Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database

Ahmed Eldawy*
University of Minnesota
eldawy@cs.umn.edu

Justin Levandoski
Microsoft Research
justin.levandoski@microsoft.com

Per-Åke Larson
Microsoft Research
palarson@microsoft.com

ABSTRACT

Main memories are becoming sufficiently large that most OLTP databases can be stored entirely in main memory, but this may not be the best solution. OLTP workloads typically exhibit skewed access patterns where some records are hot (frequently accessed) but many records are cold (infrequently or never accessed). It is still more economical to store the coldest records on secondary storage such as flash. This paper introduces Siberia, a framework for managing cold data in the Microsoft Hekaton main-memory database engine. We discuss how to migrate cold data to secondary storage while providing an interface to the user to manipulate both hot and cold data that hides the actual data location. We describe how queries of different isolation levels can read and modify data stored in both hot and cold stores without restriction while minimizing number of accesses to cold storage. We also show how records can be migrated between hot and cold stores while the DBMS is online and active. Experiments reveal that for cold data access rates appropriate for main-memory optimized databases, we incur an acceptable 7-14% throughput loss.

1. INTRODUCTION

Database systems have traditionally been designed under the assumption that data is disk resident and paged in and out of memory as needed. However, the drop in memory prices over the past 30 years is invalidating this assumption. Several database engines have emerged that store the entire database in main memory [3, 5, 7, 9, 11, 14, 19].

Microsoft has developed a memory-optimized database engine, code named Hekaton, targeted for OLTP workloads. The Hekaton engine is fully integrated into SQL Server and shipped in the 2014 release. It does not require a database be stored entirely in main memory; a user can declare only some tables to be in-memory tables managed by Hekaton. Hekaton tables can be queried and updated in the same way as regular tables. To speed up processing even further, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code. Further details about the design of Hekaton can be found in [4], [11].

OLTP workloads often exhibit skewed access patterns where some records are "hot" and accessed frequently (the working set) while others are "cold" and accessed infrequently. Clearly, good performance depends on the hot records residing in memory. Cold records can be moved to cheaper external storage such as flash with little effect on overall system performance.

The initial version of Hekaton requires that a memory-optimized table fits *entirely* in main memory. However, even a frequently accessed table may exhibit access skew where only a small fraction of its rows are hot while many rows are cold. We are investigating techniques to automatically migrate cold rows to a "cold store" residing on external storage while the hot rows remain in the in-memory "hot store". The separation into two stores is only visible to the storage engine; the upper layers of the engine (and applications) are entirely unaware of where a row is stored.

The goal of our project, called Project Siberia, is to enable the Hekaton engine to automatically and transparently maintain cold data on cheaper secondary storage. We divide the problem of managing cold data into four subproblems.

- *Cold data classification*: efficiently and non-intrusively identify hot and cold data. We propose to do this by logging record accesses, possibly only a sample, and estimating accesses frequencies off line as described in more detail in [13]. One could also use a traditional caching approach such as LRU or LRU-2 but the overhead is high in both space and time. As reported in [13], experiments showed that maintaining a simple LRU chain added 25% overhead to the cost of lookups in an in-memory hash table and added 16 bytes to each record. This we deemed too high a price.
- *Cold data storage*: evaluation of cold storage device options and techniques for organizing data on cold storage.
- *Cold storage access reduction*: reducing unnecessary accesses to cold storage for both point and range lookups by maintaining compact and accurate in-memory access filters. We propose to achieve this by storing in memory compact summaries of the cold store content. We are investigating two techniques: a version of Bloom filters for point lookups and range filters, a new compact data structure that also supports range queries. More details can be found in [1, 17].
- *Cold data access and migration mechanisms*: mechanisms for efficiently migrating, reading, and updating data on cold storage that dovetail with Hekaton's optimistic multi-version concurrency control scheme [11].

In this paper, we focus on the fourth point, namely, how to migrate records to and from the cold store and how to access and update records in the cold store in a transactionally consistent manner. This paper is *not* concerned with exact indexing and storage mechanisms used; all we assume is that the cold store provides methods for inserting, deleting, and retrieving records. To allow for maximum flexibility in the choice of cold store implementations our only

* Work done while at Microsoft Research

 Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 11

additional requirement is that the cold store guarantees durability, i.e, that it does not lose data even in the event of crashes. We do not require that the cold store be transactional. Thus, our design can work with a number of cold store implementations, for example, a traditional DBMS table, a key-value store, or even a file.

The basic idea of our approach is to completely separate hot and cold data into separate stores. We do not store information about cold records in memory (e.g., cold keys in in-memory indexes) besides compact Bloom or range filters. Transactions are free to access and update data in either store without restriction. In fact, our design hides the details of the hot/cold storage split beneath the interface to the Hekaton storage engine. Upper software layers are unaffected by the split between hot and cold storage.

Siberia is designed to fully integrate with Hekaton's optimistic multi-version concurrency control (MVCC) scheme. The basic idea of this scheme is that records are multi-versioned and versions have disjoint valid time ranges. Transactions read records as of a logical read time, while record updates create a new version. Reads operate by first checking the in-memory "hot" table. If the key is not found or the lookup index is not unique, the read then checks a Bloom filter to see if it needs to access the cold store; if so it performs the cold read. A transaction keeps records read from the cold store in a private in-memory cache.

Siberia guarantees transactional consistency for updates that span the hot and cold store (e.g., when data moves from hot to cold storage or vice versa) even if the cold store is *not* transactional. To achieve this we use an *update memo*, implemented as a durable Hekaton table that temporarily stores information about records whose update spans hot and cold stores. When a transaction updates a cold record, the record moves *from the cold to hot store*. The new version is placed in the hot store, while the transaction also records a "delete" notice in the memo signifying that the cold version will eventually become obsolete. The cold record and its corresponding notice are deleted once the version in the cold store is no longer visible to any active transaction. Records move from the *hot to cold store* using a migration process that updates a stale hot record and writes a new but identical version to the cold store. The migrating transaction also posts a "migrated" notice to the memo that records the valid begin time of the record on cold storage. The "migrated" notice is deleted once the version in the cold store becomes visible to all transactions. The old version in the hot store is removed automatically by Hekaton's garbage collection mechanism.

