Hekaton: SQL Server's Memory-Optimized OLTP Engine

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson,
Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling
Microsoft

{cdiaconu, craigfr, eriki, palarson, pravinm, ryanston, nitinver, mikezw}@microsoft.com

ABSTRACT

Hekaton is a new database engine optimized for memory resident data and OLTP workloads. Hekaton is fully integrated into SQL Server; it is not a separate system. To take advantage of Hekaton, a user simply declares a table memory optimized. Hekaton tables are fully transactional and durable and accessed 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 data in both types of tables. T-SQL stored procedures that reference only Hekaton tables can be compiled into machine code for further performance improvements. The engine is designed for high concurrency. To achieve this it uses only latch-free data structures and a new optimistic, multiversion concurrency control technique. This paper gives an overview of the design of the Hekaton engine and reports some experimental results.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *relational databases, Microsoft SQL Server*

General Terms

Algorithms, Performance, Design

Keywords

Main-memory databases, OLTP, SQL Server, lock-free data structures, multiversion concurrency control, optimistic concurrency control, compilation to native code.

1. INTRODUCTION

SQL Server and other major database management systems were designed assuming that main memory is expensive and data resides on disk. This assumption is no longer valid; over the last 30 years memory prices have dropped by a factor of 10 every 5 years. Today, one can buy a server with 32 cores and 1TB of memory for about $50K and both core counts and memory sizes are still increasing. 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. This paper gives a technical overview of the Hekaton design and reports a few performance results.

Several main memory database systems already exist, both commercial systems [5][15][18][19][21] and research prototypes [2][3][7][8] [16]. However, Hekaton has a number of features that sets it apart from the competition.

SIGMOD'13, June 22-27, 2013, New York, New York, USA.

Most importantly, the Hekaton engine is integrated into SQL Server; it is not a separate DBMS. To take advantage of Hekaton, all a user has to do is declare one or more tables in a database memory optimized. This approach offers customers major benefits compared with a separate main-memory DBMS. 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 be left unchanged. Third, stored procedures accessing only Hekaton tables can be compiled into native machine code for further performance gains. Fourth, 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 and two index types are available: hash indexes and range indexes. Hekaton tables are fully durable and transactional, though non-durable tables are also supported.

Hekaton tables can be 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 single 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. 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, multiversion concurrency control technique to avoid interference among transactions [9].

The rest of the paper is organized as follows. In section 2 we outline the considerations and principles behind the design of the engine. Section 3 provides a high-level overview of the architecture. Section 4 covers how data is stored, indexed, and updated. Section 5 describes how stored procedures and table definitions are compiled into native code. Section 6 covers transaction management and concurrency control while section 7 outlines how transaction durability is ensured. Section 8 describes garbage collection, that is, how versions no longer needed are handled. Section 9 provides some experimental results.

Terminology: We will use the terms *Hekaton table* and *Hekaton index* to refer to tables and indexes stored in main memory and managed by Hekaton. Tables and indexes managed by the traditional SQL Server engine will be called *regular tables* and *regular indexes*. Stored procedures that have been compiled to native machine code will simply be called *compiled stored procedures* and traditional non-compiled stored procedures will be called *interpreted stored procedures*.

2. DESIGN CONSIDERATIONS

An analysis done early on in the project drove home the fact that a 10-100X throughput improvement cannot be achieved by optimizing existing SQL Server mechanisms. Throughput can be increased in three ways: improving scalability, improving CPI (cycles per instruction), and reducing the number of instructions executed per request. The analysis showed that, even under highly optimistic assumptions, improving scalability and CPI can produce only a 3-4X improvement. The detailed analysis is included as an appendix.

The only real hope is to reduce the number of instructions executed but the reduction needs to be dramatic. To go 10X faster, the engine must execute 90% fewer instructions and yet still get the work done. To go 100X faster, it must execute 99% fewer instructions. This level of improvement is not feasible by optimizing existing storage and execution mechanisms. Reaching the 10-100X goal requires a much more efficient way to store and process data.

2.1 Architectural Principles

So to achieve 10-100X higher throughput, the engine must execute drastically fewer instructions per transaction, achieve a low CPI, and have no bottlenecks that limit scalability. This led us to three architectural principles that guided the design.

2.1.1 Optimize indexes for main memory

Current mainstream database systems 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. A simple key lookup in a B-tree index may require thousands of instructions even when all pages are in memory.

Hekaton indexes are designed and optimized for memory-resident data. 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.

2.1.2 Eliminate latches and locks

With the growing prevalence of machines with 100's of CPU cores, achieving good scaling 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 [4][6].

All Hekaton's internal data structures, for example, memory allocators, hash and range indexes, and transaction map, are entirely latch-free (lock-free). There are no latches or spinlocks on any performance-critical paths in the system. Hekaton uses a new optimistic multiversion concurrency control to provide transaction isolation semantics; there are no locks and no lock table [9]. The combination of optimistic concurrency control, multiversioning and latch-free data structures results in a system where threads execute without stalling or waiting.

2.1.3 Compile requests 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 statements and 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.

2.2 No Partitioning

HyPer [8], Dora [15], H-store [4], and VoltDB [21] 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. Hekaton does not partition the database and any thread can access any part of the database. We carefully evaluated a partitioned approach but rejected it.

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 quickly. It is not difficult to see why. Suppose we have a table that is partitioned on column A across 12 cores. The table has two (partitioned) indexes: an index on column A and a non-unique hash index on column B.

A query that includes an equality predicate on A (the partitioning column) can be processed quickly because only one partition needs to be accessed. However, a query that is not partition aligned can be very expensive. Consider a query that retrieves all records where B = 25. Some thread TH1 picks up the query and begins processing it. As the search predicate is not partition aligned and the index for B is not unique, all 12 partitions have to be checked. To do so TH1 has to enqueue a lookup request for each partition and wait for the results to be returned. Each request has to be dequeued by a receiving thread, processed, and the result returned.

The overhead of constructing, sending and receiving the request and returning the result is much higher than the actual work of performing a lookup in a hash table. In a non-partitioned system, thread TH1 would simply do the lookup itself in a single shared hash table. This is certainly faster and more efficient than sending 12 requests and doing 12 lookups.

After building from scratch and studying closely a prototype partitioned engine, we came to the conclusion that a partitioned approach is not sufficiently robust for the wide variety of workloads customers expect SQL Server to handle.

3. HIGH-LEVEL ARCHITECTURE

This section gives a high level overview of the various components of Hekaton and the integration with SQL Server. Later sections describe the components in more detail. 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 compile**r 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 managed by the Hekaton storage engine.
- The **Hekaton runtime system** is a relatively small component that provides integration with SQL Server resources and serves as a common library of additional functionality needed by compiled stored procedures.

