The Hekaton Memory-Optimized OLTP Engine

Per-Ake Larson
palarson@microsoft.com

Mike Zwilling
mikezw@microsoft.com

Kevin Farlee
kfarlee@microsoft.com

Abstract

Hekaton is a new OLTP engine optimized for memory resident data and fully integrated into SQL Server; a database can contain both regular disk-based tables and in-memory tables. In-memory (a.k.a.Hekaton) tables are fully durable and accessed using standard T-SQL. A query can reference both Hekaton tables and regular tables and a transaction can update data in both types of tables. T-SQL stored procedures that reference only Hekaton tables are compiled into machine code for further performance improvements. To allow for high concurrency the engine uses latch-free data structures and optimistic, multiversion concurrency control. This paper gives an overview of the design of the Hekaton engine and reports some initial results.

1 Introduction

SQL Server, like other major database management systems, was designed assuming that main memory is expensive and data resides on disk. This is no longer the case; today a server with 32 cores and 1TB of memory costs as little as $50K. The majority of OLTP databases fit entirely in 1TB and even the largest OLTP databases can keep the active working set in memory. Recognizing this trend SQL Server several years ago began building a database engine optimized for large main memories and many-core CPUs. The new engine, code named Hekaton, is targeted for OLTP workloads.

Hekaton has a number of features that sets it apart from other main-memory database engines. Most importantly, the Hekaton engine is integrated into SQL Server; it is not a separate product. A database can contain both Hekaton in-memory tables and regular disk-based tables. This approach offers customers major benefits compared with a separate system. First, customers avoid the hassle and expense of another DBMS. Second, only the most performance-critical tables need to be in main memory; other tables can remain regular SQL Server tables. Third, conversion can be done gradually, one table and one stored procedure at a time.

Memory optimized tables are managed by Hekaton and stored entirely in main memory. A Hekaton table can have several indexes which can be hash indexes or range indexes. Hekaton tables are durable and transactional, though non-durable tables are also supported. Hekaton tables are queried and updated using T-SQL in the same way as regular SQL Server tables. A query can reference both Hekaton tables and regular tables and a transaction can update both types of tables. Furthermore, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code for further performance gains. This is by far the fastest way to query and modify data in Hekaton tables. Hekaton is designed for high levels of concurrency but does not rely on partitioning to achieve this. Any thread can access any row in a table without acquiring latches or locks.

The engine uses latch-free (lock-free) data structures to avoid physical interference among threads and a new optimistic, multi-version concurrency control technique to reduce interference among transactions.

2 Design Overview

This section provides a brief overview of the design of the Hekaton engine and the guiding principles behind the design decisions. The interested reader can find more details in [2].

2.1 No partitioning

HyPer [5], Dora [8], H-store [18], and VoltDB [22] are recent systems designed for OLTP workloads and memory resident data. They partition the database by core and give one core exclusive access to a partition. Partitioning works great but only if the workload is also partitionable. If the workload partitions poorly so that transactions on average touch several partitions, performance deteriorates considerably.

Hekaton does not partition the database and any thread can access any part of the database. We prototyped and carefully evaluated a partitioned engine but came to the conclusion that a partitioned approach is not sufficiently robust to handle the wide variety of workloads customers expect SQL Server to handle.

2.2 Tables and indexes optimized for main memory

Database systems traditionally use disk-oriented storage structures where records are stored on disk pages that are brought into memory as needed. This requires a complex buffer pool where a page must be protected by latching before it can be accessed. The resulting overhead is high: a simple key lookup in a B-tree index may require thousands of instructions even when all pages are in memory.

Hekaton indexes and other data structures are designed and optimized for memory-resident data and the on-disk representation is completely different than their in-memory representation and in fact indexes are not materialized on external storage. Records are referenced directly by physical pointers, not indirectly by a logical pointer such as a page/row ID. Record pointers are stable; a record is never moved after it has been created. A table can have multiple indexes, any combination of hash indexes and range indexes. A hash index is simply an array where each entry is the head of a linked list through records. Range indexes are implemented as Bw-trees [7] which is novel version of B-trees optimized for main-memory and high concurrency.

2.3 Latch-free data structures

Machines with large numbers of CPU cores are increasingly common and core counts are expected to continue increasing. Achieving good scaling on many-core machines is critical for high throughput. Scalability suffers when the systems has shared memory locations that are updated at high rate such as latches and spinlocks and highly contended resources such as the lock manager, the tail of the transaction log, or the last page of a B-tree index.

All Hekaton's internal data structures, for example, memory allocators, hash and range indexes, and the transaction map, are entirely latch-free (lock-free) [3]. There are no latches or spinlocks on any performance-critical paths in the system.

