables is generated at the transaction commit time. This is different than disk-based tables where each change is logged at the time of operation irrespective of the final outcome of the transaction. No undo information is written to the transaction log.
- Index operations on memory-optimized tables are not logged. With the exception of compressed segments for columnstore indexes on memory-optimized tables, all indexes are completely rebuilt on recovery.

Transaction Logging

In-Memory OLTP's transaction logging is designed for both scalability and high performance. Each transaction is logged in a minimal number of potentially large log records that are written to SQL Server's regular transaction log. The log records contain information about all versions inserted and deleted by the transaction. Using this information, the transaction can be redone during recovery.

For In-Memory OLTP transactions, log records are generated only at commit time. In-Memory OLTP does not use a write-ahead logging (WAL) protocol, such as used when processing operations on disk-based tables. With WAL, SQL Server writes to the log before writing any changed data to disk, and this can happen even for uncommitted data written out during checkpoint. For In-Memory OLTP, dirty data is never written to disk. Furthermore, In-Memory OLTP groups multiple data changes into one log record to minimize the overhead both for the overall size of the log and reducing the number of writes to log buffer. Not using WAL is one of the factors that allows In-Memory OLTP commit processing to be extremely efficient.

The following simple script illustrates the greatly reduced logging for memory-optimized tables. This script creates a database that can hold memory-optimized tables, and then creates two similar tables. One is a memory-optimized table, and one is a disk-based table.

```
USE master
GO
DROP DATABASE IF EXISTS LoggingTest;
GO
CREATE DATABASE LoggingTest ON
 PRIMARY (NAME = LoggingTest_data, FILENAME = 'C:\HKdata\LoggingTest_data.mdf'),
 FILEGROUP LoggingTest_FG CONTAINS MEMORY_OPTIMIZED_DATA
 (NAME = LoggingTest_mod, FILENAME = 'C:\HKdata\ LoggingTest_mod')
 LOG ON (name = LoggingTest_log, Filename='C:\HKdata\LoggingTest.ldf', size=100MB);
GO
USE LoggingTest;
GO

DROP TABLE IF EXISTS dbo.t1_inmem
GO
-- create a simple memory-optimized table
CREATE TABLE dbo.t1_inmem
( c1 int NOT NULL,
  c2 char(100) NOT NULL,
  CONSTRAINT pk_index91 PRIMARY KEY NONCLUSTERED (c1)
) WITH (MEMORY_OPTIMIZED = ON,
 DURABILITY = SCHEMA_AND_DATA);
GO
DROP TABLE IF EXISTS dbo.t1_disk
GO
-- create a similar disk-based table
CREATE TABLE dbo.t1_disk
( c1 int NOT NULL PRIMARY KEY NONCLUSTERED,
  c2 char(100) NOT NULL)
GO
```

Next, populate the disk-based table with 100 rows, and examine the contents of the transaction log using the undocumented (and unsupported) function *fn_dblog*().

```
BEGIN TRAN
DECLARE @i int = 0
WHILE (@i < 100)
BEGIN
  INSERT INTO t1_disk VALUES (@i, replicate ('1', 100))
  SET @i = @i + 1
END
COMMIT

-- you will see that SQL Server logged 200 log records
SELECT * FROM sys.fn_dblog(NULL, NULL)
WHERE PartitionId IN
     (SELECT partition_id FROM sys.partitions
      WHERE object_id=object_id('t1_disk'))
ORDER BY [Current LSN] ASC;
GO
```

Now run a similar update on the memory-optimized table

```
BEGIN TRAN
DECLARE @i int = 0
WHILE (@i < 100)
BEGIN
  INSERT INTO t1_inmem VALUES (@i, replicate ('1', 100))
  SET @i = @i + 1
END
COMMIT
```

We can't filter based on the *partition_id*, as a single log record access rows from multiple tables or partitions, so we just look at the most recent log records. We should see the three most recent records looking similar Figure 23.

```
-- look at the log
SELECT * FROM sys.fn_dblog(NULL, NULL) order by [Current LSN] DESC;
GO
```

	Current LSN	Operation	Context	Transaction ID	LogBl
1	00000020:00000157:0006	LOP_COMMIT_XACT	LCX_NULL	0000:00000323	0
2	00000020:00000157:0005	LOP_HK	LCX_NULL	0000:00000323	0
3	00000020:00000157:0004	LOP_BEGIN_XACT	LCX_NULL	0000:00000323	0

Figure 23 SQL Server transaction log showing one log record for 100 row transaction

All 100 inserts have been logged in a single log record, of type LOP_HK. LOP indicates a 'logical operation' and HK is an artifact from the project codename, Hekaton. Another undocumented, unsupported function can be used to break apart a LOP_HK record. You'll need to replace the *Current LSN* value with the LSN that the results show for your LOP_HK record.

```
SELECT [Current LSN], [Transaction ID], Operation,
       operation_desc, tx_end_timestamp, total_size
FROM sys.fn_dblog_xtp(null, null)
WHERE [Current LSN] = '00000020:00000157:0005';
```

The first few rows of output should look like Figure 24.

Current LSN	Transaction ID	Operation	operation_desc	tx_end_timestamp	total_size
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_BEGIN_TX	19	17
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119
00000021:00000294:0002	0000:00000335	LOP_HK	HK_LOP_INSERT_ROW	19	119

Figure 24 Breaking apart the log record for the inserts on the memory-optimized table shows the individual rows affected

The single log record for the entire transaction on the memory-optimized table, plus the reduced size of the logged information, helps to make transactions on memory-optimized tables much more efficient.

Checkpoint

Just like for operations on disk-based tables, the main reasons for checkpoint operations are to reduce recovery time and to keep the active portion of the transaction log as small as possible. The checkpoint process for memory-optimized tables allows the data from DURABLE tables to be written to disk so it can be available for recovery. The data on disk is never read during query processing, it is ONLY on disk to be

used when restarting your SQL Server. The checkpoint process is designed to satisfy two important requirements.

- **Continuous checkpointing.** Checkpoint related I/O operations occur incrementally and continuously as transactional activity accumulates. Hyper-active checkpoint schemes (defined as checkpoint processes which sleep for a while after which they wake up and work as hard as possible to finish up the accumulated work) can potentially be disruptive to overall system performance.
- **Streaming I/O.** Checkpointing for memory-optimized tables relies on streaming I/O rather than random I/O for most of its operations. Even on SSD devices random I/O is slower than sequential I/O and can incur more CPU overhead due to smaller individual I/O requests. In addition, SQL Server 2016 now can read the log in parallel using multiple *serializers*, which will be discussed in the section below on the Checkpoint Process.

A checkpoint event for memory-optimized tables is invoked in these situations:

- Manual checkpoint – an explicit checkpoint command initiates checkpoint operations on both disk-based tables and memory-optimized tables.
- Automatic checkpoint – SQL Server runs the in-memory OLTP checkpoint when the size of the log has grown by about 1.5 GB since the last checkpoint. (This is increased from 512 MB in SQL Server 2014.) Note that this is not dependent on the amount of work done on memory-optimized tables, only the size of the transaction log, which contains all log records for changes to durable memory-optimized tables and to disk-based tables. It's possible that there have been no transactions on memory-optimized tables when a checkpoint event occurs.

Because the checkpoint process is continuous, the checkpoint event for memory-optimized tables does not have the job of writing all the all the changed data to disk, as happens when a checkpoint is initiated for disk-based tables. The main job of the checkpoint event on memory-optimized tables is to create a new root file and manage the states of certain other files, as are described in the next section.

Checkpoint Files

Checkpoint data is stored in four types of checkpoint files. The main types are DATA files and DELTA files. There are also LARGE OBJECT files and ROOT files. Files that are pre-created are unused, so they aren't storing any checkpoint data. These files have a type of **FREE** before they are used as one of the four types listed below. A **DATA file** contains only inserted versions of rows, which, as we saw earlier are generated by both INSERT and UPDATE operations. Each file covers a specific timestamp range. All versions with a begin timestamp within the data file's range are contained in the file. Data files are append-only while they are open and once they are closed, they are strictly read-only. At recovery time the active versions in the data files are reloaded into memory. The indexes are then rebuilt as the rows are being inserted during the recovery process.

A **DELTA file** stores information about which versions contained in a data file have been subsequently deleted. There is a 1:1 correspondence between delta files and data files. Delta files are append-only for the lifetime of the data file they correspond to. At recovery time, the delta file is used as a filter to avoid reloading deleted versions into memory. Because each data file is paired with exactly one delta file, the

smallest unit of work for recovery is a data/delta file pair. This allows the recovery process to be highly parallelizable.

A LARGE DATA file stores your large column (LOB) values or the contents of one rowgroup for a columnstore index. If you have no large columns or columnstore indexes, there will be several PRECREATED data files of the size usually used for LARGE DATA, but they will stay in the FREE state.

A **ROOT file** keeps track of the files generated for each checkpoint event, and a new ACTIVE root file is created each time a checkpoint event occurs.

As mentioned, the DATA and DELTA files are the main types, because they contain the information about all the transactional activity against memory-optimized tables. Because the DATA and DELTA files always exist in a 1:1 relationship (once they actually contain data), a DATA file and its corresponding DELTA file are sometimes referred to as a checkpoint file pair or CFP.

As soon as your first memory-optimized table is created, SQL Server will create 16 FREE files of various sizes. The sizes are always a power of two megabytes, from 8MB to 1GB. When a file of one of the other types in this list is needed, SQL Server takes one of the FREE files that is as least as big as the size needed. If a file bigger than 1GB is required, SQL Server must use a 1GB file and enlarge it to the appropriate size.

Initially SQL Server will create at least three FREE files of sufficient size for each of the four types mentioned above, plus a few 1 MB files that can grow as needed. The initial sizes for each of the file types depend on the specification of the computer that your SQL Server instance is running on. A machine with greater than 16 cores and 128 GB of memory is considered high-end. We can also consider a system as low-end if it has less than 16GB memory. Table 7 shows the sizes of the FREE files created for each of the four file types, depending on the size of your machine. (Note that although the sizes are determined in anticipation of being used for a particular type of file, there is no actual requirement that a FREE file be used for a specific purpose.)