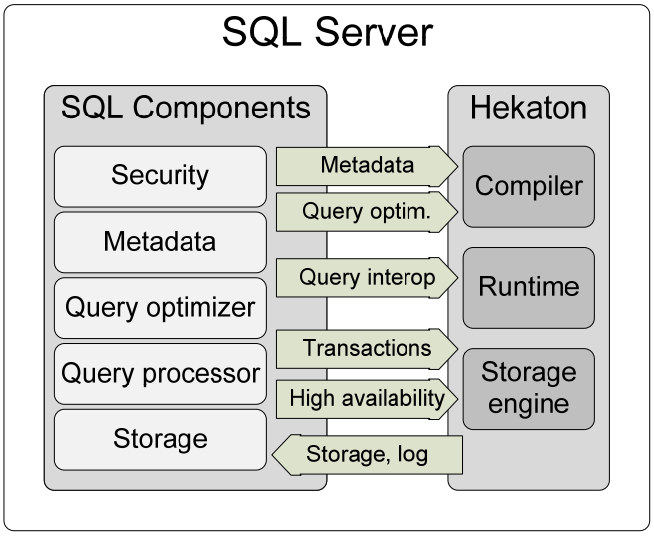


Figure 1: Hekaton's main components and integration into SQL Server.

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 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 and are also readable on secondary servers.
- **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 when a database is recovered.

4. STORAGE AND INDEXING

A table created with the new option memory_optimized is managed by Hekaton and stored entirely in memory. Hekaton supports two types of indexes: hash indexes which are implemented using lock-free hash tables [13] and range indexes which are implemented using Bw-trees, a novel lock-free version of B-trees [10]. A table can have multiple indexes and records are always accessed via an index lookup. Hekaton uses multiversioning; an update always creates a new version.

Figure 2 shows a simple bank account table containing six version records. Ignore the numbers (100) and text in red for now. The

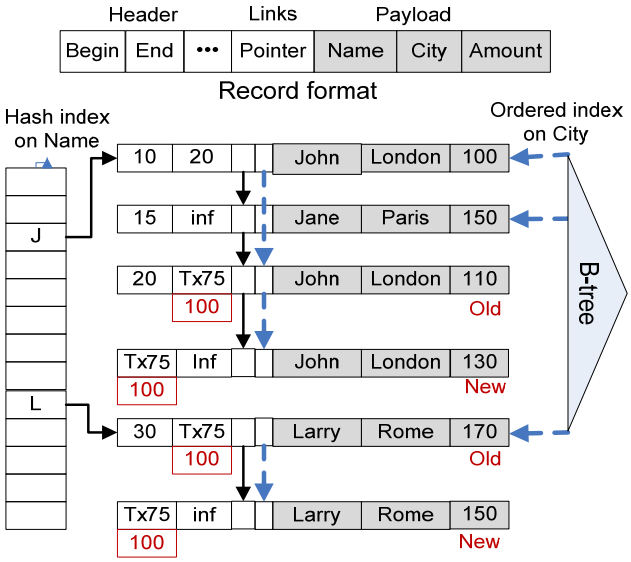


Figure 2: Example account table with two indexes. Transaction 75 has transferred $20 from Larry's account to John's account but has not yet committed.

table has three (user defined) columns: Name, City and Amount. A version record includes a header and a number of link (pointer) fields. A version's valid time is defined by timestamps stored in the Begin and End fields in the header.

The example table has two indexes; a hash index on Name and a range index on City. Each index requires a link field in the record. The first link field is reserved for the Name index and the second link field for the City index. For illustration purposes we assume that the hash function just picks the first letter of the name. Versions that hash to the same bucket are linked together using the first link field. The leaf nodes of the Bw-tree store pointers to records. If multiple records have the same key value, the duplicates are linked together using the second link field in the records and the Bw-tree points to the first record on the chain.

Hash bucket J contains four records: three versions for John and one version for Jane. Jane's single version (Jane, Paris, 150) has a valid time from 15 to infinity meaning that it was created by a transaction that committed at time 15 and it is still valid. John's oldest version (John, London, 100) was valid from time 10 to time 20 when it was updated. The update created a new version (John, London, 110). We will discuss John's last version (John, London, 130) in a moment.

4.1 Reads

Every read operation specifies a logical (as-of) read time and only versions whose valid time overlaps the read time are visible to the read; all other versions are ignored. Different versions of a record always have non-overlapping valid times so at most one version of a record is visible to a read. A lookup for John with read time 15, for example, would trigger a scan of bucket J that checks every record in the bucket but returns only the one with Name equal to John and valid time 10 to 20. If the index on Name is declared to be unique, the scan of the buckets stops as soon as a qualifying record has been found.

4.2 Updates

Bucket L contains two records that belong to Larry. Transaction 75 is in the process of transferring $20 from Larry's account to

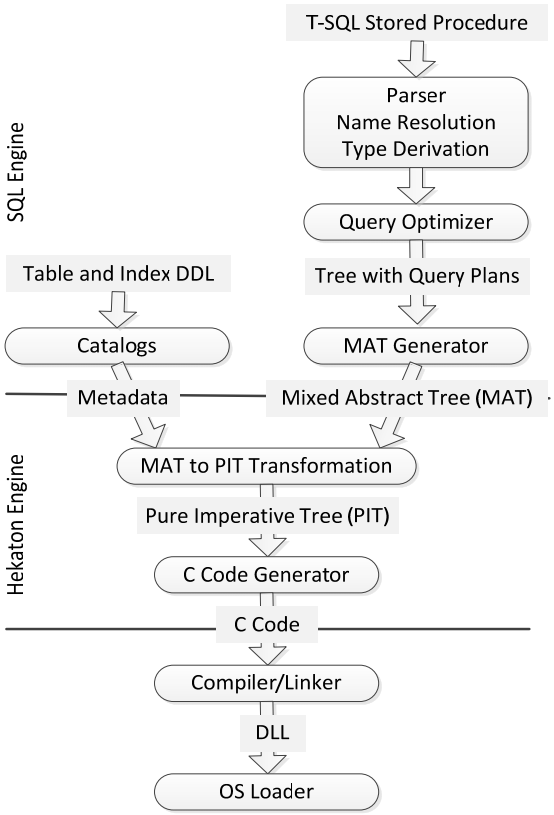


Figure 3: Architecture of the Hekaton compiler.

John's account. It has created the new versions for Larry (Larry, Rome, 150) and for John (John, London, 130) and inserted them into the appropriate buckets in the index.