2.4 Multi-versioning

Database systems traditionally maintain only a single version of a record and all updates are applied in place with transaction locks used to synchronize access which can limit scalability. A writer prevents all accesses to the record until the transaction commits while readers prevent the record from being updated. Because readers

block writers, even a few long read-only transactions (to prepare a report, for example) can drastically reduce update throughput.

To avoid readers competing with writers, Hekaton. like many other database systems, uses multi-versioning where an update creates a completely new version of a record. Once created, the user payload of a version is never modified. The lifetime of a version is defined by two timestamps, a begin timestamp and an end timestamp and different versions of the same record have non-overlapping lifetimes. A transaction specifies a logical read time for all reads, typically the transaction's begin time, and only versions whose lifetime overlaps the read time are visible to the transaction. Consequently, the transaction will read at most one version of a record.

Multi-versioning improves scalability because readers no longer block writers. (Writers may still conflict with writers though.) Read-only transactions have little effect on update activity; they simply read older versions of records as needed. In particular, long read-only transactions no longer reduce update transaction throughput. Furthermore, once a reader has a memory pointer to a record version that is visible to them, it can trust that the version will never change or move.

However, multi-version is not without cost. Updating a record in place is faster than creating a new version and obsolete versions no longer needed must be cleaned out. Hekaton's algorithms for cleaning out obsolete versions and reclaiming memory is described in more detail in [2].

2.5 Concurrency control

Hekaton uses a new optimistic multi-version concurrency control algorithm to provide transaction isolation; there are no locks and no lock table. The algorithm is described in detail in [6] but we provide a brief summary here.

A transaction optimistically assumes that it is running isolated (versioning makes it appear so), but before a transaction can commit it must validate that it indeed has not been interfered with by another transaction. This validation begins once a transaction has completed its normal processing and wants to commit. Any number of transactions can perform validation in parallel. The extent of validation required depends on the transaction's isolation level. Read-only transactions (regardless of isolation level) and transactions running under read-committed or snapshot isolation require no validation at all. Write-write conflicts are detected immediately when a transaction attempts to update a version and result in the transaction rolling back.

Transactions running under repeatable read or serializable isolation require validation before commit. During validation a transaction T checks whether the following two properties hold.

1. **Read stability.** If T read some version V1 during its processing, we must ensure that V1 is still the version visible to T as of the end of the transaction. This is implemented by validating that V1 has not been updated before T commits. Any update will have modified V1's end timestamp so all that is required is to check V1's timestamps. To enable this, T retains a pointer to every version that it has read.

2. **Phantom avoidance.** For serializable transactions we must also ensure that the transaction's scans would not return additional new versions. This is implemented by rescanning to check for new versions before commit. To enable this, a serializable transaction keeps tracks of all its index scans and retains enough information to be able to repeat each scan.

To avoid blocking during normal processing, Hekaton allows a limited form of speculative reads: a transaction T1 can read a version created by another transaction T2 that is still in validation. T1 cannot commit or return results to the user until T2 has committed. To enforce this, T1 takes a commit dependency on T2 which is released by T2 as soon as it has committed. If T2 aborts, it instructs T1 to abort.

The combination of multi-versioning, optimistic concurrency control with speculative reads, and latch-free data structures results in a system where a thread never blocks or stall during normal processing of a transaction.

2.6 Durability

Transaction durability is ensured by logging and checkpointing records to external storage; index operations are not logged. During recovery Hekaton tables and their indexes are rebuilt entirely from the latest checkpoint and logs.

A transaction that successfully passes validation is ready to commit. At this point it writes to the log all new versions that it has created and keys of all versions it has deleted. This is done in a single write and if the write succeeds, the transaction is irrevocably committed. Hekaton puts much less pressure on the log than regular SQL Server because nothing is written to the log until commit. Aborted transactions are not logged so aborting a transaction is cheap. Hekaton can spread the log over multiple log devices because commit ordering in Hekaton is determined by transactions' end timestamps, not by log ordering.

Checkpoints are computed by processing the log, not by scanning tables. The checkpoint algorithms use only sequential reads and writes to avoid the high overhead and latency of random IO. More details about logging and checkpointing can be found in [2].

2.7 Compilation to native code

SQL Server uses interpreter based execution mechanisms in the same ways as most traditional DBMSs. This provides great flexibility but at a high cost: even a simple transaction performing a few lookups may require several hundred thousand instructions. Hekaton maximizes run time performance by converting stored procedures written in T-SQL into customized, highly efficient machine code. The generated code contains exactly what is needed to execute the request, nothing more. As many decisions as possible are made at compile time to reduce runtime overhead. For example, all data types are known at compile time allowing the generation of efficient code. Hekaton's compilation process is described in [2].

3 High-Level Architecture

As illustrated in Figure 1, Hekaton consists of three major components.