	DATA	DELTA	LARGE DATA	ROOT
High-end	1GB	128MB	64MB	16MB
Standard	128MB	8MB	64MB	16MB
Low-end	16MB	8MB	8MB	8MB

Table 7 Initial sizes of the FREE files created to be used for the various file types

For SQL Server 2016, the checkpoint files can be in one of the following states:

- **PRECREATED**: The 12 FREE files that are created will be in the PRECREATED state. So this gives us a fixed minimum storage requirement in databases with memory-optimized tables. SQL Server will convert PRECREATED files to UNDER CONSTRUCTION or MERGE TARGET as those files are needed. SQL Server will create new PRECREATED files when a CHECKPOINT operation occurs, if there are not 3-5 existing PRECREATED files.
- **UNDER CONSTRUCTION**: These files are being actively filled by inserted rows and the RowIDs of deleted rows. In the event of a system shutdown or crash, these are the rows that will be recovered by applying the changes from the transaction log.

- **ACTIVE**: These files contain the inserted rows and RowIDs of deleted rows for the last checkpoint event. These are the rows that will need to be recovered after a system crash, before applying the active part of the transaction log.
- **MERGE TARGET**: These files are in the process of being constructed by merging ACTIVE files that have adjacent transaction ranges. Since these files are in the process of being constructed and they duplicate information in the ACTIVE files, they will not be used for crash recovery Once the merge operation is complete, the MERGE TARGET files will become ACTIVE
- **WAITING FOR LOG TRUNCATION**: Once a merge operation is complete, the old ACTIVE files, which were the source of the MERGE operation, will transition to WAITING FOR LOG TRUNCATION. CFPs in this state are needed for the operational correctness of a database with memory-optimized tables. For example, these files would be needed to recover from a durable checkpoint to restore to a previous point in time during a restore. A CFP can be marked for garbage collection once the log truncation point moves beyond its transaction range.

As described, files can transition from one state to another, but only a limited number of transitions are possible. Figure 25 shows the possible transitions.

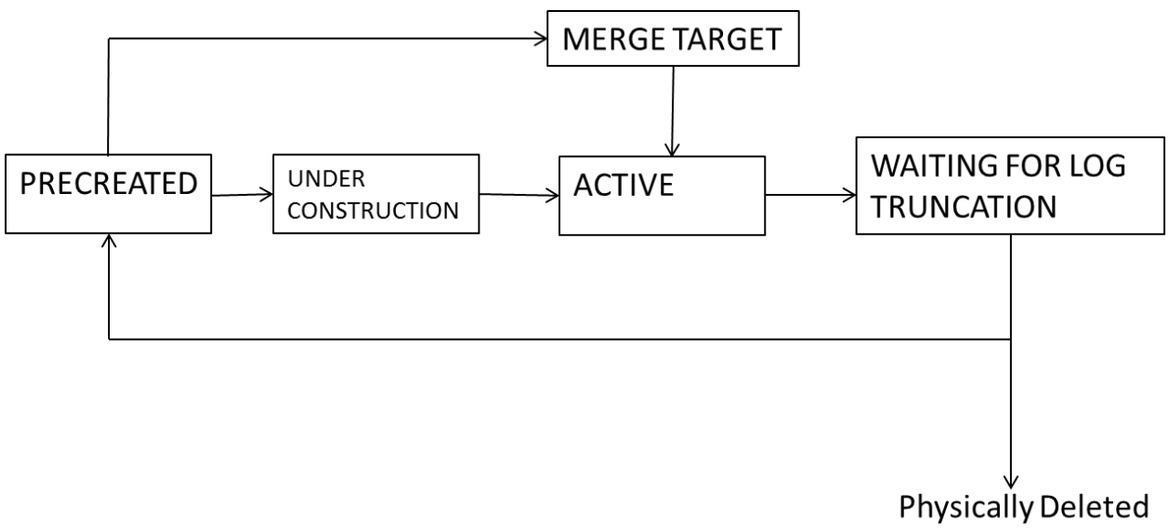

Figure 25 Possible state transitions for checkpoint files in SQL Server 2016

The main transitions are the following:

- PRECREATED to UNDER CONSTRUCTION – A PRECREATED file is converted to UNDER CONSTRUCTION when a new file is needed after a checkpoint closes the previous UNDER CONSTRUCTION file, or the previous UNDER CONSTRUCTION file is filled.
- UNDER CONSTRUCTION to ACTIVE – An UNDER CONSTRUCTION file is converted to ACTIVE when a checkpoint event occurs and closes the current UNDER CONSTRUCTION file.
- PRECREATED to MERGE TARGET– A PRECREATED file is converted to MERGE TARGET when a new file is needed to contain the results of a file merge operation.
- MERGE TARGET to ACTIVE – When a merge operation is complete, the MERGE TARGET file takes over the role of merged ACTIVE files and hence it becomes active.

- ACTIVE to WAITING FOR LOG TRUNCATION – When a merge operation is complete, the ACTIVE files that are the source of the merge operation will transition from ACTIVE to WAITING FOR LOG TRUNCATION as they are no longer needed for crash recovery. Root and large object files can be transitioned to WAITING FOR LOG TRUNCATION without the need for a merge operation.
- WAITING FOR LOG TRUNCATION to Physically deleted – Once the log truncation point moves beyond the highest transaction ID in a data file, the file is no longer needed and can be recycled. If there are already 3-5 PRECREATED files of the same type, the file can be deleted.
- WAITING FOR LOG TRUNCATION to PRECREATED - Once the log truncation point moves beyond the highest transaction ID in a data file, the file is no longer needed and can be recycled. If there are NOT 3-5 PRECREATED files of the same type, the file gets moved to the free pool and its state changes to PRECREATED.

Let's look at some code that can provide some information about the checkpoint files and their state transitions. First we'll create (or recreate) a database.

```
USE master;
GO
DROP DATABASE IF EXISTS IMDB2;
GO
CREATE DATABASE IMDB2 ON
 PRIMARY (NAME = IMDB2_data, FILENAME = 'C:\HKData\IMDB2_data.mdf'),
FILEGROUP IMDB2_mod_FG CONTAINS MEMORY_OPTIMIZED_DATA
(NAME = IMDB2_mod,  FILENAME = 'C:\HKData\IMDB2_mod')
  LOG ON (name = IMDB2_log, Filename='C:\HKData\IMDB2_log.ldf', size=100MB);
GO
ALTER DATABASE IMDB2 SET RECOVERY FULL;
GO
```

At this point, you might want to look in your folder containing the memory-optimized data files. In my example, it's the folder called C:\HKData\IMDB2_mod. Note that in my database creation script, that is the name I gave for the file, but for memory-optimized tables, the value supplied for FILENAME is used for a folder name. Within that folder is a folder called $FSLOG and one called $HKv2. Open the $HKv2 folder, and it will be empty until you create a memory-optimized table, as you can do by running the code below:

```
USE IMDB;
GO
-- create a memory-optimized table with each row of size > 8KB
CREATE TABLE dbo.t_memopt (
       c1 int NOT NULL,
       c2 char(40) NOT NULL,
       c3 char(8000) NOT NULL,

       CONSTRAINT [pk_t_memopt_c1] PRIMARY KEY NONCLUSTERED HASH (c1)
        WITH (BUCKET_COUNT = 100000)
) WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
GO
```

At this point, you can look in the folder that was empty, and see that it now has 17 files in it. My folder contents are shown in Figure 26.

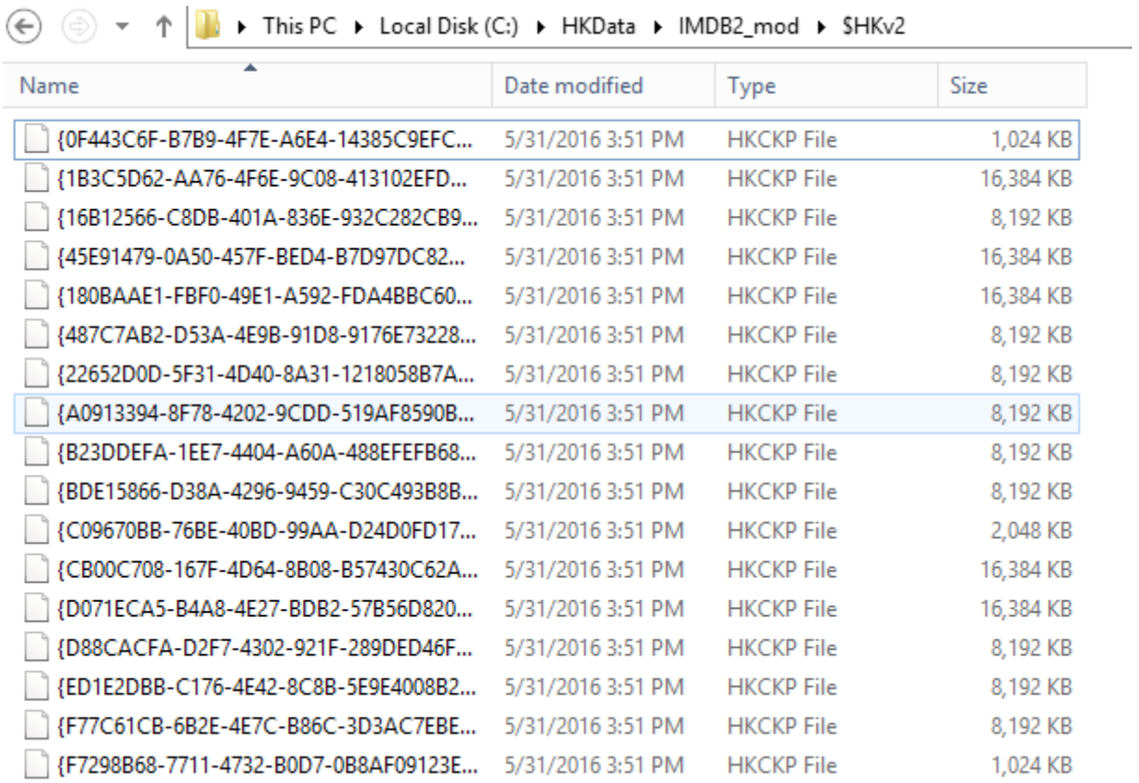

Figure 26 Files on disk after creating a table

You can refer back to the description of PRECREATED files to see what type of file is indicated by each of the sizes, or you can run the query below accessing the DMV *sys.dm_db_xtp_checkpoint_files*.

```
SELECT checkpoint_file_id, file_type_desc, state_desc,
       file_size_in_bytes/1024/1024 AS size_in_MB,
       relative_file_path
FROM sys.dm_db_xtp_checkpoint_files
ORDER BY file_type_desc;
GO
```

My results are shown in Figure 27. There are 3 larger FREE PRECREATED files and 9 smaller ones. One ROOT file is the only one that has the state ACTIVE. Note that there were 17 files on disk but only 13 in the DMV. Some of the files are being cached and in the RTM version of SQL Server 2016, they do not show up in DMV. In later builds you may see a file in the output for each of the files on disk.