Note that transaction 75 has stored its transaction Id in the Begin and End fields of the new and old versions, respectively. (One bit in the field indicates the field's content type.) A transaction Id stored in the End field prevents other transactions from updating the same version and it also identifies which transaction is updating the version. A transaction Id stored in the Begin field informs readers that the version may not yet be committed and identifies which transaction created the version.

Now suppose transaction 75 commits with end timestamp 100. (The details of commit processing are covered in section 6.) After committing, transaction 75 returns to the old and new versions and sets the Begin and End fields, respectively, to 100. The final values are shown in red below the old and new versions. The old version (John, London, 110) now has the valid time 20 to 100 and the new version (John, London, 130) has a valid time from 100 to infinity. Larry's record is updated in the same way.

This example also illustrates how deletes and inserts are handled because an update is equivalent to a deleting an old version and inserting a new version.

The system must discard obsolete versions that are no longer needed to avoid filling up memory. A version can be discarded when it is no longer visible to any active transaction. Cleaning out obsolete versions, a.k.a. garbage collection, is handled cooperatively by all worker threads. Garbage collection is described in more detail in section 8.

5. PROGRAMMABILITY AND QUERY PROCESSING

Hekaton maximizes run time performance by converting SQL statements and stored procedures into highly customized native code. Database systems traditionally use interpreter based execution mechanisms that perform many run time checks during the execution of even simple statements.

Our primary goal is to support efficient execution of compile-once-and-execute-many-times workloads as opposed to optimizing the execution of ad hoc queries. We also aim for a high level of language compatibility to ease the migration of existing SQL Server applications to Hekaton tables and compiled stored procedures. Consequently, we chose to leverage and reuse technology wherever suitable. We reuse much of the SQL Server T-SQL compilation stack including the metadata, parser, name resolution, type derivation, and query optimizer. This tight integration helps achieve syntactic and semantic equivalence with the existing SQL Server T-SQL language. The output of the Hekaton compiler is C code and we leverage Microsoft's Visual C/C++ compiler to convert the C code into machine code.

While it was not a goal to optimize ad hoc queries, we do want to preserve the ad hoc feel of the SQL language. Thus, a table and stored procedure is available for use immediately after it has been created. To create a Hekaton table or a compiled stored procedure, the user merely needs to add some additional syntax to the CREATE TABLE or CREATE PROCEDURE statement. Code generation is completely transparent to the user.

Figure 3 illustrates the overall architecture of the Hekaton compiler. There are two main points where we invoke the compiler: during creation of a memory optimized table and during creation of a compiled stored procedure.

As noted above, we begin by reusing the existing SQL Server compilation stack. We convert the output of this process into a data structure called the mixed abstract tree or MAT. This data structure is a rich abstract syntax tree capable of representing metadata, imperative logic, expressions, and query plans. We then transform the MAT into a second data structure called the pure imperative tree or PIT. The PIT is a much "simpler" data structure that can be easily converted to C code (or theoretically directly into the intermediate representation for a compiler backend such as Phoenix [17] or LLVM [11]). We discuss the details of the MAT to PIT transformation further in Section 5.2. Once we have C code, we invoke the Visual C/C++ compiler and linker to produce a DLL. At this point it is just a matter of using the OS loader to bring the newly generated code into the SQL Server address space where it can be executed.

5.1 Schema Compilation

It may not be obvious why table creation requires code generation. The reason is that the Hekaton storage engine treats records as opaque objects. It has no knowledge of the internal content or format of records and cannot directly access or process the data in records. The Hekaton compiler provides the engine with customized callback functions for each table. These functions perform tasks such as computing a hash function on a key or record, comparing two records, and serializing a record into a log buffer. Since these functions are compiled into native code, index operations such as inserts and searches are extremely efficient.

```
CREATE PROCEDURE SP_Example @id INT
WITH NATIVE_COMPILATION, SCHEMABINDING,
     EXECUTE AS OWNER
AS BEGIN ATOMIC
WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT,
      LANGUAGE = 'English')
   SELECT Name, Address, Phone
   FROM dbo.Customers WHERE Id = @id
```

Figure 4: Sample T-SQL procedure.

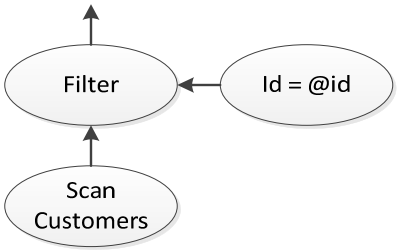


Figure 5: Query plan for sample T-SQL procedure

5.2 Compiled Stored Procedures

There are numerous challenging problems that we had to address to translate T-SQL stored procedures into C code. Perhaps the most obvious challenge is the transformation of query plans into C code and we will discuss our approach to this problem momentarily. There are, however, many other noteworthy complications. For example, the T-SQL and C type systems and expression semantics are very different. T-SQL includes many data types such as date/time types and fixed precision numeric types that have no corresponding C data types. In addition, T-SQL supports NULLs while C does not. Finally, T-SQL raises errors for arithmetic expression evaluation failures such as overflow and division by zero while C either silently returns a wrong result or throws an OS exception that must be translated into an appropriate T-SQL error.

These complexities were a major factor in our decision to introduce the intermediate step of converting the MAT into the PIT rather than directly generating C code. The PIT is a data structure that can be easily manipulated, transformed, and even generated out of order in memory. It is much more challenging to work directly with C code in text form.

The transformation of query plans into C code warrants further discussion. To aid in this discussion, consider the simple T-SQL example in Figure 4. This procedure retrieves a customer name, address, and phone number given a customer id. The procedure declaration includes some additional syntax; we will explain below why this syntax is required.

As with many query execution engines, we begin with a query plan which is constructed out of operators such as scans, joins, and aggregations. Figure 5 illustrates one possible plan for executing our sample query. For this example, we are naively assuming that the DBA has not created an index on Customer.Id and that the predicate is instead evaluated via a filter operator. In practice, we ordinarily would push the predicate down to the storage engine via a callback function. However, we use the filter operator to illustrate a more interesting outcome.

Each operator implements a common interface so that they can be composed into arbitrarily complex plans. In our case, this interface consists of "get first," "get next," "return row," and "return done." However, unlike most query execution engines, we do not implement these interfaces using functions. Instead, we collapse an entire query plan into a single function using labels and gotos to implement and connect these interfaces. Figure 6 illustrates graphically how the operators for our example are interconnected. Each hollow circle represents a label while each arrow represents a goto statement. In many cases, we can directly link the code for the various operators bypassing intermediate operators entirely. The X's mark labels and gotos that have been optimized out in just such a fashion. In conventional implementations, these same scenarios would result in wasted instructions where one operator merely calls another without performing any useful work.