- The Hekaton storage engine manages user data and indexes. It provides transactional operations on tables of records, hash and range indexes on the tables, and base mechanisms for storage, checkpointing, recovery and high-availability.

- The Hekaton compiler takes an abstract tree representation of a T-SQL stored procedure, including the queries within it, plus table and index metadata and compiles the procedure into native code designed to execute against tables and indexes man-aged by the Hekaton storage engine.

Figure 1: Hekaton components and integration with SQL Server

- The Hekaton runtime system provides integration with SQL Server resources and serves as a common library of additional functionality needed by compiled stored procedures.

Hekaton leverages a number of services already available in SQL Server. The main integration points are illustrated in Figure 1.
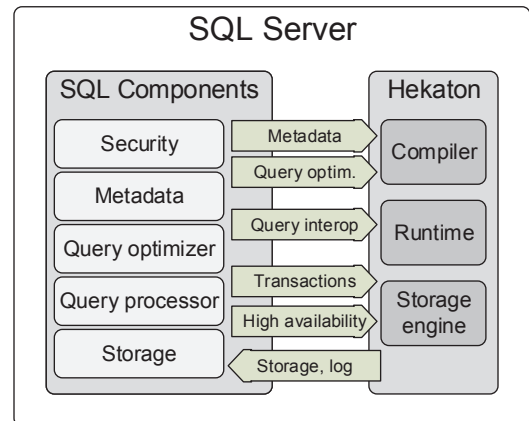
- *Metadata:* Metadata about Hekaton tables, indexes, etc. is stored in the regular SQL Server catalog. Users view and manage them using exactly the same tools as regular tables and indexes.

- *Query optimization:* Queries embedded in compiled stored procedures are optimized using the regular SQL Server optimizer. The Hekaton compiler compiles the query plan into native code.

- *Query interop:* Hekaton provides query operators for accessing data in Hekaton tables that can be used in interpreted SQL Server query plans. There is also an operator for inserting, deleting, and updating data in Hekaton tables.

- *Transactions:* A regular SQL Server transaction can access and update data both in regular tables and Hekaton tables. Commits and aborts are fully coordinated across the two engines.

- *High availability:* Hekaton is integrated with AlwaysOn, SQL Server's high availability feature. Hekaton tables in a database fail over in the same way as other tables.

- *Storage, log:* Hekaton logs its updates to the regular SQL Server transaction log. It uses SQL Server file streams for storing checkpoints. Hekaton tables are automatically recovered in parallel while the rest of the database is recovered.

4 Performance Illustration

We now report results from an experiment that illustrates the performance and scalability improvements offered by Hekaton compared with regular SQL Server. The experiment emulates an order entry system for, say, a large online retailer. The load on the system is highly variable and during peak periods throughput is limited by lock and latch contention. The system is simply not able to take advantage of additional processor cores so throughput levels off or even decreases. When SQL Server customers experience scalability limitations, the root cause is frequently lock or latch contention. Hekaton is designed to eliminate lock and latch contention, allowing it to continue to scale with the number of processor cores.

System throughput

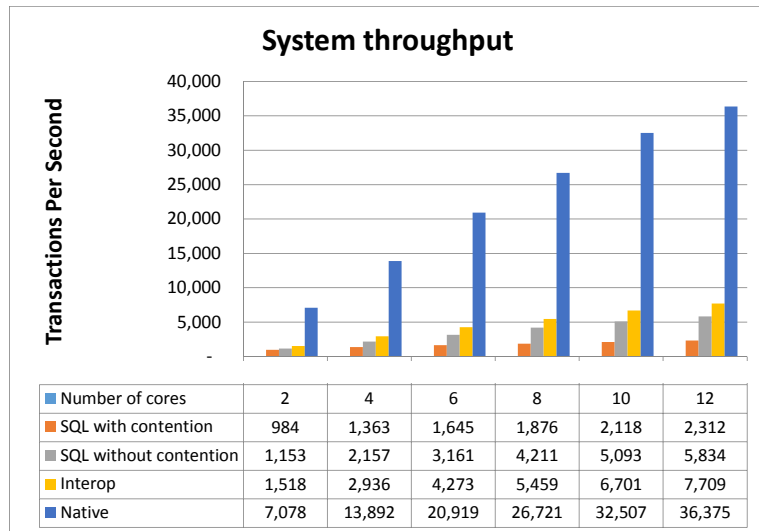|	2	4	6	8	10	12
■ Number of cores	2	4	6	8	10	12
■ SQL with contention	984	1,363	1,645	1,876	2,118	2,312
■ SQL without contention	1,153	2,157	3,161	4,211	5,093	5,834
■ Interop	1,518	2,936	4,273	5,459	6,701	7,709
■ Native	7,078	13,892	20,919	26,721	32,507	36,375

Figure 2: Improved throughput and scalability offered by Hekaton

The main activity is on a table SalesOrderDetails that stores data about each item ordered. The table has a unique index on the primary key which is a clustered B-tree index if the table is a regular SQL Server table and a hash index if it is Hekaton table. The workload in the experiment consists of 60 input streams, each a mix of 50% update transactions and 50% read-only transactions. Each update transaction acquires a unique sequence number, which is used as the order number, and then inserts 100 rows in the SalesOrderDetails table. A read-only transaction retrieves the order details for the latest order.