checkpoint_file_id	file_type_desc	state_desc	size_in_MB	relative_file_path
D88CACFA-D2F7-4302-921F-289DED46F4EC	FREE	PRECREATED	8	$HKv2\{D88CACFA-D2F7-4302-921F-289DED46F4EC}.hkckp
F77C61CB-6B2E-4E7C-B86C-3D3AC7EBEB00	FREE	PRECREATED	8	$HKv2\{F77C61CB-6B2E-4E7C-B86C-3D3AC7EBEB00}.hkckp
487C7AB2-D53A-4E9B-91D8-9176E732289F	FREE	PRECREATED	8	$HKv2\{487C7AB2-D53A-4E9B-91D8-9176E732289F}.hkckp
B23DDEFA-1EE7-4404-A60A-488EFEFB6828	FREE	PRECREATED	8	$HKv2\{B23DDEFA-1EE7-4404-A60A-488EFEFB6828}.hkckp
BDE15866-D38A-4296-9459-C30C493B8B25	FREE	PRECREATED	8	$HKv2\{BDE15866-D38A-4296-9459-C30C493B8B25}.hkckp
1B3C5D62-AA76-4F6E-9C08-413102EFDA8E	FREE	PRECREATED	16	$HKv2\{1B3C5D62-AA76-4F6E-9C08-413102EFDA8E}.hkckp
CB00C708-167F-4D64-8B08-B57430C62A00	FREE	PRECREATED	16	$HKv2\{CB00C708-167F-4D64-8B08-B57430C62A00}.hkckp
22652D0D-5F31-4D40-8A31-1218058B7A27	FREE	PRECREATED	8	$HKv2\{22652D0D-5F31-4D40-8A31-1218058B7A27}.hkckp
16B12566-C8DB-401A-836E-932C282CB9CB	FREE	PRECREATED	8	$HKv2\{16B12566-C8DB-401A-836E-932C282CB9CB}.hkckp
A0913394-8F78-4202-9CDD-519AF8590B4E	FREE	PRECREATED	8	$HKv2\{A0913394-8F78-4202-9CDD-519AF8590B4E}.hkckp
ED1E2DBB-C176-4E42-8C8B-5E9E4008B298	FREE	PRECREATED	8	$HKv2\{ED1E2DBB-C176-4E42-8C8B-5E9E4008B298}.hkckp
D071ECA5-B4A8-4E27-BDB2-57B56D820E73	FREE	PRECREATED	16	$HKv2\{D071ECA5-B4A8-4E27-BDB2-57B56D820E73}.hkckp
C09670BB-76BE-40BD-99AA-D24D0FD177FE	ROOT	ACTIVE	2	$HKv2\{C09670BB-76BE-40BD-99AA-D24D0FD177FE}.hkckp

Figure 27 The 13 checkpoint files after creating a memory-optimized table

You might also notice that the values in the column *relative_file_path* are the actual path names of files in the *C:\HKData\IMDB2_mod* folder. In addition, the file component of the path, once the brackets are removed, is the *checkpoint_file_id*.

If we had created multiple containers in the filegroup containing the memory-optimized tables, there would be multiple folders under C:\HKData, and the checkpoint files would be spread across them. SQL Server 2016 uses a round-robin allocation algorithm for each type of file (DATA, DELTA, LARGE OBJECT and ROOT). Thus each container contains all types of files.

Multiple containers can be used as a way to parallelize data load. Basically, if creating a second container reduces data load time (most likely because it is on a separate hard drive), then use it. If the second container does not speed up data transfer (because it is just another directory on the same hard drive), then don't do it. The basic recommendation is to create one container per spindle (or I/O bus).

Let's now put some rows into the table and backup the database (so that we can make log backups later:

```
-- INSERT 8000 rows.
-- This should load 5 16MB data files on a machine with <= 16GB of memory.
SET NOCOUNT ON;
DECLARE @i int = 0
WHILE (@i < 8000)
BEGIN
        INSERT t_memopt VALUES (@i, 'a', REPLICATE ('b', 8000))
        SET @i += 1;
END;
GO
BACKUP DATABASE IMDB2 TO  DISK = N'C:\HKBackups\IMDB2-populate-table.bak'
    WITH NOFORMAT, INIT,  NAME = N'IMDB2-Full Database Backup', SKIP, NOREWIND,
       NOUNLOAD,  STATS = 10;
GO
```

Take a look in the folder again, and you should see 16 new files. This is because we needed seven DATA files to hold all the 8000 rows of 8K each, and each of those DATA files has a corresponding DELTA file. . Two new PRECREATED files were also created.

Now let's look at the metadata in a little more detail. As seen above, the DMV *sys.dm_db_xtp_checkpoint_files,* has one row for each file, along with property information for each file. We can use the following query to look at a few of the columns in the view.

```
SELECT file_type_desc, state_desc, internal_storage_slot,
```

```
       file_size_in_bytes/1024/1024 AS size_in_MB,,
       logical_row_count,
       lower_bound_tsn, upper_bound_tsn,
       checkpoint_file_id, relative_file_path
FROM sys.dm_db_xtp_checkpoint_files
ORDER BY file_type_desc;
GO
```

The first few columns of the rows for the DATA and DELTA files are shown in Figure 28, and include the following:

- *file_type_desc*

  This value is one of DATA, DELTA, LARGE OBJECT or ROOT.

- *state_desc*

  This value is one of the state values listed above.

- *internal_storage_slot*

  This value is the pointer to the internal storage array described below, but is not populated until a file becomes ACTIVE.

- *file_size_in_bytes*

  Note that we have just fixed sizes so far, the same as the PRECREATED sizes; the DATA files are 16777216 bytes (16 MB) and the DELTA files are 1048576 bytes(1 MB).

- *logical_row_count*

  This column contains either the number of inserted rows contained in the file (for DATA files) or the number of deleted row id contained in the file (for DELTA files).

- *lower_bound_tsn*

  This is the timestamp for the earliest transaction covered by this checkpoint file.

- *upper_bound_tsn*

  This is the timestamp for the last transaction covered by this checkpoint file.

As will be discussed more in the next section, when a checkpoint event occurs, the UNDER CONSTRUCTION files will be closed, and 1 or more new files will be opened (converted from the PRECREATED files) to store data. We can query *sys.dm_db_xtp_checkpoint_stats* DMV and look at the column *last_closed_checkpoint_ts*._Akk the UNDER CONSTRUCTION files have an *upper_bound_tsn* greater than this value and are considered open. At this point, when I query *sys.dm_db_xtp_checkpoint_stats*, the *last_closed_checkpoint_ts* is 0. None of the files are closed because there has been no checkpoint event yet, even though (because of the continuous checkpoint) there are 7990 rows in the files. (Five files have 1179 rows, one has 1178 and one has 917.) So most of the 8000 rows that I just inserted have been

written on to the checkpoint files. However, if SQL Server needed to recover this table's data at this point, it would do it completely from the transaction log.

file_type_desc	state_desc	internal_storage_slot	size_in_MB	logical_row_count	lower_bound_tsn	upper_bound_tsn	checkpoint_file_id
DATA	UNDER CONSTRUCTION	5	16	1179	5896	7075	9E7CAEBA-080D-4237
DATA	UNDER CONSTRUCTION	3	16	1179	3538	4717	53252E3C-8042-4134-
DATA	UNDER CONSTRUCTION	6	16	917	7075	7992	536446D7-1B59-4D8E-
DATA	UNDER CONSTRUCTION	0	16	1178	0	1180	638228D6-5A1A-47BB-
DATA	UNDER CONSTRUCTION	2	16	1179	2359	3538	1B9A2051-2023-4287-
DATA	UNDER CONSTRUCTION	4	16	1179	4717	5896	4E77C9B7-5C6C-4D36
DATA	UNDER CONSTRUCTION	1	16	1179	1180	2359	C0663261-18CE-416E-
DELTA	UNDER CONSTRUCTION	0	1	0	0	1180	13B11853-0FA1-4343-
DELTA	UNDER CONSTRUCTION	5	1	0	5896	7075	5B7DB5DF-D651-4807-
DELTA	UNDER CONSTRUCTION	1	1	0	1180	2359	294ACBB2-0AD9-4B0C
DELTA	UNDER CONSTRUCTION	6	1	0	7075	7992	524F9DEB-CD1E-4EF5
DELTA	UNDER CONSTRUCTION	4	1	0	4717	5896	0CF32A8A-1434-4618-
DELTA	UNDER CONSTRUCTION	3	1	0	3538	4717	6AC2B423-3DA4-4CE7
DELTA	UNDER CONSTRUCTION	2	1	0	2359	3538	6E1B02E7-1798-4121-
FREE	PRECREATED	NULL	8	NULL	NULL	NULL	4F600C6F-8A45-4205-
FREE	PRECREATED	NULL	16	NULL	NULL	NULL	8AA42002-AC95-4605-
FREE	PRECREATED	NULL	8	NULL	NULL	NULL	5F878FAD-72ED-4D98
FREE	PRECREATED	NULL	16	NULL	NULL	NULL	77529720-683D-407A-
FREE	PRECREATED	NULL	8	NULL	NULL	NULL	92CC5120-C5C9-4EF5
FREE	PRECREATED	NULL	8	NULL	NULL	NULL	75586ADE-8ACD-4608

Figure 28 Checkpoint files after inserted 8000 rows

But now, if we actually execute the CHECKPOINT command in this database, you'll have one or more closed checkpoint files. To observe this behavior, I run the following code:

```
CHECKPOINT
GO
select sum(logical_row_count), max(upper_bound_tsn)
FROM sys.dm_db_xtp_checkpoint_files;
GO
SELECT last_closed_checkpoint_ts
FROM sys.dm_db_xtp_checkpoint_stats;
GO
```

After executing CHECKPOINT in this example, I see a *last_closed_checkpoint_ts* value of 8003 in *sys.dm_db_xtp_checkpoint_stats*, which is also now the highest value in any of the files for *upper_bound_tsn*. I also see that there are now 8000 rows in the files, as all the inserted rows have now been written. So all of the files DATA and DELTA files are now closed. The only change to the number of files is that a new ROOT file was created. Subsequent CHECKPOINT commands will each create a new ROOT file, and new DATA and DELTA files containing 0 rows and increasing values for *upper_bound_tsn*.