While this may seem complex at first glance, the update memo provides several important benefits. (1) The cold store is not required to be transactional. (2) It enables live migration of data to and from cold storage while the database is online and active. (3) The timestamp updates done during postprocessing of a transaction are done entirely in memory without accessing cold storage. (4) Validation is done entirely in memory without accessing cold storage (validation and postprocesing details of Hekaton are covered in Section 3.1). The last two points are particularly salient, as they allow Sibieria to make only the absolute minimal accesses to cold storage, i.e., for reading, inserting, or deleting cold records.

We prototyped Siberia within the Hekaton engine and performed extensive experiment evaluation. We find that for cold data access rates appropriate for a main-memory optimized database (e.g., 5-10%), Siberia leads to an acceptable throughput loss of 7-14% when the cold store resides on a commodity flash SSD.

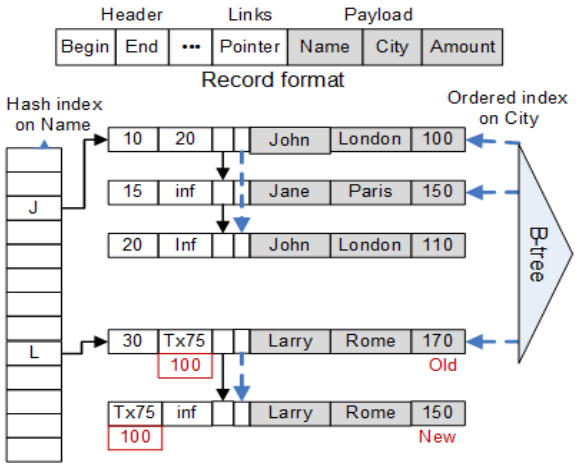

Figure 1: Hekaton record structure and indexing.

The rest of this paper is organized as follows. Section 2 and 3 describe Hekaton's storage and indexing, as well as its concurrency control technique. Section 4 and 5 covers the Siberia architecture and describe how it integrates into Hekaton. Section 6 provides an experimental evaluation of Siberia, while Section 7 covers related work. Section 8 concludes this paper.

2. HEKATON STORAGE AND INDEXING

In this section we summarize how records are stored, accessed and updated by the Hekaton storage engine. This background information is needed to understand our design for cold data.

A table created with 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 [16] and range indexes which are implemented using Bw-trees, a novel lock-free version of B-trees [14]. A table can have multiple indexes and records are always accessed via an index lookup. Hekaton uses multiversioning where an update always creates a new version.

Figure 1 shows a simple bank account table containing five version records. Ignore the numbers (100) and text in red for now. The 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 an ordered 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 three records: two 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).

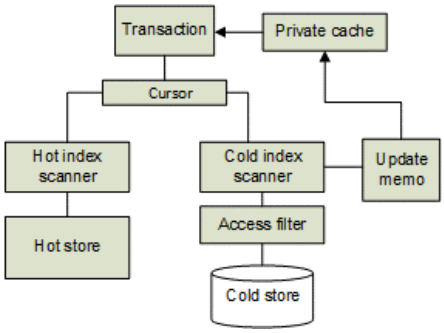

Figure 2: Main architectural components of Siberia.

2.1 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.

2.2 Updates

Bucket L contains two records that belong to Larry. Transaction 75 is in the process of deducting $20 from Larry's account. It has created the new versions (Larry, Rome, 150) and inserted it into the index.

Note that transaction 75 has stored its transaction Id in the Begin and End fields of the new and old version, 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. 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. After committing, transaction 75 returns to the old and new versions and sets the Begin and End fields to 100. The final values are shown in red below the old and new versions. The old version (Larry, Rome, 170) now has the valid time 30 to 100 and the new version (Larry, Rome, 150) has a valid time from 100 to infinity.

This example also illustrates how deletes and inserts are handled because an update is equivalent to a deleting an existing 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 [4].

3. CONCURRENCY CONTROL

Hekaton utilizes optimistic multi-version 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 reference [11].

Hekaton uses timestamps produced by a monotonically increasing counter for several purposes.

- **Commit/End Time** for a transaction: every update transaction 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 transaction serialization order.
- **Valid Time** for a version: Every version in the database contain two timestamps – begin and end. The valid time for a version is the timestamp range defined by its begin and end timestamps.
- **Logical Read Time**: the read time of a transaction is set to the transaction's start time. Only versions whose valid time overlaps the logical read time are visible to the transaction.

The notion of version visibility is fundamental to 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.

3.1 Transaction Commit Processing

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

3.1.1 Validation

Validation is required only for update transactions running at repeatable read or serializable isolation but not for read-only transactions or update transactions at lower isolation levels. Validation begins with the transaction obtaining an end timestamp. It then verifies its reads and, if it runs under serializable isolation, it also verifies that no phantoms have appeared.

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 it is only required for update transactions at higher isolation levels and, furthermore, most likely the versions visited during validation remain in the L1 or L2 cache.

3.1.2 Logging and Post-processing

A transaction T is committed as soon as its updates 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.

Once T's updates have been successfully logged, T is irreversibly committed. T then begins a postprocessing 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.

4. SIBERIA ARCHITECTURE

In our proposed architecture, shown in Figure 2, some records are stored in a hot store and some records in a cold store. The hot store contains records that are accessed frequently while the cold store contains archival data that is rarely accessed but still needs to be part of the database.

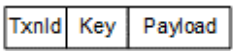

Figure 3: Structure a record in the cold store.

Our goal is to provide a unified interface for processing records coming from hot or cold stores and hide the physical location of a record to higher software layers. We achieve this by implementing all the Siberia machinery below the cursor interface of the Hekaton engine. At the same time, we need an efficient way to access the cold store that minimizes the overhead caused by accessing secondary storage. The intent is to move data automatically and transparently between hot and cold stores according to their access pattern. This migration of data between stores should be done seamlessly without affecting the active transactions and while the DBMS is online and running.