The experiment was run on a machine with 2 sockets, 12 cores (Xeon X5650, 2.67GHz), 144GB of memory, and Gigabit Ethernet network cards. External storage consisted of four 64GB Intel SSDs for data and three

80GB Fusion-IO SSDs for logs. There was sufficient memory to hold the entire database using either regular or Hekaton tables.

Figure 2 shows the throughput as the number of cores used varies. The regular SQL Server engine shows limited scalability as we increase the number of cores used. Going from 2 cores to 12 cores throughput increases from 984 to 2,312 transactions per second, only 2.3X. Latch contention limits the CPU utilization to only 40% for more than 6 cores.

Converting the table to a Hekaton table and accessing it through interop already improves throughput to 7,709 transactions per second for 12 cores, a 3.3X increase over plain SQL Server. Accessing the table through compiled stored procedures improves throughput further to 36,375 transactions per second at 12 cores which is 15.7X improvement over regular SQL Server.

The Hekaton engine shows excellent scaling. Going from 2 to 12 cores, throughput improves by 5.1X for the interop case (1,518 to 7,709 transactions per second). If the stored procedures are compiled, throughput also improves by 5.1X (7,078 to 36,375 transactions per second).

We also investigated the performance of the regular SQL Server engine when there was no contention. We partitioned the database and rewrote the stored procedure so that different transactions did not interfere with each other. The results are shown in the row labeled "SQL with no contention". Removing contention increased maximum throughput to 5,834 transaction/sec which is still lower than the throughput achieved through interop. Without contention scaling improved to 5.1X going from 2 cores to 12 cores, compared with 2.3X with contention.

5 Initial Customer Experiences

Bwin is the largest regulated online gaming company in the world, and their success depends on positive customer experiences. They use SQL Server extensively and were keen to try out an early preview of Hekaton. Prior to using Hekaton, their online gaming systems were handling about 15,000 requests per second, a huge number for most companies. However, Bwin needed to be agile and stay at ahead of the competition so they wanted access to the latest technology speed. Using Hekaton Bwin were hoping they could at least double the number of transactions. They were pretty amazed to see that the fastest tests so far have scaled to 250,000 requests per second.

Edgenet is a leading provider of product data management, product listings, and retail selling solutions, aggregating and curating product information from various suppliers, and correlating it to availability and inventory information from the stores of many retailers to present end users with a comprehensive shopping experience. The project prototyped was an inventory tracking system, which receives batch updates from each retailer with updates for all products in each of their stores. The baseline SQL measurement hit maximum performance with 2 threads, at 7,450 rows/second. Hekaton scaled smoothly up to 50 threads, yielding 126,665 rows per second.

SBI Liquidity Market(SBILM), located in Japan, provides financial trading infrastructure for tenant companies. SBILM's current volume of trading is $1.75 Trillion, around twice that of the Japanese government's budget. They are investing in their next-generation foreign exchange trading platform. The system explored aggregate trading data to calculate prices for currency pairs. The timeliness of this data is critical to profitability. When trading greatly increases, their current system yields 4 second aggregation time. Their goal is less than one second, which equates to 4000 TPS. In lab tests, they achieved a maximum of 2812 TPS with traditional SQL tables, and 5313 with Hekaton, significantly exceeding their goals.

6 Conclusion

Hekaton is a new main-memory engine under development a Microsoft. Hekaton is fully integrated into SQL Server, which allows customers to gradually convert their most performance-critical tables and applications to

take advantage of the performance improvements offered by Hekaton. Hekaton achieves high performance and scalability by using very efficient latch-free data structures, multi-versioning, optimistic concurrency control, and by compiling T-SQL stored procedure into efficient machine code. Transaction durability is ensured by logging and checkpointing to external storage. High availability and transparent failover is provided by integration with SQL Server's AlwaysOn feature. The Hekaton engine is capable of delivering more than an order of magnitude improvement in efficiency and scalability with minimal and incremental changes to user applications or tools.

7 Acknowledgements

References

[1] *VoltDB*. http://voltdb.com.

[2] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Sigmod*, page (to appear), 2013.

[3] K. Fraser and T. L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[4] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[5] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[6] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[7] J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, pages 3012–313, 2013.

[8] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.