Execution of the code represented by Figure 6 begins by transferring control directly to the GetFirst entry point of the scan operator. Note already the difference as compared to traditional query processors which typically begin execution at the root of the plan and invoke repeated function calls merely to reach the leaf of the tree even when the intermediate operators have no work to do. Presuming the Customers table is not empty, the scan operator retrieves the first row and transfers control to the filter operator ReturnRow entry point. The filter operator evaluates the predicate and either transfers control back to the scan operator GetNext entry point if the current row does not qualify or to the output operator entry point if the row qualifies. The output operator adds the row to the output result set to be returned to the client and then transfers control back to the scan operator GetNext entry point again bypassing the filter operator. When the scan operator reaches the end of the table, execution terminates immediately. Again control bypasses any intermediate operators.

This design is extremely flexible and can support any query operator including blocking (e.g., sort and group by aggregation) and non-blocking (e.g., nested loops join) operators. Our control flow mechanism is also flexible enough to handle operators such as merge join that alternate between multiple input streams. By keeping all of the generated code in a single function, we avoid costly argument passing between functions and expensive function calls. Although the resulting code is often challenging to read due in part to the large number of goto statements, it is important to keep in mind that our intent is not to produce code for human consumption. We rely on the compiler to generate efficient code. We have confirmed that the compiler indeed does so through inspection of the resulting assembly code.

We compared this design to alternatives involving multiple functions and found that the single function design resulted in the

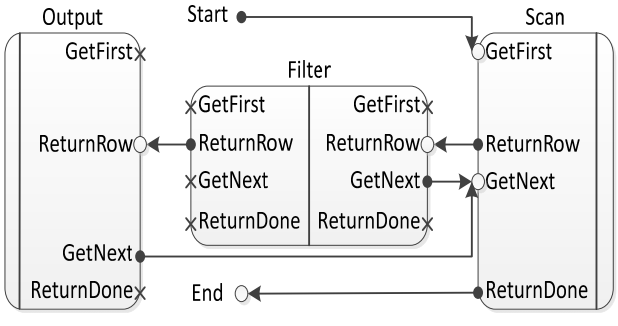


Figure 6: Operator interconnections for sample procedure.

fewest number of instructions executed as well as the smallest overall binary. This result was true even with function inlining. In fact, the use of gotos allows for code sharing within a single function. For example, an outer join needs to return two different types of rows: joined rows and NULL extended rows. Using functions and inlining with multiple outer joins, there is a risk of an exponential growth in code size [14]. Using gotos, the code always grows linearly with the number of operators.

There are cases where it does not make sense to generate custom code. For example, the sort operator is best implemented using a generic sort implementation with a callback function to compare records. Some functions (e.g., non-trivial math functions) are either sufficiently complex or expensive that it makes sense to include them in a library and call them from the generated code.

5.3 Restrictions

With minor exceptions, compiled stored procedures look and feel just like any other T-SQL stored procedures. We support most of the T-SQL imperative surface area including parameter and variable declaration and assignment as well as control flow and error handling. The query surface area is a bit more limited but we are expanding it rapidly. We support SELECT, INSERT, UPDATE, and DELETE. Queries currently can include inner joins, sort and top sort, and basic scalar and group by aggregation.

In an effort to minimize the number of run time checks and operations that must be performed at execution time, we do impose some limitations. First, compiled stored procedures support a limited set of options and the options that can be controlled must be set at compile time only. This policy eliminates unnecessary run time checks. Second, compiled stored procedures must execute in a predefined security context so that we can run all permission checks once at creation time. Third, compiled stored procedures must be schema bound; i.e., once a procedure is created, any tables referenced by that procedure cannot be dropped without first dropping the procedure. This avoids acquiring costly schema stability locks before execution. Fourth, compiled stored procedures must execute in the context of a single transaction. This requirement ensures that a procedure does not block midway through to wait for commit.

5.4 Query Interop

Compiled stored procedures do have limitations in the current implementation. The available query surface area is not yet complete and it is not possible to access regular tables from a compiled stored procedure. Recognizing these limitations, we implemented an additional mechanism that enables the conventional query execution engine to access memory optimized tables. This feature enables several important scenarios:

- Import and export of data to and from memory optimized tables using the existing tools and processes that already work for regular tables.
- Support for ad hoc queries and data repair.
- Feature completeness. E.g., users can leverage interop to execute virtually any legal SQL query against memory optimized tables and can use features such as views and cursors that are not supported in compiled stored procedures.
- Support for transactions that access both memory optimized and regular tables.
- Ease of app migration. Existing tables can be converted to memory optimized tables without extensive work to convert existing stored procedures into compiled stored procedures.

6. TRANSACTION MANAGEMENT

Hekaton utilizes optimistic multiversion concurrency control (MVCC) to provide snapshot, repeatable read and serializable transaction isolation without locking. This section summarizes the core concepts of the optimistic MVCC implemented in Hekaton. Further details can be found in [9].

A transaction is by definition serializable if its reads and writes logically occur as of the same time. The simplest and most widely used MVCC method is snapshot isolation (SI). SI 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. However, 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.

To ensure that a transaction T is serializable we must ensure that the following two properties hold.

1. **Read stability**. If T reads 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. This ensures that nothing has disappeared from the view.
2. **Phantom avoidance**. 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. This ensures that nothing has been added to the view.

Lower isolation levels are easier to support. For repeatable read, we only need to guarantee read stability. For snapshot isolation or read committed, no validation at all is required.

The discussion that follows covers only serializable transactions. For other isolation levels and a lock-based alternative technique, see [9].

6.1 Timestamps and Version Visibility

Timestamps produced by a monotonically increasing counter are used to specify the following.

- **Logical Read Time**: the read time of a transaction 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 transaction. For all supported isolation levels, the logical read time of a transaction is set to the start time of the transaction.
- **Commit/End Time** for a transaction: every transaction that modifies data commits at a distinct point in time called the commit or end timestamp of the transaction. The commit time determines a transaction's position in the serialization history.
- **Valid Time** for a version of a record: All records in the database contain two timestamps – begin and end. 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 version of a record denotes the timestamp range where the version is visible to other transactions.

The notion of version visibility is fundamental to proper concurrency control in Hekaton. 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. A transaction must of course also see its own updates.

6.2 Transaction Commit Processing

Once a transaction has completed its normal processing, it begins commit processing.