To allow seamless migration between hot and cold stores, we perform migration using normal operations (insert and delete) wrapped in a transaction. This allows the migration transaction to run while other transactions are working and still keeping transactions isolated and the database consistent.

4.1 Cold Store

The cold store is a persistent store where we can insert, read, and delete records. The store does *not* have to be transactional; all we require is durability. Optionally, the cold store may provide indexes to speed up queries. With these minimal requirements, different implementations of the cold store are possible. It could, for example, be implemented as a traditional DBMS table because it is durable, efficient and supports different types of indexes.

As shown in Figure 3, records in the cold store contain the same key and payload fields as their hot counterparts plus a field TxnId that stores the Id of the (migration) transaction that inserted the record into the cold store. TxnId serves as a version number that helps uniquely identify different versions of the same record.

4.2 Access Filters

The data in the cold store may have multiple indexes, typically the same indexes as the corresponding in-memory table. Each cold store index is associated with an in-memory access filter. The access filter stores a compact (and accurate) summary of the contents of cold store. Before accessing the cold store we check the access filter; it the filter indicates that no records in the cold store satisfy the search condition we have avoided an unnecessary trip to the cold store. In previous work we proposed two novel access filters: one for point lookups [17] and another that supports range queries [1] (details are available in these references).

Our prototype only supports point lookups via hashing and uses a new form of Bloom filters [17]. Instead of one large Bloom filter per hash index, the filter is split into many smaller Bloom filters, each one covering the records in a few adjacent hash buckets. The filters are stored in an in-memory Hekaton table. This design has several advantages over using a single large Bloom filter: (a) building them is faster; (b) filter lookups are faster (fewer cache misses); (c) access skew can be exploited by assigning fewer bits to seldom used filters and more to heavily accessed ones; and (d) filters that have deteriorated because of inserts and deletes can be quickly rebuilt by scanning a few hash buckets.

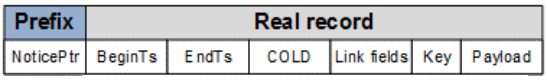

Figure 4: Structure of a cached record.

4.3 Private Cache

Each transaction has a private cache that stores the records it has read from the cold store. We use private caches because each record in the cold store is accessed infrequently so the probability of another transaction reading the same record within a short time window is very low. This simplifies the cache implementation because there is no need to consider concurrency issues. If a cold record is read twice by two different transactions, we will actually keep two cached versions, one for each transaction. There is a chance of using extra memory here but we view this as an acceptable price to pay.

Figure 4 shows the structure of a record in the cache. The part labelled "Real record" has exactly the same structure as a record in the hot store. The field NoticePtr in the prefix contains a pointer to a notice in the update memo (described in the next section) or null. This pointer is only used during validation.

When a cursor receives a record from a cold scanner, it checks whether the record is visible and satisfies the cursor's filter predicate (if any). If it finds that the record qualifies, it copies the record into its private cache, marks it as COLD, clears link fields, and sets the NoticePtr. It then returns a pointer to the Real record part of the copy. Each transaction has a separate memory heap and the cached copies are stored in this heap. When a transaction terminates, all memory reserved by its heap is released.

To software layers above the cursor interface, records in a private cache look exactly the same as records in the hot store. However, during validation we need to be able to identify which records originate from cold storage. We mark a record as cold by setting its first link field to a special value "COLD".

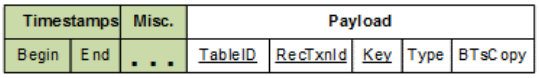

Figure 5: Structure of a timestamp notice in the update memo.

4.4 Update Memo

The update memo is an in-memory table that temporarily stores *timestamp notices* that specify the current status of records in the cold store. Its primary purpose is to enable validation and detection of write-write conflicts to be done entirely in memory without accessing the cold store. The update memo is implemented as a durable Hekaton table. There is one update memo per database. A notice is structured as shown in Figure 5.

As all records in Hekaton tables, a notice contains a begin timestamp (Begin) and an end timestamp (End). The record header contains miscellaneous other fields used internally by Hekaton. The notice's target record is uniquely identified by the three underlined fields: TableID, RecTxnId, and Key. TableID contains the ID of the table to which the record belongs. RecTxnId contains the value from the field TxnId of the target record. The field Key is a variable length field storing the record's unique key. The cold store may contain multiple versions of the same record but the versions have different RecTxnId values. The field BTsCopy is used for storing a copy of a timestamp – it is seldom used but still needed.

A timestamp notice carries timestamps for its target records which are used when the record is read into a private cache. The Type field specifies which timestamps the notice carries as follows.

- N (None) – temporary notice with no timestamps. The target record is orphaned because of an in-progress or failed migration transaction and is to be ignored.
- B (Begin) – the begin timestamp of the notice equals the begin timestamp of the record
- E (End) – the begin timestamp of the notice equals the end timestamp of the record
- BE (Begin & End) – the BTsCopy field contains the begin timestamp of the record and the begin timestamp of the notice equals the end timestamp of the record

The update memo is used when reading a record from the cold store and during validation. When reading a record, we check whether there is matching notice in the update memo. If so, the notice specifies the record's current status and its timestamps.

An update transaction running under repeatable read or serializable isolation must validate its reads by rechecking the timestamps of every record it read. The update memo stores the current timestamps of all records that have been updated since the begin time of the oldest active transaction. Consequently, a transaction can validate its reads simply by checking notices in the update memo.

A serializable update transaction must also validate its scans. We completely avoid repeating scans against the cold store by always inserting new versions in the hot store. This is explained in more detail in the next section.

5. SIBERIA INTEGRATION IN HEKATON

In this section, we describe how each basic operation is done in the presence of the cold store. First, we describe how we insert a record into the cold store during migration. Then, we discuss how we delete and update records in the cold store. After that, we explain how to read a record from cold storage. Finally, we show how a transaction can be validated when it reads cold records.

5.1 Insert

New records are always inserted into the hot store. When a record is updated the new version is always inserted into the hot store regardless of where the old version resides.