The metadata of all checkpoint file pairs that exist on disk is stored in an internal array structure referred to as the Storage *Array* in which. each entry refers to a CFP. As of SQL Server 2016 the number of entries (or slots) in the array is dynamic. The entries in the storage array are ordered by timestamp and as mentioned, each CFP contains transactions in a particular range. The CFPs referenced by the storage array (along with the tail of the log) represent all the on-disk information required to recover the memory-optimized tables in a database. The *internal_storage_slot* value in the *sys.dm_db_xtp_checkpoint_files* DMV refers to a location in the storage array.

The Checkpoint Process

As of SQL Server 2016, the checkpoint process for memory-optimized tables is run multithreaded, with a separate thread for each NUMA node, which is much more efficient thatn the single-threaded checkpoint process used in SQL Server 2014. The checkpoint process is actually comprised of various threads and tasks for different purposes. These are described below:

Controller thread

This thread scans the transaction log to find ranges of transactions that can be given to sub-threads, called 'serializers'. Each range of transactions is referred to as a 'segment'. Segments are identified by a special *segment log record* which has information about the range of transactions within the segment. When the controller sees such a log record, the referenced segment is assigned to a serializer thread.

Segment Generation

A set of transactions is grouped into a segment when a user transaction (with transaction_id T) generates log records that cause the log to grow and cross the 1MB boundary from the end of the previous segment end point. The user transaction T will then close the segment. Any transactions with a transaction_id less than T will continue to be associated with this segment and their log records will be part of the segment. The last transaction will write the segment log record when it completes. Newer transactions, with a transaction_id greater than T, will be associated with subsequent segments.

Serializer Threads

As each segment log record, representing a unique transaction range, is encountered by the Controller Thread, it is assigned to a different serializer thread. The serializer thread processes all the log records in the segment, writing all the inserted and deleted row information to the data and delta files.

Timer Task

A special Timer Task wakes up at regular intervals to check if the active log has exceeded 1.5GB since the last checkpoint event. If so, an internal transaction is created which closes the current open segment in the system and marks it as a special segment that should close a checkpoint. Once all the transactions associated with the segment have completed, the segment definition log record is written. When this special segment is processed by the controller thread, it wakes up a 'close thread'.

Close Thread

The Close Thread generates the actual checkpoint event by generating a new root file which contains information about all the files that are active at the time of the checkpoint. This operation is referred to a 'closing the checkpoint'.

Unlike a checkpoint on disk-based tables, where we can think of a checkpoint as the single operation of writing all dirty data to disk, the checkpoint process for memory-optimized tables is actually a set of processes that work together to make sure your data is recoverable, but that it also can be processed extremely efficiently.

Merging Checkpoint Files

The set of checkpoint files that SQL Server manages for an In-memory OLTP-enabled database can grow with each checkpoint operation. However, the active content of a data file decreases as more and more of its versions are marked as deleted in the corresponding delta file. Since the recovery process will read the contents of all data and delta files, performance of crash recovery degrades as the relevant number of rows in each data file decreases.

The solution to this problem is for SQL Server to *merge* data files that have adjacent timestamp ranges, when their active content (the percentage of undeleted versions in a data file) drops below a threshold. Merging two data files DF1 and DF2 results in a new data file DF3 covering the combined range of DF1 and DF2. All deleted versions identified in the delta files for DF1 and DF2 are removed during the merge. The delta file for DF3 is empty immediately after the merge, except for deletions that occurred after the merge operation started.

Merging can also occur when two adjacent data files are each less than 50% full. Data files can end up only partially full if a manual checkpoint has been run, which closes the currently open checkpoint data file and starts a new one.

Automatic Merge

To identify files to be merged, a background task periodically looks at all active data/delta file pairs and identifies zero more sets of files that qualify. Each set can contain two or more data/delta file pairs that are adjacent to each other such that the resultant set of rows can still fit in a single data file of size 128MB. Figure 29 shows are some examples of files that will be chosen to be merged under the merge policy.

Adjacent Source Files (%full)	Merge Selection
DF0 (30%) DF1 (50%), DF2 (50%), DF3 (90%)	(DF1, DF2)
DF0 (30%) DF1 (20%), DF2 (50%), DF3 (10%)	(DF0, DF1, DF2). Files are chosen starting from left
DF0 (80%), DF1 (10%), DF2 (10%), DF3(20%)	(DF0, DF1, DF2). Files are chosen starting from left

Figure 29 Examples of files that can be chosen for file merge operations

It is possible that two adjacent data files are 60% full. They will not be merged and 40% of storage is unused. So effectively, the total disk storage used for durable memory-optimized tables is larger than the corresponding memory-optimized size. In the worst case, the size of storage space taken by durable tables could be two times larger than the corresponding memory-optimized size.

Garbage Collection of Checkpoint Files

Once the merge operation is complete, the source files are not needed and their state changes to WAITING FOR LOG TRUNCATION. These files can then be removed by a garbage collection process as

long as the log is being regularly truncated. Truncation will happen if regular log backups are taken, or, if the database is in auto_truncate mode. Before a checkpoint file can be removed, the In-Memory OLTP engine must ensure that it will not be further required. The garbage collection process is automatic, and does not require any intervention.

Recovery

Recovery of In-Memory OLTP tables, during a database or instance restart, or as part of a RESTORE operation, starts after the location of the most recent checkpoint file inventory has been determined by reading the most recent ROOT file. SQL Server recovery of disk-based tables and In-Memory OLTP recovery proceed in parallel.

In-Memory OLTP recovery itself is parallelized. Each delta file represents a filter for rows that need not be loaded from the corresponding data file. This data/delta file pair arrangement means that data loading can proceed in parallel across multiple IO streams with each stream processing a single data file and delta file. SQL Server uses one thread per container to create a delta-map for each delta file in that container. Once the delta maps are created, SQL Server streams data files across all cores, with the contents of each data file being filtered through the data map so that deleted rows are not reinserted. This means that if you have 64 cores and 4 containers, 4 threads will be used to create the delta maps but the data file streaming will be done by 64 cores.

Finally, once the checkpoint file load process completes, the tail of the transaction log is read, starting from the timestamp of the last checkpoint, and the INSERT and DELETE operations are reapplied. As of SQL Server 2016, this process of reading and reapplying the logged operations is performed in parallel. After all the transactions are reapplied, the database will be the state that existed at the time the server stopped, or the time the backup was made.

Native Compilation of Tables and Native Modules

In-Memory OLTP provides the ability to natively compile modules that access memory-optimized tables, including stored procedures, views and inline table-valued functions. In fact, In-memory OLTP also natively compiles memory-optimized tables themselves. Native compilation allows faster data access and more efficient query execution than traditional interpreted Transact-SQL provides.

What is native compilation?

Native compilation refers to the process of converting programming constructs to native code, consisting of processor instructions that can be executed directly by the CPU, without the need for further compilation or interpretation.

The Transact-SQL language consists of high-level constructs such as CREATE TABLE and SELECT … FROM. The In-Memory OLTP compiler takes these constructs, and compiles them down to native code for fast

runtime data access and query execution. The In-Memory OLTP compiler takes the table and module definitions as input. It generates C code, and leverages the Visual C compiler to generate the native code.

The result of the compilation of tables and modules are DLLs that are loaded in memory and linked into the SQL Server process.

SQL Server compiles both memory-optimized tables and natively compiled modules to native DLLs at the time the object is created. In addition, the table and module DLLs are recompiled after database or server restart. The information necessary to recreate the DLLs is stored in the database metadata; the DLLs themselves are not part of the database. Thus, for example, the DLLs are not part of database backups.

Maintenance of DLLs

The DLLs for memory optimized tables and natively compiled modules are stored in the filesystem, along with other generated files, which are kept for troubleshooting and supportability purposes.

The following query shows all table and module DLLs currently loaded in memory on the server:

```
SELECT name, description FROM sys.dm_os_loaded_modules
WHERE description = 'XTP Native DLL'
```

Database administrators do not need to maintain the files that are generated by native compilation. SQL Server automatically removes generated files that are no longer needed, for example on table and module deletion, when the database is dropped, and also on server or database restart.

Native compilation of tables

Creating a memory optimized table using the CREATE TABLE statement results in the table information being written to the database metadata, table and index structures being created in memory, and also the table being compiled to a DLL.

Consider the following sample script, which creates a database and a single memory optimized table:

```
USE master
GO
DROP DATABASE IF EXISTS IMDBmodules;
GO
CREATE DATABASE IMDBmodules;
GO
ALTER DATABASE IMDBmodules ADD FILEGROUP IMDBmodules_mod_FG CONTAINS MEMORY_OPTIMIZED_DAT
A;
GO
-- adapt filename as needed
ALTER DATABASE IMDBmodules
      ADD FILE (name = 'IMDBmodules_mod', filename = 'c:\HKData\IMDBmodules_mod')
      TO FILEGROUP IMDBmodules_mod_FG;
GO
USE IMDBmodules
GO
CREATE TABLE dbo.t1
(c1 int not null primary key nonclustered,
 c2 int)
WITH (MEMORY_OPTIMIZED = ON);
GO
-- retrieve the path of the DLL for table t1
SELECT name, description FROM sys.dm_os_loaded_modules
WHERE name LIKE '%xtp_t_' + cast(db_id() AS varchar(10))
                  + '_' + cast(object_id('dbo.t1') AS varchar(10)) + '%.dll';
GO
```

The table creation results in the compilation of the table DLL, and also loading that DLL in memory. The DMV query immediately after the CREATE TABLE statement retrieves the path of the table DLL. Note that the name of the DLL includes a 't' for table, followed by the database id and the object id.

The table DLL for *t1* incorporates the index structures and row format of the table. SQL Server uses the DLL for traversing indexes and retrieving rows, as well as the contents of the rows.

Native compilation of modules

Modules that are marked with NATIVE_COMPILATION are natively compiled. This means the Transact-SQL statements in the module are all compiled down to native code, for efficient execution of performance-critical business logic.

Consider the following sample stored procedure *p1*, which inserts rows in the table *t1* from the previous example:

```
CREATE PROCEDURE dbo.p1
WITH NATIVE_COMPILATION, SCHEMABINDING
AS
BEGIN ATOMIC
WITH (TRANSACTION ISOLATION LEVEL=snapshot, LANGUAGE=N'us_english')

  DECLARE @i int = 1000000
  WHILE @i > 0
  BEGIN
    INSERT dbo.t1 VALUES (@i, @i+1)
  SET @i -= 1
  END

END
GO
EXEC dbo.p1
GO
-- reset
DELETE FROM dbo.t1
GO
```

The DLL for the procedure *p1* can interact directly with the DLL for the table *t1*, as well as the In-Memory OLTP storage engine, to insert the rows as fast as possible.