6.2.1 Validation and Dependencies

At the time of commit, a serializable transaction must verify that the versions it read have not been updated and that no phantoms have appeared. The validation phase begins with the transaction obtaining an end timestamp. This end timestamp determines the position of the transaction within the transaction serialization history.

To validate its reads, the transaction checks that the versions it read are visible as of the transaction's end time. To check for phantoms, it repeats all its index scans looking for versions that have become visible since the transaction began. To enable validation each transaction maintains a *read set*, a list of pointers to the versions it has read, and a *scan set* containing information needed to repeat scans. While validation may sound expensive, keep in mind that most likely the versions visited during validation remain in the L1 or L2 cache. Furthermore, validation overhead is lower for lower isolation: repeatable read requires only read validation and snapshot isolation and read committed require not validation at all.

The validation phase is a time of uncertainty for a transaction. If the validation succeeds the transaction is likely to commit and if it commits its effects must be respected by all other transactions in the system as if they occurred atomically as of the end timestamp. If validation fails, then nothing done by the transaction must be visible to any other transaction.

Any transaction T1 that begins while a transaction T2 is in the validation phase becomes dependent on T2 if it attempts to read a version created by T2 or ignores a version deleted by T2. In that case T1 has two choices: block until T2 either commits or aborts, or proceed and take a *commit dependency* on T2. To preserve the non-blocking nature of Hekaton, we have T1 take a commit dependency on T2. This means that T1 is allowed to commit only if T2 commits. If T2 aborts, T1 must also abort so cascading aborts are possible.

Commit dependencies introduce two problems: 1) a transaction cannot commit until every transaction upon which it is dependent has committed and 2) commit dependencies imply working with uncommitted data and such data should not be exposed to users.

When a transaction T1 takes a commit dependency on transaction T2, T2 is notified of the dependency and T1 increments its dependency count. If T2 commits, it decrements the dependency count in T1. If T2 rolls back, it notifies T1 that it too must roll back. If T1 attempts to commit and completes its validation phase (which may itself acquire additional commit dependencies) and it still has outstanding commit dependencies, it must wait for the commit dependencies to clear. If all transactions upon which T1 is dependent commit successfully, then T1 can proceed with logging its changes and completing post processing.

To solve the second problem we introduced read barriers. This simply means that a transaction's result set is held back and not delivered to the client while the transaction has outstanding commit dependencies. The results are sent as soon as the dependencies have cleared.

6.2.2 Commit Logging and Post-processing

A transaction T is committed as soon as its changes to the database have been hardened to the transaction log. Transaction T writes to the log the contents of all new versions created by T and the primary key of all versions deleted by T. More details on logging can be found in section 7.

Once T's updates have been successfully logged, it is irreversibly committed. T then begins a post-processing phase during which the begin and end timestamps in all versions affected by the transaction are updated to contain the end timestamp of the transaction. Transactions maintain a *write-set*, a set of pointers to all inserted and deleted versions that is used to perform the timestamp updates and generate the log content.

6.2.3 Transaction Rollback

Transactions can be rolled back at user request or due to failures in commit processing. Rollback is achieved by invalidating all versions created by the transaction and clearing the end-timestamp field of all versions deleted by the transaction. If there are any other transactions dependent on the outcome of the rolled-back transaction, they are so notified. Again the write-set of the transaction is used to perform this operation very efficiently.

7. TRANSACTION DURABILITY

While Hekaton is optimized for main-memory resident data, it must ensure transaction durability that allows it to recover a memory-optimized table after a failure. Hekaton achieves this using transaction logs and checkpoints to durable storage. Though not covered in this paper, Hekaton is also integrated with the AlwaysOn component that maintains highly available replicas supporting failover.

The design of the logging, checkpointing and recovery components was guided by the following principles.

- Optimize for sequential access so customers can spend their money on main memory rather than I/O devices with fast random access.
- Push work to recovery time to minimize overhead during normal transaction execution.
- Eliminate scaling bottlenecks.
- Enable parallelism in I/O and CPU during recovery

The data stored on external storage consists of transaction log streams and checkpoint streams.

- *Log streams* contain the effects of committed transactions logged as insertion and deletion of row versions.
- *Checkpoint streams* come in two forms: a) *data streams* which contain all inserted versions during a timestamp interval, and b) *delta streams*, each of which is associated with a particular data stream and contains a dense list of integers identifying deleted versions for its corresponding data stream.

The combined contents of the log and checkpoint streams are sufficient to recover the in-memory state of Hekaton tables to a transactionally consistent point in time. Before we discuss the details of how they are generated and used, we first summarize a few of their characteristics.

- Log streams are stored in the regular SQL Server transaction log. Checkpoint streams are stored in SQL Server file streams which in essence are sequential files fully managed by SQL Server.

- The log contains the logical effects of committed transactions sufficient to redo the transaction. The changes are recorded as insertions and deletions of row versions labeled with the table they belong to. No undo information is logged.
- Hekaton index operations are not logged. All indexes are reconstructed on recovery.
- Checkpoints are in effect a compressed representation of the log. Checkpoints allow the log to be truncated and improve crash recovery performance.

7.1 Transaction Logging

Hekaton's transaction log is designed for high efficiency and scale. Each transaction is logged in a single, potentially large, log record. The log record contains information about all versions inserted and deleted by the transaction, sufficient to redo them.

Since the tail of the transaction log is typically a bottleneck, reducing the number of log records appended to the log can improve scalability and significantly increase efficiency. Furthermore the content of the log for each transaction requires less space than systems that generate one log record per operation.

Generating a log record only at transaction commit time is possible because Hekaton does not use write-ahead logging (WAL) to force log changes to durable storage before dirty data. Dirty data is never written to durable storage. Furthermore, Hekaton tries to group multiple log records into one large I/O; this is the basis for group commit and also a significant source of efficiency for Hekaton commit processing.

Hekaton is designed to support multiple concurrently generated log streams per database to avoid any scaling bottlenecks with the tail of the log. Multiple log streams can be used because serialization order is determined solely by transaction end timestamps and not by ordering in the transaction log. However the integration with SQL Server leverages only a single log stream per database (since SQL Server only has one). This has so far proven sufficient because Hekaton generates much less log data and fewer log writes compared with SQL Server.

7.2 Checkpoints

To reduce recovery time Hekaton also implements checkpointing. The checkpointing scheme is designed to satisfy two important requirements.

- **Continuous checkpointing.** Checkpoint related I/O occurs incrementally and continuously as transactional activity accumulates. Customers complain that 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) are disruptive to overall system performance.
- **Streaming I/O.** Checkpointing 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 and can incur more CPU overhead due to smaller individual I/O requests.