5.2 Migration to Cold Storage

Siberia classifies candidate records for migration *to* and *from* cold storage using a technique we proposed in previous work [13]. The basic idea is to log a sample of records to reduce overhead on Hekaton's main execution path. Our classification algorithm is capable of classifying hot and cold records from a log of 1B accesses in sub-second time (and can be pushed to a different machine or given its own core, if necessary). Siberia triggers migration to and from cold storage using the results of classification. We now describe how data migrates to cold storage. We use the update operation to bring records from cold to hot storage (cold record updates are discussed in Section 5.4).

All inserts into the cold store are done by a special migration transactions that move records from the hot store to the cold store. Migration is preferably done in the background when the system is lightly loaded. We discuss how to migrate a single record but the same techniques apply when multiple records are migrated at once.

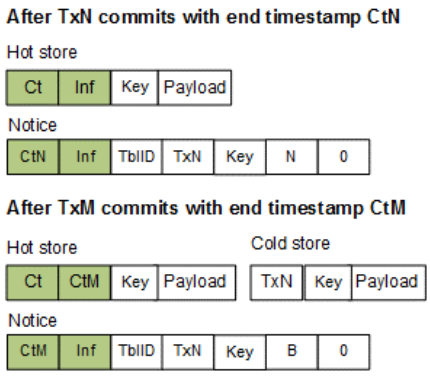

Figure 6: Contents of cold store, hot store, and update memo during migration of a record.

A record is migrated to the cold store by deleting it from the hot store and re-inserting a record with the same content in the cold store. During migration, the system is still active and other transactions may be running, some of which may wish to read or update the record being migrated.

Because the cold store is not transactional, migration must be done carefully in two steps, each in a separate transaction. If it were done in a single transaction and the transaction aborted we could end up with two valid instances of the same record, one in the hot store and one in the cold store.

The first transaction, TxN, reads a target record and installs a preliminary migration notice Ntc of type N in the update memo. The notice essentially says "If you read a record that matches this notice, ignore it – it's not supposed to be there." A record is not migrated unless it is a latest version (end timestamp equals infinity) and it is visible to *all* transactions, current and future. Hekaton's versioning is transient meaning that an old version is kept only until it is no longer visible to any currently active transactions. Old versions are cleaned out quickly so there is no point in migrating them to the cold store.

The second transaction. TxM, performs the actual migration. It creates a copy of the target record, inserts it into the cold store, and updates any affected Bloom filters. Once the record has safely been inserted into the cold store, the migration notice is updated, setting its type to B. Finally the record in the hot store is deleted and the transaction commits. The notice stores the begin timestamp of the version in the hot store. The notice is updated in the same transaction as the version in the hot store is deleted which ensures that the two versions will have non-overlapping timestamp ranges. Hence, only one of the two version will be visible to a transaction, never both.

If transaction TxM doesn't commit because of a crash or any other reason, all its changes except the insert into the cold store will be rolled back automatically by Hekaton. In that case, the new version will still exist in the cold store but readers will find a notice of type N and ignore it.

The effect of migrating a record to the cold store is illustrated in Figure 6. It shows the status after commit of transactions TxN and TxM. Transction TxN just inserts a notice into the update memo. Transaction TxM deletes the version in the hot store by setting its end timestamp. It inserts a copy of the record into the cold store and updates the notice in the update memo.

The old version in the hot store and the old version of the notice (of type N) will be garbage collected once they are no longer visible to any active transactions. The new version of the notice can be deleted as soon as the version in the cold store is visible to all active transactions, that is, when the read time of the oldest active transaction in the system becomes higher than CtM. Cleaning out notices that are no longer needed is done in the background by special memo cleanup transactions (described in Section 5.6).

It is possible that a transaction T might attempt to read or update a record R in the hot store that is under active migration. In this case, R will have an end timestamp of TxM, meaning TxM is in the process of moving R to cold storage and changing R's update notice type. In this case T follows Hekaton's visibility rules. If T is a reader it ignores R if TxM is active or aborted. If TxM is preparing to commit, then R is visible to T if TxM's commit time is greater than T's read time, otherwise T *speculatively ignores* R[1]. If TxM is committed then T uses TxM's end timestamp to test visibility. T is allowed to update R only if TxM has aborted; if TxM is active or preparing then this is *write-write conflict* and T must abort. The details of Hekaton's visibility rules are covered in [11].

5.3 Delete

Deleting a record from the cold store is also a two-step process. We first mark the record logically deleted by creating a notice of type E or BE in the update memo. The notice specifies the end timestamp of the record. Readers always check the update memo so they will find the notice and check the record's visibility. If the record is not visible, the reader ignores the record. The record is physically removed later by a memo cleaner transaction. The record can be removed and the notice deleted only when it is no longer visible to any of the active transactions.

A record in the cold store may be deleted because it is being updated. If that is the case, the new version is inserted into the hot store in the same transaction.

Figure 7 shows the effect on the update memo of a deletion. The net effect is simply to add or update a notice in the update memo. The begin timestamp of the notice specifies the end timestamp of the target record. The notice is essentially saying "As of my begin timestamp the record is no longer visible to anybody so ignore it".

We also need to consider whether write-write conflicts will be correctly detected. Suppose we have two transactions that attempt to delete or update the same record concurrently. A write-write conflict in the hot store is easily detected because each record has exactly one copy in memory. One of the transactions will be the first one to update the record's end timestamp. When the other transaction attempts to change the timestamp it will notice that it is no longer equal to infinity, conclude that another transaction has or is about to delete the version, and abort.

With cold records, each transaction reads its own copy from disk and obtains a cached version in its own buffer. An update or delete is done by inserting or updating a notice in the update memo. If there is an existing notice that needs to be modified, the conflict is detected when the transactions attempts to modify the end timestamp of the notice. However, there may be no old notice to modify. The update memo has a unique index built on the fields TableID, RecTxnId, and Key. Two transactions trying to modify the same record version will end up trying to insert two notices with the same key value. The unique index on the update memo will detect the violation and one of the transactions will abort.