The In-Memory OLTP compiler leverages the query optimizer to create an efficient execution plan for each of the queries in the stored procedure. Note that, for natively compiled modules, the query execution plan is compiled into the DLL. SQL Server 2016 does not support automatic recompilation of natively compiled modules, However, you can use the *sp_recompile* stored procedure to force a natively compiled module to be recompiled at its next execution. You also have the option of ALTERing a natively compiled module definition, which will also cause a recompilation. While recompilation is in progress, the old version of the module continues to be available for execution. Once compilation completes, the new version of the module is installed.

In addition to forcing a module recompile, when you ALTER a natively compiled module. For natively compiled procedures the following options can be changed:

- Parameter list
- EXECUTE AS
- TRANSACTION ISOLATION LEVEL
- LANGUAGE

- DATEFIRST
- DATEFORMAT
- DELAYED_DURABILITY

Note that the only time natively compiled modules are recompiled automatically is on first execution after server restart, as well as after failover to an AlwaysOn secondary

Compilation and Query Processing

Figure 30 illustrates the compilation process for natively compiled stored procedures. The process is similar for other types of natively compiled modules.

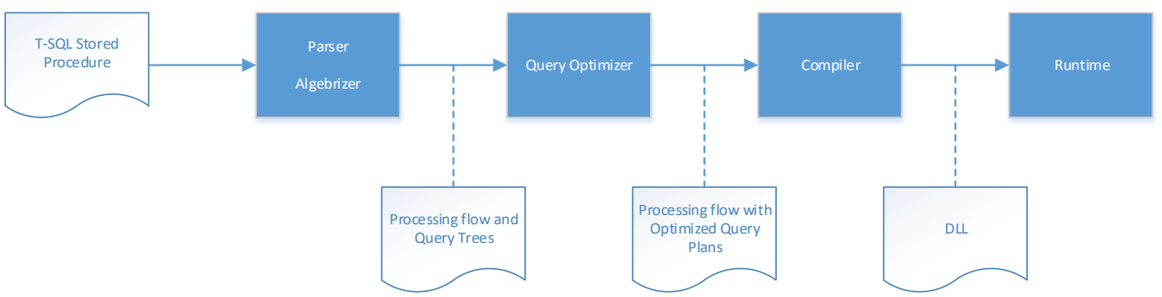

Figure 30: Native compilation of stored procedures

1. The user issues a CREATE PROCEDURE statement to SQL Server
2. The parser and algebrizer create the processing flow for the procedure, as well as query trees for the Transact-SQL queries in the stored procedure
3. The optimizer creates optimized query execution plans for all the queries in the stored procedure
4. The In-Memory OLTP compiler takes the processing flow with the embedded optimized query plans and generates a DLL that contains the machine code for executing the stored procedure
5. The generated DLL is loaded in memory and linked to the SQL Server process

Invocation of a natively compiled stored procedure translates to calling a function in the DLL, as shown in Figure 31.

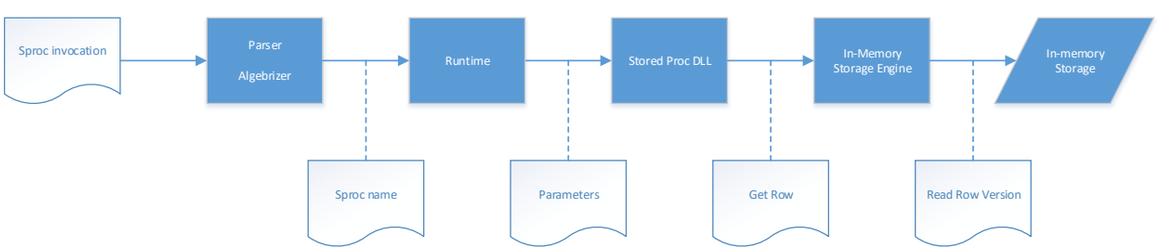

Figure 31: Execution of natively compiled stored procedures

1. The user issues an 'EXEC *myproc*' statement

2. The parser extracts the name and stored procedure parameters
3. The In-Memory OLTP runtime locates the DLL entry point for the stored procedure
4. The DLL executes the procedure logic and returns the results to the client

Parameter sniffing

Interpreted Transact-SQL stored procedures are compiled into intermediate physical execution plans at first execution (invocation) time, in contrast to natively compiled stored procedures, which are natively compiled at create time. When interpreted stored procedures are compiled at invocation, the values of the parameters supplied for this invocation are used by the optimizer when generating the execution plan. This use of parameters during compilation is called "parameter sniffing".

Parameter sniffing is not used for compiling natively compiled stored procedures. All parameters to the stored procedure are considered to have UNKNOWN values.

Optimization of natively compiled stored procedures has most of the same goals as optimization of interpreted procedures. That is, the optimizer needs to find query plans for each of the statements in the procedure so that those statements to be executed as efficiently as possible. As described in the earlier section on T-SQL Support, the surface area of allowed Transact-SQL constructs is limited in natively compiled procedures. The optimizer is aware of these limitations, so certain transformations it might perform for queries outside of a natively compiled procedure are not supported.

SQL Server Feature Support

Many SQL Server features are supported for In-Memory OLTP and databases containing memory-optimized tables, but not all. For example, AlwaysOn components, log shipping, and database backup and restore are fully supported. Transactional replication is supported, with memory-optimized tables as subscribers. However, database mirroring is 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.

Manageability Experience

In-Memory OLTP is completely integrated into the manageability experience of SQL Server. As mentioned above, SQL Server Management Studio is able to work with your memory-optimized tables, filegroups and natively compiled procedures. In addition, you can use Server Management Objects (SMO) and PowerShell to manage your memory-optimized objects.

Memory Requirements

When running In-Memory OLTP, SQL Server will need to be configured with sufficient memory to hold all your memory-optimized tables. Failure to allocate sufficient memory will cause transactions to fail at run-time during any operations that require additional memory. Normally this would happen during INSERT or UPDATE operations, but could also happen for DELETE operations on memory-optimized table with range

indexes. As we saw in the section above on Bw-trees, a DELETE can cause a page merge to happen, and because index pages are never updated, the merge operation allocates new pages. The In-Memory OLTP memory manager is fully integrated with the SQL Server memory manger and can react to memory pressure when possible by becoming more aggressive in cleaning up old row versions.

When predicting the amount of memory you'll need for your memory-optimized tables, a rule of thumb is that you should have twice the amount of memory that your data will take up. Beyond this, the total memory requirement depends on your workload; if there are a lot of data modifications due to OLTP operations, you'll need more memory for the row versions. If you're doing lots of reading of existing data, there might be less memory required.

For planning space requirements for indexes, hash indexes are straightforward. Each bucket requires 8 bytes, so you can just compute the number of buckets times 8 bytes. The size for your range indexes depends on both the size of the index key and the number of rows in the table. You can assume each index row is 8 bytes plus the size of the index key (assume K bytes), so the maximum number of rows that fit on a page would be 8176/(K+8). Divide that result into the expected number of rows to get an initial estimate. Remember that not all index pages are 8K, and not all pages are completely full. As pages need to be split and merged, new pages are created and you'll need to allow space for them, until the garbage collection process removes them.

Memory Size Limits

Although there is no hard limit on the amount of memory that can be used for memory-optimized tables in SQL Server 2016, Microsoft recommends an upper limit of no more than 2 TB of table data. Not only is this the limit Microsoft uses in its testing, but there are other system resource issues that can cause performance degradation if this limit is exceeded. Note that this 2 TB refers only to your active data in durable tables. It does not include old row versions, index storage, or data in non-durable tables. In addition, if you have temporal memory-optimized tables (a new feature in SQL Server 2016) you can choose to have the history information stored in a disk-based table so that less memory is used for in-memory data.

Managing Memory with the Resource Governor

A tool that allows you to be proactive in managing memory is the SQL Server Resource Governor. A database can be *bound* to a resource pool and you can assign a certain amount of memory to this pool. The memory-optimized tables in that database cannot use more memory than that, and this becomes the maximum memory value of which no more than 80% can be used for memory-optimized tables. This limit is needed to ensure the system remains stable under memory pressure. In fact, any memory consumed by memory-optimized tables and their indexes is managed by the Resource Governor, and no other class of memory is managed by the Resource Governor. If a database is not explicitly mapped to a pool, it will implicitly be mapped to the Default pool.

Using Resource Governor to limit the memory for memory-optimized tables makes your in-memory OLTP usage much more predictable and is definitely recommended. In addition, by setting an upper limit on memory which is less than the actual memory available, you'll have a bit of leeway to expand if you actually do run out of your configured limit. The first step is to create a memory pool for your In-Memory OLTP database specifying a MAX_MEMORY_PERCENT value. This specifies the percentage of the SQL

Server memory which may be allocated to memory-optimized tables in databases associated with this pool.

For example:

```
CREATE RESOURCE POOL IMPool WITH (MAX_MEMORY_PERCENT=50);
ALTER RESOURCE GOVERNOR RECONFIGURE;
```

Once you have created your resource pool(s), you need to bind the databases which you want to manage to the respective pools using the procedure *sp_xtp_bind_db_resource_pool*. Note that one pool may contain many databases, but a database is only associated with one pool at any point in time.

Here is an example:

```
EXEC sp_xtp_bind_db_resource_pool 'IMDB', 'IMPool';
```

Because memory is assigned to a resource pool as it is allocated, simply associating a database with a pool will not transfer the assignment of any memory already allocated. In order to do that, you need to take the database offline and bring it back online. As the data is read into the memory-optimized tables, the memory is associated with the new pool.

For example:

```
ALTER DATABASE [IMDB] SET OFFLINE;
ALTER DATABASE [IMDB] SET ONLINE;
```

Should you wish to remove the binding between a database and a pool, you can use the procedure *sp_xtp_unbind_db_resource_pool*. For example, you may wish to move the database to a different pool, or to delete the pool entirely, to replace it with some other pool(s).

```
EXEC sp_xtp_unbind_db_resource_pool 'IMPool';
```

The only hard limit is the amount of memory on your system and SQL Server limits your memory-optimized tables to a percentage of your SQL Server's Target Committed Memory and the memory allowed in the resource pool bound to the database. For example, if your target is 100 GB, SQL Server allows 90% of the memory available to be used for memory-optimized tables. So if your resource pool has a MAX_MEMORY_PERCENT value of 70, you will have 70% of 90% of 100 GB, or 63 GB for memory-optimized tables.

More details, including the percentages of target memory available for memory-optimized tables, can be found in the online documentation: https://msdn.microsoft.com/library/dn465873.aspx.