7.2.1 Checkpoint Files

Checkpoint data is stored in two types of checkpoint files: data files and delta files. A complete checkpoint consists of multiple data and delta files and a checkpoint file inventory that lists the files comprising the checkpoint.

A **data file** contains only inserted versions (generated by inserts and updates) covering 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 opened and once closed, they are strictly read-only. At recovery time the versions in data files are reloaded into memory and re-indexed, subject to filtering by delta files discussed next.

A delta file stores information about which versions contained in a data file have been subsequently deleted. There is a 1:1 correspondence between a delta file and a data file. 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. The choice to pair one delta file with each data file means that the smallest unit of work for recovery is a data/delta file pair leading to a recovery process that is highly parallelizable.

A **checkpoint file inventory** tracks references to all the data and delta files that make up a complete checkpoint. The inventory is stored in a system table.

A complete checkpoint combined with the tail of the transaction log enable Hekaton tables to be recovered. A checkpoint has a timestamp which indicates that the effects of all transactions before the checkpoint timestamp are recorded in the checkpoint and thus the transaction log is not needed to recover them.

7.2.2 Checkpoint Process

A checkpoint task takes a section of the transaction log not covered by a previous checkpoint and converts the log contents into one or more data files and updates to delta files. New versions are appended to either the most recent data file or into a new data file and the IDs of deleted versions are appended to the delta files corresponding to where the original inserted versions are stored. Both operations are append-only and can be done buffered to allow for large I/Os. Once the checkpoint task finishes processing the log, the checkpoint is completed with the following steps.

1. Flush all buffered writes to the data and delta files and wait for them to complete.
2. Construct a checkpoint inventory that includes all files from the previous checkpoint plus any files added by this checkpoint. Harden the inventory to durable storage.
3. Store the location of the inventory in a durable location available at recovery time. We record it both in the SQL Server log and the root page of the database.

The set of files involved in a checkpoint grows with each checkpoint. However the active content of a data file degrades as more and more of its versions are marked deleted in its delta file. Since crash recovery will read the contents of all data and delta files in the checkpoint, performance of crash recovery degrades as the utility of each data file drops.

The solution to this problem is to *merge* temporally adjacent data files 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, that is, versions identified in the DF1 and DF2's delta files, are dropped during the merge. The delta file for DF3 is empty immediately after the merge.

7.3 Recovery

Hekaton recovery starts after the location of the most recent checkpoint inventory has been recovered during a scan of the tail of the log. Once the SQL Server host has communicated the loca-

tion of the checkpoint inventory to the Hekaton engine, SQL Server and Hekaton recovery proceed in parallel.

Hekaton recovery itself is parallelized. Each delta file represents in effect a filter for rows that need not be loaded from the corresponding data file. This data/delta file pair arrangement means that checkpoint load can proceed in parallel across multiple IO streams at file pair granularity. The Hekaton engine takes advantage of parallel streams for load I/O, but also creates one thread per core to handle parallel insertion of the data produced by the I/O streams. The insert threads in effect replay the transactions saved in the checkpoint files. The choice of one thread per core means that the load process is performed as efficiently as possible.

Finally, once the checkpoint load process completes, the tail of the transaction log is replayed from the timestamp of the checkpoint, with the goal of bringing the database back to the state that existed at the time of the crash.

8. GARBAGE COLLECTION

Multiversioning systems inevitably face the question of how to cleanup versions that are no longer visible to running transactions. We refer to this activity as garbage collection. While the term garbage collection can conjure visions of poor performance, lengthy pause times, blocking, and other scaling problems often seen in the virtual machines for managed languages, Hekaton avoids these problems. Unlike a programming language runtime where the notion of garbage is defined by the "reachability" of a pointer from any location in the process address space, in Hekaton, garbage is defined by a version's "visibility" – that is, a version of a record is garbage if it is no longer visible to any active transaction.

The design of the Hekaton garbage collection (GC) subsystem has the following desirable properties.

- Hekaton GC is **non-blocking**. Garbage collection runs concurrently with the regular transaction workload, and never stalls processing of any active transaction.
- The GC subsystem is **cooperative.** Worker threads running the transaction workload can remove garbage when they encounter it. This can save processing time as garbage is removed proactively whenever it is “in the way” of a scan.
- Processing is **incremental**. Garbage collection may easily be throttled and can be started and stopped to avoid consuming excessive CPU resources.
- Garbage collection is **parallelizable and scalable**. Multiple threads can work in parallel on various phases of garbage collection and in isolation with little to no cross-thread synchronization.

8.1 Garbage Collection Details

The garbage collection process is described in this section to illustrate how these properties are achieved.

8.1.1 GC Correctness

First, care must be taken to identify which versions might be garbage. Potential garbage versions may be created by one of two processes. First, a version becomes garbage if a) it was deleted (via explicit DELETE or through an UPDATE operation) by a committed transaction and b) the version cannot be read or otherwise acted upon by any transaction in the system. A second, and less common way for versions to become garbage is if they were created by a transaction that subsequently rolls back.

The first and most important property of the GC is that it correctly determines which versions are actually garbage. The visibility of a version is determined by its begin and end timestamps. Any version whose end timestamp is less than the current oldest active transaction in the system is not visible to any transaction and can be safely discarded.

A GC thread periodically scans the global transaction map to determine the *begin* timestamp of the oldest active transaction in the system. When the GC process is notified that it should begin collection, transactions committed or aborted since the last GC cycle are ordered by their end timestamps. Any transaction T in the system whose end timestamp is older than the oldest transaction watermark is ready for collection. More precisely, the versions deleted or updated by T can be garbage collected because they are invisible to all current and future transactions.

8.1.2 Garbage Removal

In order for a garbage version to be removed it must first be unlinked from all indexes in which it participates. The GC subsystem collects these versions in two ways: (1) a cooperative mechanism used by threads running the transaction workload, and (2) a parallel, background collection process.

Since regular index scanners may encounter garbage versions as they scan indexes, index operations are empowered to unlink garbage versions when they encounter them. If this unlinks a version from its last index, the scanner may also reclaim it. This cooperative mechanism is important in two dimensions. First, it naturally parallelizes garbage collection in the system, and makes collection efficient since it piggybacks on work the scanner already did to locate the version. Second, it ensures that old versions will not slow down future scanners by forcing them to skip over old versions encountered, for example, in hash index bucket chains.