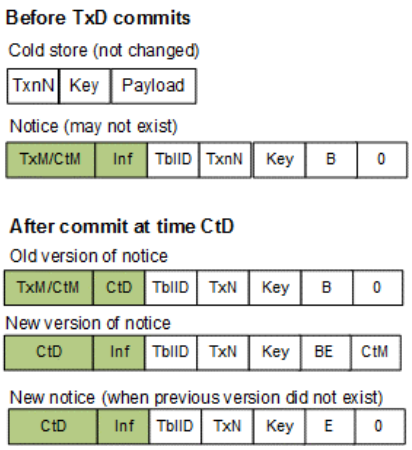

Figure 7: Effect on the cold store and update memo of a record deletion.

5.4 Updates

A record in the cold store is updated by deleting the old version from the cold store and inserting the new version into the hot store. The new version may, of course, be later migrated into the cold store but new versions are never directly inserted into the cold store. Regular insertions of completely new records also go into the hot store (as described in Section 5.1).

There are two reasons for always inserting new versions into the hot store. First, a new record is considered hot and updating a record is interpreted as a signal that it is (potentially) becoming hot. Second, it has the effect that the scan validation required by serializable transactions can be done entirely in the hot store, that is, entirely in memory. Scan validation checks for new records satisfying the scan predicate that have appeared since the transaction began. Since all modified and new records are always inserted into the hot store, it is unnecessary to check the cold store.

5.5 Read

A point lookup or scan first probes the in-memory access filter to see if it must access cold storage. If a cold access is necessary, it begins by retrieving the record from the cold store into an IO buffer and padding it so it has the same format as in-memory records. The next step is to look for a matching notice in the update memo. If notice of type N is found the record is ignored. Otherwise its begin timestamp and end timestamp are set and we check whether it is visible and passes any user-defined filter functions. If it passes all the tests, it is copied into the transaction's cache and a pointer to the cached record is returned to the transaction.

5.6 Update Memo and Cold Store Cleaning

Migrating records to the cold store and updating or deleting records in the cold store adds notices to the update memo. Deletes from the cold store do not physically remove records. To prevent the update memo from growing indefinitely, we need to delete stale notices. Similarly, we also need to remove records in the cold store that are no longer visible to any transactions.

[1] Speculatively ignoring R means T takes a commit dependency on TxM. Hekaton commit dependency handling is covered in [11].

This required maintenance of the update memo is done by a cleaner process that is invoked periodically. While the cleaner process is running migration is blocked. The cleaner process scans through the memo once, checking each committed notice. The action taken depends on the type of the notice.

- An N type notice indicates a failed insert into the cold store. The associated record is not visible to anybody so it can be safely removed and the notice deleted.
- A notice of type BE or E corresponds to a (logically) deleted records. If the record was deleted before the begin timestamp of the oldest active transaction, it is not visible to any current or future transactions and can be safely removed and the notice deleted.
- A notice of type B corresponds to a currently visible record (its end timestamp is infinity). If its begin timestamp is earlier than the begin timestamp of the oldest active transaction, the record is visible to all current and future transaction. If so the notice can be safely deleted because a record with no notice will get default timestamps (1, infinity) when being read.

The actual delete of a notice has to be done in a transaction that begins after the corresponding cold record, if any, has been deleted. This is to prevent a situation where a notice was deleted but the corresponding record was not.

5.7 Validation

An update transaction running at repeatable read or serializable isolation needs to validate before it commits. If validation fails, the transaction aborts.

For both repeatable read and serializable isolation, we must verify that records read by the transaction have not been updated or deleted by another transaction before commit. In a memory-only scenario, this is done by keeping a pointer to each record read. At commit time, we test to make sure that all the read versions are still visible as of the commit time. If all versions are still visible, it means that no other transaction changed them during the lifetime of the validating transaction.

With cold records, we do not have a single copy of the record. However, we know that all updates to cold records are done through the update memo. The test is similar to that in memory-only scenario but we need to fix the end timestamp of the records in the transaction's cache before performing the visibility check.

To update the end timestamp of a cached record Rc, recall that Rc is prefixed with a field NoticePtr that is either null or points to the notice used when the record was read. If NoticePtr is not null, we copy the notice's begin timestamp if it is of type E. Otherwise, the notice is of type BE and we we copy from its BTsCopy field.

For a serializable update transaction, we must also validate the transactions scans to detect phantoms. In a memory-only scenario, we accomplish this by repeating the scan against the main-memory index and checking whether any new records have appeared. Repeating a scan in the cold store could be very expensive, so we want to avoid doing so. By design, we insert newly inserted records and new versions resulting from an update of a cold record directly into the hot table. This means that a scan of the hot table is enough to detect phantoms.

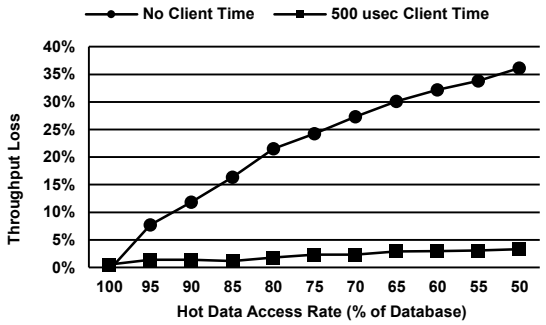

Figure 8: In-memory overhead of the Siberia framework.

However, a serializable transaction may still fail in the following scenario:

1. TxS (Serializable) scans the table. TxS is still active.
2. TxI (Insert) inserts a new record to the hot store and commits.
3. TxM (Migrate) migrates the newly inserted record to the cold store and commits.
4. TxS validates by repeating the scan over the hot table. The newly inserted record will not be visible anymore because the version in the hot store has been migrated.

To solve this problem we enforce an additional constraint on migrating records. When the migration transaction starts, it first computes TsBoundSer, the begin timestamp of the oldest serializable transaction that is still active (uncommitted) in the system. The migration transaction does not migrate any record with a begin timestamp later than TsBoundSer. This ensures that a newly inserted record in the hot store will remain there until the serializable transaction validates. This additional constraint is not likely to delay migration of very many records so its effect is minimal. However, it is necessary for correctness.