Metadata

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

There is one function that has been enhanced:

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

Catalog Views

The following system views have been enhanced:

- *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. (Range indexes have a *type_value* of 2 and a *type_desc* of NONCLUSTERED, just as for a nonclustered B-tree index.)
- *sys.index_columns* now has different semantics for the column *is_descending_key*, in that for HASH indexes, the value is meaningless and ignored.
- *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 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.

Dynamic Management Objects

The following SQL Server Dynamic Management Views provide metadata for In-Memory OLTP. (The *xtp* identifier stands for 'eXtreme transaction processing'.) 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. Some of these DMVs have already been mentioned in earlier relevant sections of this paper.

For more information about DMVs that support memory-optimized tables, see Memory-Optimized Table Dynamic Management Views.

- sys.dm_db_xtp_checkpoint_stats
- sys.dm_db_xtp_checkpoint_files
- sys.dm_db_xtp_gc_cycles_stats
- sys.dm_xtp_gc_stats
- sys.dm_xtp_gc_queue_stats
- sys.dm_xtp_threads
- sys.dm_xtp_system_memory_consumers
- sys.dm_db_xtp_memory_consumers
- sys.dm_db_xtp_table_memory_stats

- sys.dm_xtp_transaction_stats
- sys.dm_db_xtp_transactions
- sys.dm_xtp_transaction_recent_rows
- sys.dm_db_xtp_index_stats
- sys.dm_db_xtp_hash_index_stats
- sys.dm_db_xtp_nonclustered_index_stats
- sys.dm_db_xtp_object_stats

XEvents

The In-Memory OLTP engine provides about 160 XEvents to help you in monitoring and troubleshooting. You can run the following query to see the XEvents currently available:

```
SELECT p.name, o.name, o.description
FROM sys.dm_xe_objects o JOIN sys.dm_xe_packages p
  ON o.package_guid=p.guid
WHERE p.name = 'XtpEngine';
GO
```

Performance Counters

The In-Memory OLTP engine provides performance counters to help you in monitoring and troubleshooting your in-memory OLTP operations. You can see 8 different objects encompassing 89 counters, when looking at Windows Performance Monitor as shown in Figure 32.

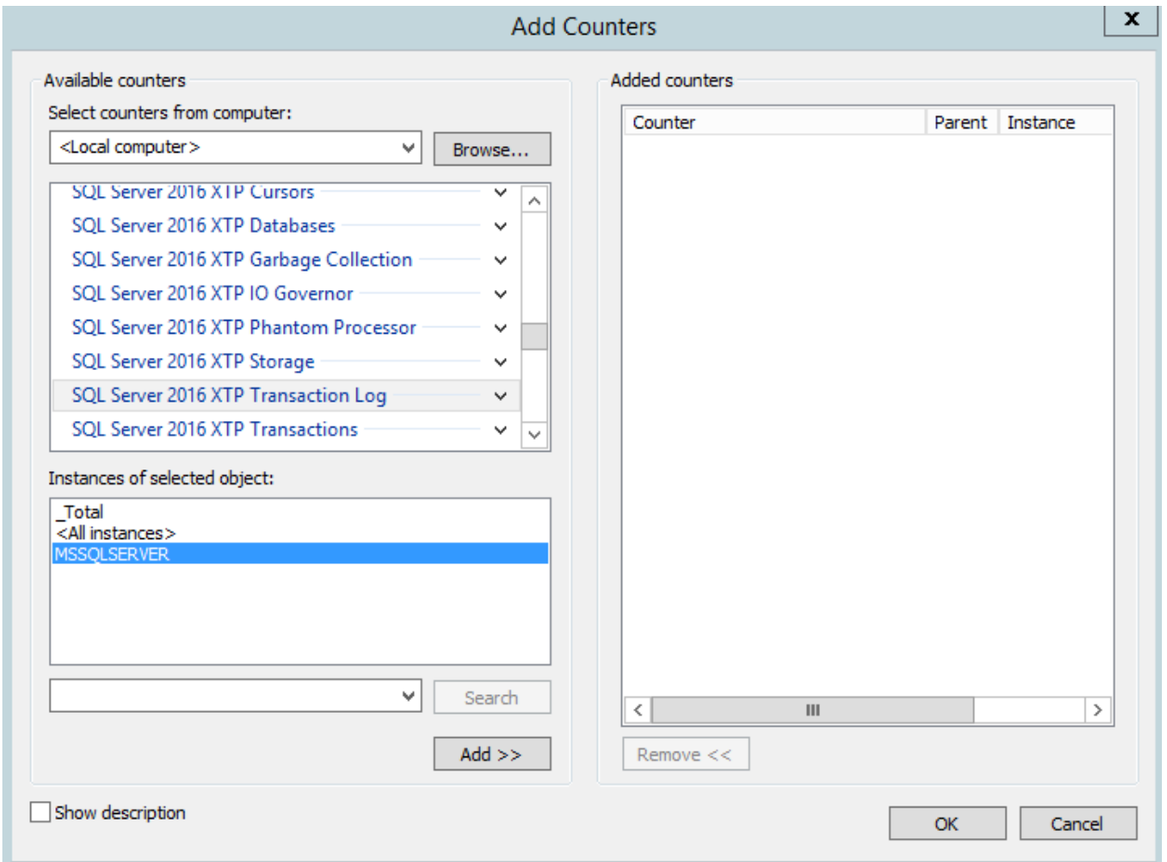

Figure 32: SQL Server in-memory OLTP objects in Performance Monitor

You can also see the all the objects and counters in the DMV *sys.dm_os_performance_counters*.

```
SELECT object_name as ObjectName, counter_name as CounterName
FROM sys.dm_os_performance_counters
WHERE object_name LIKE 'XTP%';
GO
```

The eight SQL Server 2016 XTP objects are listed and described in Table 8.

SQL Server 2016 XTP Cursors	The XTP Cursors performance object contains counters related to internal XTP engine cursors. Cursors are the low-level building blocks the XTP engine uses to process Transact-SQL queries. As such, you do not typically have direct control over them.
SQL Server 2016 XTP Databases	The XTP Databases object has the greatest number of counters (32) measuring things like log writes and checkpoint thread activity. Most of the counters are for not intended for customer use.
SQL Server 2016 XTP Garbage Collection	The XTP Garbage Collection performance object contains counters related to the XTP engine's garbage collector. Counters include the number of rows processed, the number of scans per second, and the number of rows expired per second.
SQL Server 2016 IO Governor	The XTP IO Governor performance object contains information about IO work performed by the in-memory OLTP engine.

SQL Server 2016 XTP Phantom Processor	The XTP Phantom Processor performance object contains counters related to the XTP engine's phantom processing subsystem. This component is responsible for detecting phantom rows in transactions running at the SERIALIZABLE isolation level.
SQL Server 2016 XTP Storage	The XTP Storage object contains counters related to the checkpoint files. Counters include the number for checkpoints closed, and the number of files merged.
XTP Transaction Log	The XTP Transaction Log performance object contains counters related to XTP transaction logging in SQL Server. Counters include number of log bytes and number of log records written by the In-Memory OLTP engine per second.
XTP Transactions	The XTP Transactions performance object contains counters related to XTP engine transactions in SQL Server. Counters include the number of commit dependencies taken and the number of commit dependencies that rolled back.

Table 8: Categories of performance counters for In-Memory OLTP processing

Memory Usage Report

SQL Server Management Studio provides a number of predefined reports, which are available by right-clicking on a database name in the Object Explorer and choosing 'Reports'. When expanding the 'Standard Reports' selection, you'll see list of options dealing with disk usage, blocking and transactions, memory and indexes. One of the reports, as shown in Figure 33, is called 'Memory Usage By Memory Optimized Objects".

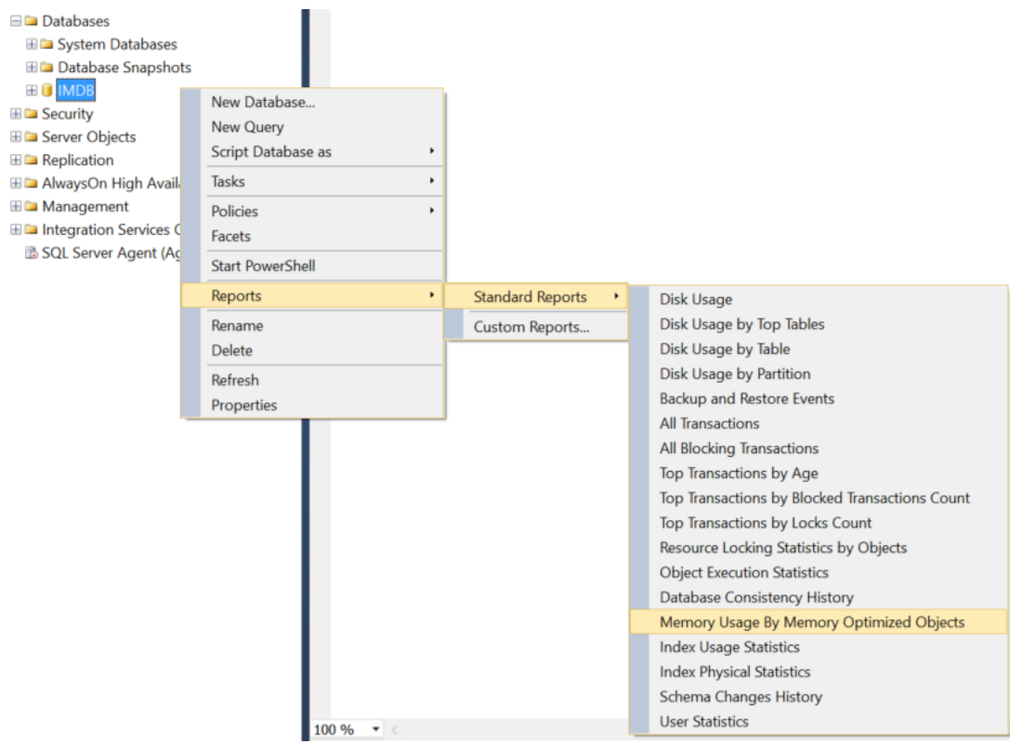

Figure 33: Choosing the Standard Report for memory used by memory-optimized tables

Figure 34 shows the output of this report after creating a memory-optimized table and populating it with a million rows.