This cooperative process naturally ensures that 'hot' regions of an index are constantly maintained to ensure they are free of obsolete versions. However, this process is insufficient to ensure that either (1) 'cold' areas of an index which are not traversed by scanners are free of garbage, or that (2) a garbage version is removed from other indexes that it might participate in. Versions in these “dusty corners” (infrequently visited index regions) do not need to be collected for performance reasons, but they needlessly consume memory and, as such, should be removed as promptly as possible. Since reclamation of these versions is not time critical, the work to collect these versions is offloaded to a background GC process.

Each version examined by the background collection process may potentially be removed immediately. If the version no longer participates in any index (because cooperative scanners have removed it) it can be reclaimed immediately. However, if GC finds a version that is still linked in one or more indexes, it cannot immediately unlink the version since it has no information about the row's predecessor. In order to remove such versions, GC first scans the appropriate part of each index and unlinks the version, after which it can be removed. While scanning, it of course unlinks any other garbage versions encountered.

8.1.3 Scalability

Early versions of the Hekaton GC used a fixed set of threads for collection. A main GC thread was responsible for processing garbage versions and attempting to directly unlink those that it could, while two auxiliary threads were used to perform the 'dusty corner' scans to remove versions that needed additional work.

However, under high transaction loads, we found that it was difficult to ensure that a single GC thread could maintain the necessary rate of collection for a high number of incoming transactions, especially for those workloads that were more update/delete heavy. In order to address this problem, the garbage collection has been parallelized across all worker threads in the system.

A single GC process is still responsible for periodically recalculating the oldest transaction watermark and partitioning completed transactions accordingly. However, once this work has been done, transactions that are ready for collection are then distributed to a set of work queues. After a Hekaton worker has acknowledged its commit or abort to the user, it then picks up a small set of garbage collection work from its CPU-local queue, and completes that work. This serves two scalability benefits. First, it naturally parallelizes the work across CPU cores, without the additional overhead and complexity of maintaining dedicated worker threads, and second, it allows the system to self-throttle. By ensuring that each thread in the system that is responsible for user work is also responsible for GC work, and by preventing a user thread from accepting more transactional work until a bit of garbage has been collected, this scheme introduces a small delay in the processing of transactions in the system, making sure that the system does not generate more garbage versions than the GC subsystem can retire.

9. EXPERIMENTAL RESULTS

9.1 CPU Efficiency

The Hekaton engine is significantly more efficient than the regular SQL Server engine; it processes a request using far fewer instructions and CPU cycles. The goal of our first experiments is to quantify the improvement in CPU efficiency of the core engine and how this depends on request size. The experiments in this section were run on a workstation with a 2.67GHz Intel Xeon W3520 processor, 6 GB of memory and an 8 MB L2 cache.

For the experiments we created two identical tables, T1 and T2, with schema (c1 int, c2 int, c3 varchar(32)), each containing 1M rows. Column c1 is the primary key. T1 was a Hekaton table with a hash index on c1 and T2 was a regular table with a B-tree index on c1. Both tables resided entirely in memory.

9.1.1 Lookup Efficiency

We created a T-SQL procedure RandomLookups that does *N* random lookups on the primary key (column c1) and computes the average, min, and max of column c2. The key values are randomly generated using the T-SQL function RAND(). There are two versions of the procedure, one doing lookups in T1 and compiled into native code and one doing lookups in T2 using the regular SQL Server engine.

As the goal was to measure and compare CPU efficiency of the core engines, we used another (regular) stored procedure as the driver for RandomLookups. The driver calls RandomLookups in a loop and computes the average CPU cycles consumed per call. The results are shown in Table 1 for different numbers of lookups per call.

The speedup is 20X when doing 10 or more lookups per call. Expressed differently, the Hekaton engine completed the same work using 5% of the CPU cycles used by the regular SQL Server engine. The fixed overhead of the stored procedure (call and return, create and destroy transaction, *etc.*) dilutes the speedup for few lookups. For a single lookup the speedup is 10.8X.

The absolute lookup performance is very high. Finishing 100,000 lookups in 98.1 million cycles on a 2.67GHz core equals 2.7M lookups per second per core.

9.1.2 Update Efficiency

To measure the CPU efficiency of updates we wrote another T-SQL procedure RandomUpdates that updates the c2 column of *N* randomly selected rows. Again, there are two versions of the procedure, one compiled into native code and updating T1, and one regular procedure updating T2. We varied the number of updates per transaction from 1 to 10,000. The results are shown in Table 2.

As the goal was to measure CPU efficiency and not transaction latency, we enabled write caching on the disk used for the transaction log. With write caching disabled, we would essentially have measured the write latency of the disk which is not the information we were after. CPU efficiency is largely independent of the type of logging device.

The speedup is even higher than for lookups, reaching around 30X for transactions updating 100 or more records. Even for transactions consisting of a single updated, the speedup was around 20X. In other words, Hekaton got the work done using between 3% and 5% of the cycles used by the regular engine.

Again, the absolute performance is very high.10,000 updates in 14.4 million cycles equals about 1.9M updates per second using a single core.

As mentioned earlier, Hekaton generally logs less data than the regular SQL Server engine. In this particular case, it reduced log output by 57%. However, how much Hekaton logs depends on the record size; for large records the relative gain would be smaller.

9.2 Scaling Under Contention

Scalability of database systems is often limited by contention on locks and latches [5]. The system is simply not able to take ad-

Table 1: Comparison of CPU efficiency for lookups.

Transaction size in #lookups	CPU cycles (in millions)		Speedup
	Interpreted	Compiled	
1	0.734	0.040	10.8X
10	0.937	0.051	18.4X
100	2.72	0.150	18.1X
1,000	20.1	1.063	18.9X
10,000	201	9.85	20.4X

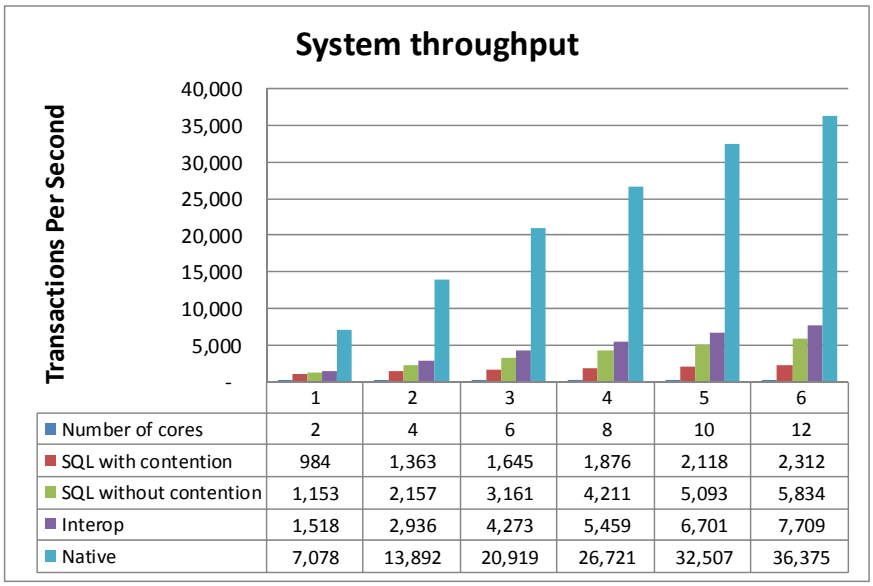
Table 2: Comparison of CPU efficiency for updates.