5.8 Discussion

Any cold-data management framework should limit accesses to slower cold storage to an absolute minimum. This section briefly discusses how the Siberia achieves this goal within Hekaton. A *read* of a cold record[2] requires a single cold store read to bring the record into memory (the transaction's private cache); after that all processing is done in memory. An *update* or *delete* of a cold record requires (a) a single cold store read to bring the existing record into memory (also to verify its existence) and (b) a single cold store delete to remove the record from secondary storage (in case of an update the new version is placed in memory). The *migration* of a record from *hot to cold storage* requires a single insert into the cold store. Insert operations place a new record in memory, thus do not incur an access to cold storage. Since the update memo and private cache ensure all validation and postprocessing occur in memory, no extra cold storages accesses are necessary.

[2] We assume the read is necessary and that the access filters are accurate.

6. Experiments

To evaluate the Siberia framework, we prototyped it within the Microsoft SQL Server Hekaton engine. This section reports experimental evaluation of the cold data access techniques described in this paper.

6.1 Setup

6.1.1 Machine Configuration

Our experiment machine is an Intel Xeon W3550 (at 3.07 GHz) with 24 GB of RAM and 8MB L3 cache. The machine contains four cores that we hyper-thread to eight logical processors. All cold data is stored on a Samsung 840 SSD with 500GB capacity. Tests using the SQLIO utility [22] revealed that this drive is capable of sustaining 140K IOPs for 512 byte random reads (at queue depth 32). All I/O in our experiments is un-buffered.

6.1.2 Workload

We use two workloads in our experiments:

YCSB Benchmark. The YCSB cloud benchmark [23] consists of a 20GB single-table database. Each record is 1KB consisting of ten 100 byte columns. YCSB consists of single-step transactions that either read or upate a record. We run three workload types from the benchmark: (1) *Read-heavy*: 90% reads and 10% updates; (2) *Write-heavy*: 50% reads and 50% updates; (3) *Read-only*: 100% reads. YCSB uses a scrambled Zipfian distribution to generate key accesses for a workload. We vary distribution skew between 0.5 (lower skew) and 1.5 (higher skew).

Multi-step read/update workload. This workload consists of a single Hekaton table containing 20M records. Each record has a size of 56 bytes (including header), meaning the database occupies roughly 1.04 GB of memory; this is safely outside of the L3 cache of our experiment machine. We employ two types of transactions in our workload. *(1) Read-only*: a transaction consisting of four reads of four distinct records**.** *(2) Update-only.* a transaction consists of four updates of four distinct records While simple, this single-table workload allows us to closely control the experiment parameters to understand Siberia's performance.

Unless otherwise noted, 70% of the database resides in the cold store. The workload is generated by 32 clients. Each client maps to a single thread and calls directly into the Hekaton engine. To introduce realistic client processing and communication delay, each client waits 500 microseconds before issuing its next transaction. We chose 500 microseconds since it roughly estimates the round-trip time for an inter-datacenter network message [2].

6.1.3 Cold Stores

Our experiments use two cold store implementations.

- **Memory-only**: The cold store is in-memory Hekaton table. We use this store to provide pinpoint measurements of in-memory overhead of the Siberia machinery.
- **Direct-mapped file**: The cold store maps a record id to a direct offset in a file on flash. All I/O against the file is unbuffered. Unless explicitly mentioned, the direct-mapped file is the default cold store for all experiments.

As mentioned previously in Section 4.1.1, an important aspect of Siberia is its flexibility in interfacing with various cold storage implementations (e.g., a traditional DBMS table, a file on secondary storage, or a key-value store [14]). However, we experiment with two simple cold stores to avoid observing overhead of a second system stack. As noted previously in Section 1, it is not our focus to study the optimal medium and architecture for cold storage; we are concerned with the overhead of integrating the cold data migration and access machinery within Hekaton. Our Bloom access filters are allotted 0.1% of the database memory budget. Since our filters are adaptive to skew (see Section 4.2), this budget provides sufficient accuracy.

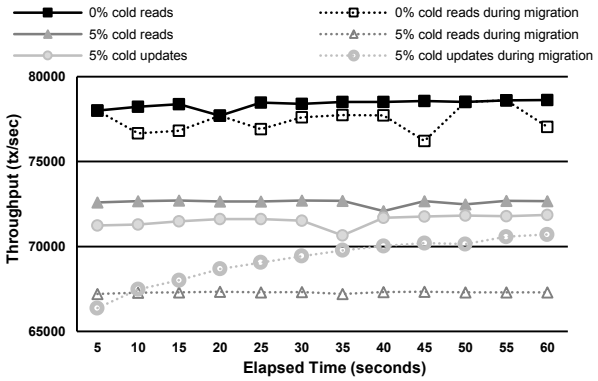

Figure 9: Migration overhead.

6.2 In-Memory Performance

This experiment measures the pure in-memory overhead of the Siberia machinery by running the read-only workload on (1) Hekaton *without* the Siberia implementation and (2) Hekaton with Siberia using the memory-only cold store. We run the workload two ways: (a) with no client time delay where the workload runs as a stored procedure compiled against the Hekaton engine; this measurement represents the performance of *only* the core Hekaton engine and (b) with a 500 microsecond client delay; representing a more realistic client/server transaction processing scenario, e.g., when Hekaton runs inside SQL Server connected to a client on a separate node.

Figure 8 reports the throughput loss when running Hekaton with Siberia for decreasing hot data access rates (for Hekaton without Siberia, the workload accesses random keys in the table, while Hekaton with Siberia accesses hot/cold keys at the specified rate). As expected, for the *no client delay* case the overhead rises as the cold access rate increases. Each cold store probe consists of two extra in-memory table probes: one to check the Bloom filter and another to access cold storage. However, the performance loss ratio is less than one-to-one in the amount of extra probes need to access cold records (e.g., a 5% cold access rate leads to 7% throughput loss, while 50% cold accesses leads to a 37% loss). This is likely due to CPU caching effects, especially for the Bloom table that stays relatively small. The story is different for the 500 usec client delay case. With realistic client delay the in-memory overhead of the Siberia framework accounts for roughly 1% loss in throughput for realistic cold data access rates of up to 15%. For extreme cold access rates the throughput loss is only 3%, thus the in-memory Siberia machinery accounts for a very low performance loss overall.