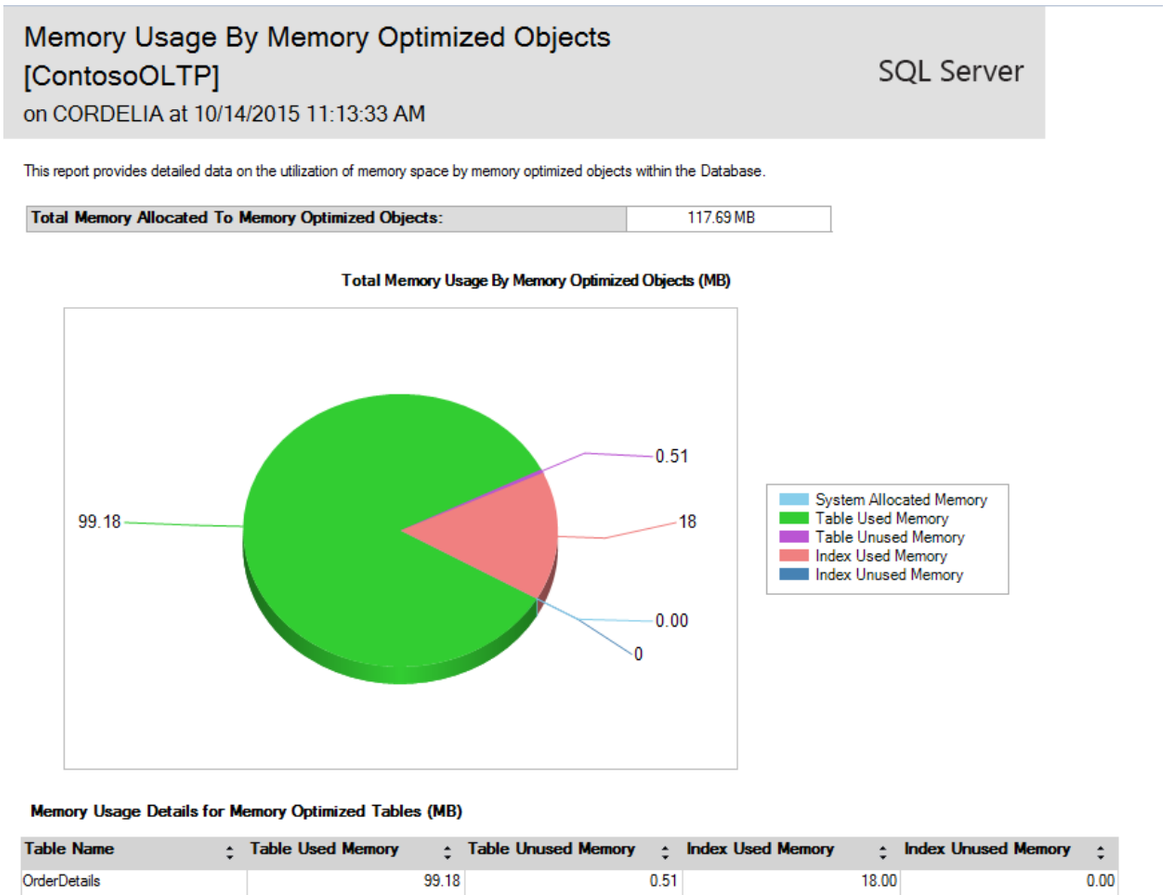

Figure 34 Report of Memory Usage By Memory Optimized Objects

This report shows you the space used by the table rows and the indexes, as well as the small amount of space used by the system. Remember that hash indexes will have memory allocated for the declared number of buckets as soon as they're created, so this report will show memory usage for those indexes before any rows are inserted. For range indexes, memory will not be allocated until rows are added, and the memory requirement will depend on the size of the index keys and the number of rows.

Migration to In-Memory OLTP

Although it might sound like In-Memory OLTP is a panacea for all your relational database performance problems, this of course is not true. There are some applications that can experience enormous improvement when using memory-optimized tables and natively compiled stored procedures, and others that will not see drastic gains, or perhaps no gains at all. The kinds of applications that will achieve the best improvements are the ones that are currently experiencing the bottlenecks that In-Memory OLTP addresses and removes.

The main bottlenecks that In-Memory OLTP addresses are the following:

Lock or latch contention

The lock and latch-free design of memory-optimized tables is probably the performance benefit that is most well-known. As discussed in detail in earlier chapters, the data structures used for the memory-optimized tables' row versions, and the fact that the rows are not stored in memory buffers, allow for high concurrency data access and modification without the need for locks or latches. Tables used by an application showing excessive lock or latch wait times can be considered for migration to In-Memory OLTP, and the application's performance will most likely show substantial improvement.

I/O and logging

Rows for your memory-optimized tables are always in memory, so no disk reads are ever required to make the data available. The streaming checkpoint operations are also highly optimized to use a minimal amount of resources to write the durable data to disk in the checkpoint files. In addition, index information is never written to disk, reducing the I/O requirements even further. If your application shows excessive page I/O latch waits, or any other waits associated with reading from or writing to disk, you will likely get a performance improvement with memory-optimized tables.

Transaction logging

Log I/O can be another bottleneck with disk-based tables, as (in most cases for OLTP operations) every table and index row modified is written to the transaction log on disk as a separate log record. Not only does In-Memory OLTP allow you to create SCHEMA_ONLY tables that do not require any logging, but even for tables defined as SCHEMA_AND_DATA, the logging overhead is significantly reduced. Each log record for changes to a memory-optimized table can contain information about many modified rows, and changes to indexes are never logged at all. If your application experiences high wait times due to log writes, you can see an improvement after migrating your most heavily modified tables to memory-optimized tables.

Tempdb contention

Your SQL Server's *tempdb* database is used for multiple purposes including temporary tables, table variables, internal worktables, trigger processing and row versions for snapshot isolation on disk-based tables. All databases share a single *tempdb*, so the demands on it can become extreme. Replacing temporary tables with non-durable memory-optimized tables and table variables with memory-optimized table variables can go a long way towards reducing *tempdb* contention among all the many uses of this database. Take a look at this blog post for further details: https://blogs.msdn.microsoft.com/sqlserverstorageengine/2016/03/21/improving-temp-table-and-table-variable-performance-using-memory-optimization

Hardware resource limitations

In addition to the limits on disk I/O that can cause performance problems with disk-based tables, other hardware resources can also be the cause of bottlenecks. CPU resources are frequently stressed in compute-intensive OLTP workloads. In addition, CPU resources are also cause slowdowns when small queries need to be executed repeatedly and the interpretation of the code needed by your queries needs to be repeated over and over again. Migrating your code to use

natively compiled procedures can greatly reduce the CPU resources required because the natively compiled code requires far few CPU instructions than the interpreted code needs to perform the same operations. If you have many small code blocks running repeatedly, especially if you are noticing a high number of recompiles, you may notice a substantial performance improvement after putting your code into natively compiled procedures.

The next section will describe some of the most common data access and manipulation scenarios that your application might include that would experience some of the bottlenecks listed above.

High Volume of INSERTs

Applications that are very INSERT oriented, such as sales order entry systems, frequently encounter bottlenecks with locks and latches on the last page of a table or index, if the rows are being inserted in a particular order. Even if row locks are being used, there are still latches acquired on the page, and for very high volumes this can be problematic. Another impact to performance occurs with the logging required for the inserted rows and for the index rows created for each inserted data row.

SQL Server In-Memory OLTP addresses these problems by eliminating the need for locks and latches. Logging overhead is reduced because operations on memory-optimized tables log their changes more efficiently. In addition, the changes to the indexes are not logged at all. If the application is such that the INSERT operations initially load data into a staging table, you can consider creating the staging table to be SCHEMA_ONLY, and then there will be no logging for the table rows also.

Finally, the code to process the INSERTs must be run repeatedly, for each row inserted. Using interop TSQL imposes a lot of overhead. If the code to process the INSERTs meets the criteria for creating a natively compiled procedure, executing the INSERTs through compiled code can make a major difference in performance.

High Volume of SELECTs

An application needing to quickly process a high volume of SELECT operations and to be able to scale to support even greater numbers faces some of the same bottlenecks with locking and latching as in the previous example. Of course, there is no logging requirement, but the other considerations will still apply.

You do need to be aware of the fact that operations on memory-optimized tables are always executed on a single thread; there is no support for parallel operations on memory-optimized tables. If you are processing large number of rows in each SELECT, this can be problematic, but fortunately, this is not the typical type of query for OLTP workloads. If you do have datasets that would benefit from parallelism, you can consider moving the relevant data to a separate table for processing. If that table is a disk-based table, then of course parallelism can be considered by the query optimizer. Alternatively, if moving the needed data to its own table reduces the number of rows that need to be scanned, that in itself can speed up the processing. Finally, if the code for processing these needed rows can be executed in a natively

compiled procedure, the speed improvement for compiled code can sometimes outweigh the cost of having the run the queries single threaded.

CPU-intensive operations

Similar to the first example where large volumes of data have to be inserted, there are additional considerations if the data needs to be manipulated before it is available for reading by the application. The manipulation can involve updating or deleting some of the data if it is deemed inappropriate, or can involve computations to put the data into the proper form for use.

The biggest bottleneck in this scenario will be the locking and latching as the data is read for processing and then the processing is invoked. Additional bottlenecks such as CPU resources can occur depending on the complexity of the actual code being executed.

As discussed, In-Memory OLTP can provide a solution for all of these bottlenecks.

Extremely fast business transactions

Applications that need to run a large volume of simple transactions very quickly from hundreds if not thousands of concurrent users, experience bottlenecks both from the latching and locking required and from the CPU associated with the query processing stack. If the queries themselves are relatively short and simple, the cost of repeated recompilation and query interpretation can become a major component of the overall processing time.

In-Memory OLTP solves these problems by providing a lock and latch free environment. Also, the ability to run the code in a truly compiled form, with no compiling or interpretation, can give an enormous performance boost for these kinds of applications.

Session state management

Although this is not specifically a type of application, but rather a function required by many different types of applications, it still bears mention here.

Session state management involves maintaining state information across various boundaries where normally there is no communication. The most prominent example is web-based interactions using HTTP. When users connect to a website, information about their choices and actions needs to be maintained across multiple HTTP requests. In general, this is something that can be maintained by the database system, but typically at a high cost. The state information is usually very dynamic and highly concurrent which each user's information changing very frequently. It also can involve lookup queries for each user to gather other information the system might be keeping for that user, such as past activity. Although the data maintained might be minimal in size, the number of requests to access that data can be large, leading to extreme locking and latching requirements resulting in very noticeable bottlenecks and serious slowdowns in responses to user requests.

In-Memory OLTP is perfect solution for this application requirement, as a small memory-optimized table can handle an enormous number of concurrent lookups and modifications. In addition, a session state table almost always is transient and does not need to be preserved across server restarts, so a SCHEMA_ONLY table can be used an improve the performance even further.