Transaction size in #updates	CPU cycles (in millions)		Speedup
	Interpreted	Compiled	
1	0.910	0.045	20.2X
10	1.38	0.059	23.4X
100	8.17	0.260	31.4X
1,000	41.9	1.50	27.9X
10,000	439	14.4	30.5X

vantage of additional processor cores so throughput levels off or even decreases. When SQL Server customers experience scalability limitations, the root cause is frequently contention.

Hekaton is designed to eliminate lock and latch contention, allowing it to continue to scale with the number of processor cores. The next experiment illustrates this behavior. This experiment simulates 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 latch contention.

The problem is caused by a SalesOrderDetails table 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.



	1	2	3	4	5	6
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 7: Experiment illustrating the scalability of the Hekaton engine. Throughput for the regular SQL Server engine is limited by latch contention.

This 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.

Figure 7 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 core to 12 cores throughput increases from 984 to 2,312 transactions per second, only 2.3X. Latch contention limits the CPU utilization to just 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, a total increase of 15.7X.

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 were wondering what the performance of the regular SQL Server engine would be if there were 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. Removing contention improved scaling significantly from 2.3X to 5.1X going from 2 cores to 12 cores.

10. Concluding Remarks

Hekaton is a new database engine targeted for OLTP workloads under development at Microsoft. It is optimized for large main memories and many-core processors. It is fully integrated into SQL Server, which allows customers to gradually convert their most performance-critical tables and applications to take advantage of the very substantial performance improvements offered by Hekaton.

Hekaton achieves its high performance and scalability by using very efficient latch-free data structures, multiversioning, a new optimistic concurrency control scheme, and by compiling T-SQL stored procedure into efficient machine code. Transaction durability is ensured by logging and checkpointing to durable storage. High availability and transparent failover is provided by integration with SQL Server's AlwaysOn feature.

As evidenced by our experiments, the Hekaton engine delivers more than an order of magnitude improvement in efficiency and scalability with minimal and incremental changes to user applications or tools.

11. REFERENCES

[1] Florian Funke, Alfons Kemper, Thomas Neumann: HyPersonic Combined Transaction AND Query Processing. PVLDB 4(12): 1367-1370 (2011)

[2] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, Samuel Madden: HYRISE - A Main Memory Hybrid Storage Engine. PVLDB 4(2): 105-116 (2010)

[3] Martin Grund, Philippe Cudré-Mauroux, Jens Krüger, Samuel Madden, Hasso Plattner: An overview of HYRISE - a Main Memory Hybrid Storage Engine. IEEE Data Eng. Bull. 35(1): 52-57 (2012)

[4] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker: OLTP through the looking glass, and what we found there. SIGMOD 2008: 981-992

[5] IBM SolidDB, http://www.ibm.com/software/data/soliddb

[6] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, Babak Falsafi: Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35

[7] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, Daniel J. Abadi: H-store: a high-performance, distributed main memory transaction processing system. PVLDB 1(2): 1496-1499 (2008)

[8] Alfons Kemper, Thomas Neumann: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE 2011: 195-206

[9] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, Mike Zwilling: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309 (2011)

[10] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, The Bw-Tree: A B-tree for New Hardware Platforms, ICDE 2013 (to appear).

[11] The LLVM Compiler Infrastructure, http://llvm.org/

[12] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Trans. Parallel Distrib. Syst. 15, 6 (June 2004), 491-504.

[13] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '02): 73-82.

[14] Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 4(9): 539-550 (2011)

[15] Oracle TimesTen, http://www.oracle.com/technetwork/products/timesten/overview/index.html

[16] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, Anastasia Ailamaki: Data-Oriented Transaction Execution. PVLDB 3(1): 928-939 (2010)

[17] Phoenix compiler framework, http://en.wikipedia.org/wiki/Phoenix_(compiler_framework)

[18] SAP In-Memory Computing, http://www.sap.com/solutions/technology/in-memory-computing-platform/hana/overview/index.epx

[19] Sybase In-Memory Databases, http://www.sybase.com/manage/in-memory-databases

[20] Håkan Sundell, Philippas Tsiga, Lock-free deques and doubly linked lists, Journal of Parallel and Distributed Computing - JPDC , 68(7): 1008-1020, (2008)

[21] VoltDB, http://voltdb.com

12. Appendix

Building an engine that is 10 to 100 times faster than SQL Server today required development of fundamentally new techniques. The analysis below shows why the goal cannot be achieved by optimizing existing mechanisms.

The performance of any OLTP system can be broken down into base performance and a scalability factor as follows.

$$SP = BP * SF^{lg(N)}$$

where

BP = performance of a single core in business transactions,
SF = scalability factor,
lg(N) = log base two of the number of cores in the system, and
SP = system performance in business transactions.

BP can be expressed as the product of CPI (cycles per instruction) and instructions retired, *IR*. The equation can then be expressed as

$$SP = IR * CPI * SF^{lg(N)}.$$

With a CPI of less than 1.6 achieved on some common OLTP benchmarks, SQL Server's runtime efficiency is fairly high for a commercial software system. More importantly, however, it means that CPI improvements alone cannot deliver dramatic performance gains. Even an outstanding CPI of 0.8, for instance, would barely double the server's runtime performance, leaving us still far short of the 10-100X goal.

SQL Server also scales quite well. On the TPC-E benchmark it has a scalability factor of 1.89 up to 256 cores; that is, for every doubling of cores, throughput increases by a factor of 1.89. This again implies that limited performance gains can be achieved by improving scalability. At 256 cores, SQL Server throughput increases to $(1.89)^8 = 162.8$. With perfect scaling the throughput increase would be 256 but that would only improve throughput over SQL Server by a factor of $256/162.8 = 1.57$.

Combining the best case CPI and scalability gains improves performance at most by a factor of $2*1.57 = 3.14$. So even under the most optimistic assumptions, improving CPI and scalability cannot yield orders-of-magnitude speedup. The only real hope is to drastically reduce the number of instructions executed.