6.3 Migration

This experiment studies the overhead of running live migration while transactions are actively running within Hekaton. We continuously run transactions on our database for 60 seconds while a migration worker migrates 10% of the database to cold storage

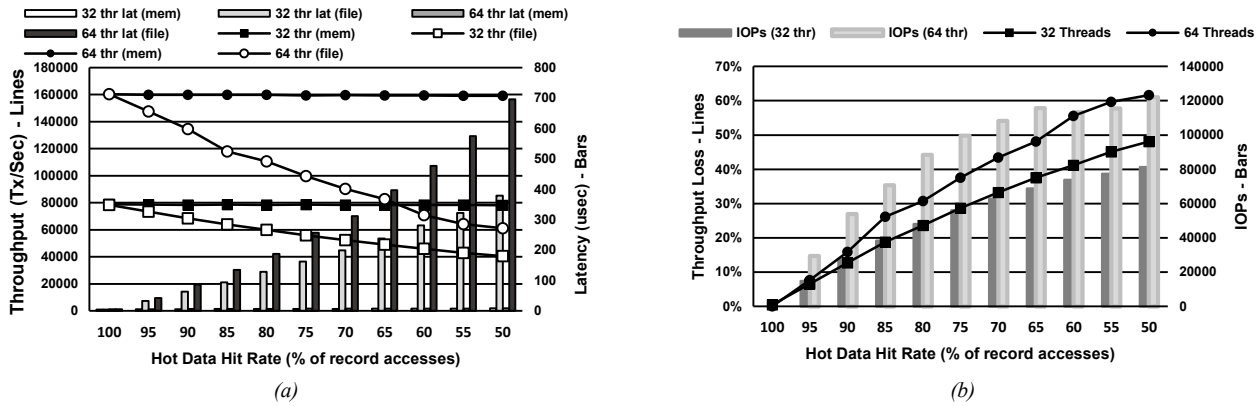

Figure 11 : Read-only workload results for decreasing hot data hit rates. Figure (a) plots throughput (left axis) as lines and latency (right axis) as bars.

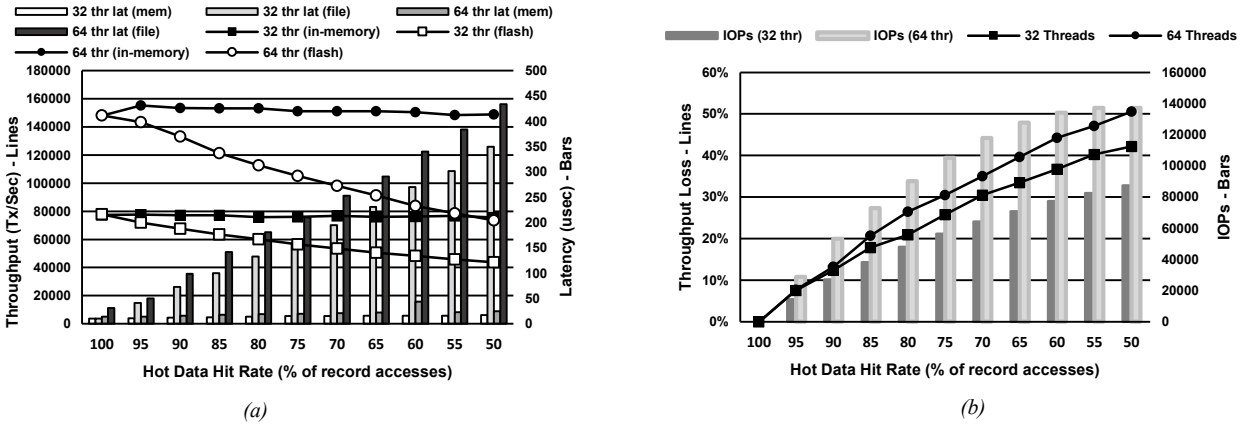

Figure 12 : Update-only workload results for decreasing hot data hit rates. Figure (a) plots throughput (left axis) as lines and latency (right axis) as bars.

(each migration transaction contains a batch of 100 records); this is sufficient to keep migration running for the whole experiment. We perform three experiments. (a) Using the read-only workload that accesses 0% cold records (black line with square markers); this run does not touch records in the cold store nor "in transit" migrating records in order to measure the overhead of the migration thread without interference from cold accesses. (b) Using the read-only workload that reads 5% cold keys (grey line with triangle markers). (c) Using the update-only workload that touches 5% cold keys (light grey line with circle markers). For experiment (b) and (c), the 5% includes cold records, recently migrated records (due to the migrating thread), and in-transit migrating records. For each experiment, we compare against the same workload running without active migration.

Figure 9 reports the numbers for this experiment over the 60 second experiment period in 5 second increments (x-axis). Overall, performance of the system remains stable when live migration is active; this is important as users will not experience sporadic performance degradation. Experiment (a) reveals that the overhead of active migration is at most 2% when transactions do not access cold data; this overhead is due to both the migration thread stealing cycles as well as creating garbage (old record version) in the hot store. Experiment (b) reveals that read transactions experience a 7% overhead when accessing cold data while migration is active. On top of the 2% ambient overhead from the migration thread, these transactions also require an additional probe of the update memo on the cold data access path (since the migration creates fresh memo notices) as well as copying timestamps from the update memo if it accesses a cold record that was freshly migrated. Experiment (c) reveals that update transactions initially incur an 8% overhead to both read then update the cold record. However as time moves forward the overhead slowly diminishes since more records are found in the hot store due to updates automatically writing new versions to the hot store.