Unsuitable Application Scenarios

Although there are many types of applications that can gain considerable performance improvement when using In-Memory OLTP, either just by creating memory-optimized tables or by including natively compiled stored procedures, not every application is suited to In-Memory OLTP. In most cases, at least part of any application could be better served using traditional disk-based tables, or at least, you might not see any improvement with In-Memory OLTP. If your application meets any of the following criteria, you may need to rethink whether In-Memory OLTP is right for you.

Inability to make changes

If your application requires table features that are not supported by memory-optimized tables, you will not be able to create in-memory tables, or you may need to redefine the table. In addition, if your application code for accessing and manipulating your table data uses constructs not supported for natively compiled procedures, you may have to limit your TSQL to using only interop code.

Memory limitations

Memory-optimized tables must be completed in memory. If the size of your tables exceeds what SQL Server In-Memory OLTP, or your particular machine, supports, you will not be able to have all the required data in memory. Of course, you can have some memory-optimized tables and some disk-based tables, but you'll need to carefully analyze the workload to find which tables will provide the most benefit by being created as memory-optimized tables.

Non-OLTP workload

In-Memory OLTP, as the name implies, is designed to be the most benefit to Online Transaction Processing operations. There of course may be of benefit to other types of processing, such as reporting and data warehousing, but those are not the design goals of the feature. If you are working with processing that is not OLTP in nature, you should make sure you carefully test all operations, and you may find that In-Memory OLTP does not provide you with any measurable improvements.

Dependencies on locking behavior

Although not best practice in most cases, you might have application processing that depends on the locking behavior supplied with pessimistic concurrency on disk-based tables. For example, if you're using the READPAST hint to manage work queues, you need to have locks in order to find the next row in the queue to process. If your application was written to expect the behavior experienced in SNAPSHOT ISOLATION on disk-based tables when a write-write conflict occurs, i.e. that the conflict is not reported until the first process commits, then you will not want to use SNAPSHOT isolation with memory-optimized tables. As mentioned, it is usually not best practice to write an application that depends on specific locking behavior, as that can change even without switching to In-Memory OLTP. However, that doesn't mean that no one will have code that does this. If yours is one of those applications, you'll need to either delay converting to In-Memory OLTP, or rewrite the relevant sections of your code.

The Migration Process

SQL Server In-Memory OLTP can make migration a very straight forward and manageable process, because migration doesn't have to be an all or nothing decision. You could choose to just convert one or two critical tables, for which you'd noticed a very large number of locks or latches, and/or long durations on waits for locks or latches. Tables should be converted before stored procedures, since natively compiled procedures will only be able to access memory-optimized tables.

Ideally, before starting any migration of tables or stored procedures to use In-Memory OLTP, you perform a thorough analysis of your current workload, and establish a baseline. Monitoring, analysis and baselining is well beyond the scope of this book, but you can take a look at this page in the SQL Server 2016 documentation to get several pointers on performance this kind of analysis: https://msdn.microsoft.com/library/ms189081(v=sql.130).aspx .

You might consider something like the following list of steps as you work through a migration to In-Memory OLTP:

1. Identify the tables with the biggest bottlenecks
2. Address the constructs in the table DDL that are not supported for memory-optimized tables. Note that there is Memory Optimization Advisor tool, available through SQL Server Management Studio by right-clicking any table in any database, that can tell you what constructs are not supported for memory-optimized tables.
3. Recreate the tables as in-memory, to be accessed using interop code.
4. Identify procedures or sections of code with biggest performance bottlenecks, that access the converted tables.
5. Address the T-SQL limitations in the code. If the code is in a stored procedure, you can use the Native Compilation Advisor tool, available through SQL Server Management Studio by right-clicking any stored procedure. Recreate the code in a natively compiled procedure.
6. Compare performance against the baseline.

You can think of these migration steps as a cyclic process. Start with a few tables and convert them. Then convert the most critical procedures that access those tables. Then convert a few more tables, and then a couple of additional procedures. You can repeat this cycle as needed, until the performance gains are minimal.

Best Practice Recommendations

Although I have mentioned some best practice recommendations earlier in this paper as I described various features and the choices you can make, I want to include a specific list here of the most important ones. Keep these principals in mind as you design your memory-optimized tables and indexes:

Index Tuning

The costing formula that the optimizer uses for operations on memory-optimized tables is similar to the formula for operations on disk-based tables, with only a few exceptions. In this section, we'll look at some

of the issues involving index usage selection for memory-optimized tables by the SQL Server query optimizer.

Statistics and recompilations

SQL Server In-Memory OLTP does not keep any modification counters for individual columns. There is a modification counter kept for each index so that SQL Server can determine when to invoke the automatic update of statistics. The threshold for automatic updating of statistics is the same as for disk-based tables, and the sampling formulas for computing the new statistics are the same. Updated statistics will trigger a recompile for interop queries, but natively compiled procedure plans will never be recompiled on the fly. The only way to get a new plan is to drop and recreate the procedure.

Note also that DBCC SHOW_STATISTICS will not return any information for the statistics on a memory-optimized table.

General indexing guidelines

It is recommended that you first build your memory-optimized tables with range indexes, and only then consider hash indexes if you already get reasonable performance with range indexes, and you want to further optimize point lookups and inserts.

- Keep in mind these guidelines for using hash indexes on memory-optimized tables:
    - Do not over or underestimate the bucket count for hash indexes if at all possible. The bucket count should be at least equal to the number of distinct values for the index key columns.
    - Each bucket uses memory, so provisioning too many buckets will waste memory that could be used for more in-memory data. However, too few buckets means that there will be more hash collisions, making the seeking into a specific bucket much less efficient. There is no way to guarantee that there are no collisions when hashing, but having at least as many buckets as the expected number of distinct values will minimize them. It is better to have too many buckets than too few.
- For very low cardinality columns, create range indexes instead of hash indexes.

    If there are only a small number of unique values, there will be many duplicates of each one, and the bucket chains can be very long. And because it is impossible to avoid hash collisions, when more than one value shares the same bucket, there will be a lot of unwanted values to scan through and ignore. An example here would be an index on a *state* column in a *customers* table. If both 'Florida' and 'Wisconsin' are getting hashed to the same bucket, and there are tens of thousands of rows for each value, a query that is searching for state = 'Florida' will have to access and then ignore all the tens of thousands of Wisconsin rows while seeking for the Florida rows. In general, if you are expecting more 100 or more duplicates for most values of an index key, create a range index, and add columns to make the index key more unique. For example, you can add the primary key columns. Having large numbers of duplicates leads to inefficiency in garbage collection, which can lead to high CPU utilization.

- Create columnstore indexes only for tables that will be used for hybrid transaction processing and analytical workloads. They are useful if you have a transactional workload, and want to do analytics on real-time data.

- Create indexes on your foreign key columns.

- As we saw in the discussion on post processing, an appropriate index on foreign key columns means that SQL Server does not have to scan the table when validating any concurrent updates, and will be less likely to generate validation errors.

Also, because of differences in the way that memory-optimized tables are organized and managed, the optimizer does need to be aware of different choices it may need to make and certain execution plan options that are not available when working with memory-optimized tables. The most important differences are listed here:

- There are no ordered scans with hash indexes.

- If your query is looking for a range of values or requires that the results be returned in sorted order, a hash index will not be useful at all, and thus will be not even be considered by the optimizer.

- A hash index can only be used if the filter is based on an equality comparison.

- This is a similar situation to the previous bullet. If the query does not specify an exact value for one of the columns in the hash index key, the hash value cannot be determined. So if we have a hash index on *city*, and the query is looking for *city LIKE 'San%*', a hash lookup is not possible.

- A hash index cannot be used if not all columns are included in filter conditions.

- The examples shown earlier for hash indexes illustrated an index on just a single column. However, just like indexes on disk-based tables, hash indexes on memory-optimized tables can be composite. However, the hash function used to determine which bucket a row belongs in is based on all columns in the index. So if we had an index on *(city, state)*, a row for a customer from Springfield, Illinois would hash to a completely different value than a row for a customer from Springfield, Missouri, and also would hash to a completely different value than a row for a customer from Chicago, Illinois. If a query only supplies a value for city, a hash value cannot be generated and the index cannot be used, unless the entire index is used for a scan.

- Range indexes cannot be scanned in reverse order.

- There is no concept of 'previous pointers' in a range index on a memory-optimized tables. If your query requests the data to be sorted in DESC order, the optimizer could only use an index that was built as a *descending* index. In fact, it is possible to have two indexes on the same column, one defined as *ascending* and one defined as *descending*. It is also possible to have both a range and a hash index on the same column. In general, the optimizer will choose to use a hash index over a range index if the cost estimations are the same.

- Halloween protection is not incorporated into the query plans.

- Halloween protection provides guarantees against accessing the same row multiple times during query processing. Operations on disk-based tables use spooling operators to make sure rows are not accessing repeatedly, but this is not necessary for plans on memory-optimized tables. Halloween protection is provided in the storage engine for memory-optimized tables by including a statement ID as part of the row version overhead bytes. The statement ID that introduced a row version is stored with the row, so if the same statement encounters that row again, it knows it has already been processed.
-

If no index can be used efficiently, the plan chosen will be a table scan. For SQL Server 2014, , there really was no concept of a table scan for memory-optimized tables because all rows were only connected through their indexes. However, for SQL Server 2016, now that all tables have their own dedicated varheap memory consumer, SQL Server can just access the rows in the varheap to do a table scan. In addition, as of SQL Server 2016, parallel processing operations are possible, and a table scan can be parallelized.

General Suggestions

- There are no automatic recompiles of any natively compiled modules.
- Memory optimized table variables behave the same as regular table variables, but are stored in your database's memory space, not in *tempdb*. You can consider using memory-optimized table variables anywhere, as they are not transactional and will can help relieve *tempdb* contention.
- Use SNAPSHOT isolation level for your memory-optimized tables if at all possible and enable the database option `MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT`. Remember that there are two transaction contexts when using memory-optimized tables, and when memory-optimized tables use SNAPSHOT, your disk-based tables must use an isolation level other than SNAPSHOT The SET option is only applicable to disk-based tables.

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. Natively compiled stored procedures are the fastest way to access your memory-optimized tables and performance business logic computations.

For more information:

https://msdn.microsoft.com/library/dn133186.aspx : SQL Server 2016 in-memory OLTP documentation

http://www.microsoft.com/sqlserver/: SQL Server Web site

http://technet.microsoft.com/sqlserver/: SQL Server TechCenter

http://msdn.microsoft.com/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 whitepapers we release.

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 book, please submit them through the feedback link.

Send feedback.