6.4 Effect of the Update Memo

This experiment studies the overhead of the update memo on the path to accessing a cold record. The experiment consists of all read-only transactions, where each read touches a cold record in the *memory-only* cold store (*there are no* hot accesses). We run each workload in three different configurations: (a) with no entries in the update memo, meaning each read avoids probing the update memo altogether; (b) the update memo is populated, but the read does not find a matching entry in the memo; (c) the update memo is populated and the read finds a matching entry in the memo.

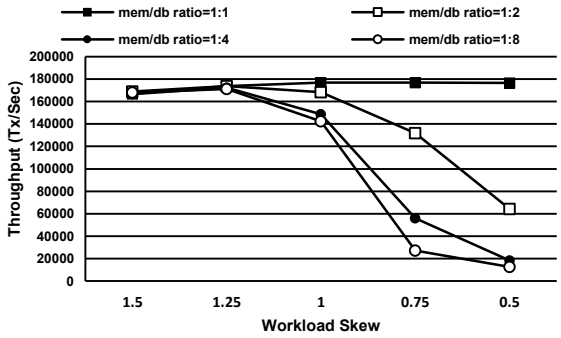

Figure 13: YCSB write-heavy workload.

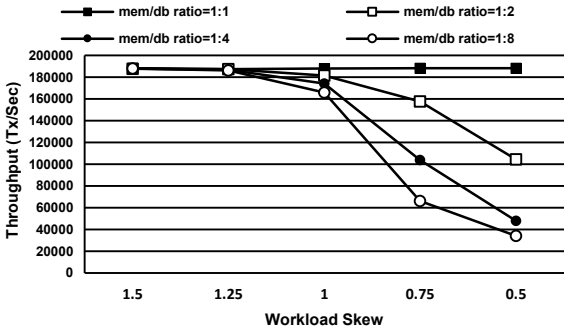

Figure 14: YCSB read-heavy workload.

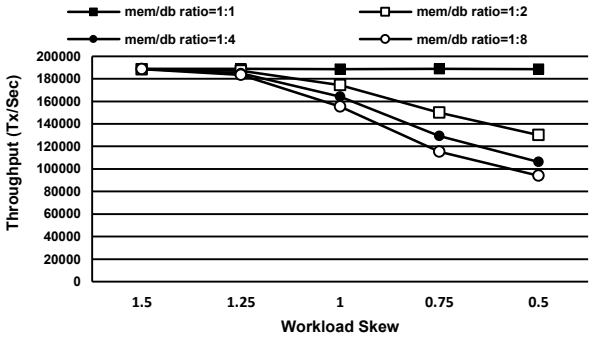

Figure 15: YCSB read-only workload.

	Empty Memo	Probe (no hit)	Probe (hit)
Latency	11.14 usec	13.89 usec	15.14 usec

Table 1: Effect of the update memo.

Table 1 reports the average transaction latency (in microseconds) for reading a cold record for each configuration. Without a memo probe, a transaction accessing only cold records takes approximately 11.14 microseconds (including transaction setup and teardown time). Accessing the memo without a match takes 13.89 microseconds (a 24% increase) while accessing the probe with a match takes 15.14 microseconds (a 35% increase). Clearly, the overhead of accessing the memo is expensive, since this involves allocating a memo stub for a probe and performing the table probe for the update memo. This means memo cleaning is necessary for good performance.

6.5 Synthetic End-to-End Workload

This experiment tests end to end workloads of Siberia for varying hot data hit rates (each transaction selects a hot/cold record id based on the access rate). We run the workloads on both cold store configuration (denoted "in-memory" and "file") to test the relative performance loss when I/O is on the critical path. Since the direct map file I/O is sequential, we run the workloads using both 64 and 32 worker threads in order to experiment with different traffic patterns to the I/O subsystem. For all runs, migration is inactive to ensure the workload is achieving its exact hot/cold hit rate. The update cleaner process is active for the duration of the workload.

6.5.1 Read-Only Transactions

Figure 11(a) reports the throughput and latency results for decreasing hot data hit rates (increasing cold data hit rates), while Figure 11(b) reports the IOPs for the direct-mapped file cold store configuration along with the relative performance loss compared against the in-memory cold store (representing an "optimal" cold storage implementation).

In terms of latency, using 64 threads a 5% cold access rate leads to a latency increase of 36 usec (for 32 threads it is 28 usec) – a minor fraction of a transaction's end-to-end processing time (including network and client processing time). For 32 threads the throughput loss is linear to the amount of cold records accessed as we observe that 50% cold accesses lead to 50% less throughput. This is because the system is not I/O bound (IOPs are below the flash drive's capability). At 64 threads, the workload is IO bound at a cold access rate of 35%, thus throughput loss is more than linear. We also observe in Figure 11(b) that as cold access rates increase, the throughput loss is greater for 64 threads than for 32 threads. This is likely due to the fact that as the number of workers increase, I/O queue length increases thereby increasing the time a transaction waits on I/Os. Longer waits naturally lead to increased transaction latency, as we observe in Figure 11(a), which in turn affects throughput since fewer transactions complete per second.

This experiment clearly shows that for high cold data access rates, having I/O on the critical path of a main-memory database adversely affects performance. However, the main takeaway from this experiment is that for realistic cold data access rates, the performance loss is acceptable: for 5% cold data access rates, the throughput loss between the direct-mapped file and in-memory cold store is 7%; for a 10% cold access rate, the throughput loss is roughly 14%.

6.5.2 Update-Only Transactions

Figure 12 reports the experimental results for the update-only workload. The general trends are similar to that of the read-only workload. Throughput is slightly lower overall since each update requires a read prior to installing a new record version. Since updates of cold records go to the hot store, a cold read accounts for a large portion of the update cost. In addition, garbage collection is necessary to both clean the update memo and collect old in-memory versions.

Similar to the read-only workload, for realistic cold data update rates the performance degradation is acceptable at 8% throughput loss for 5% cold data updates and 13% throughput loss for 10% cold data update rates.

6.6 YCSB Workload

Figure 13 reports the results of the YCSB write-heavy workload. We report throughput (y-axis) for various workload skew (x-axis)

and different memory to database size ratios (e.g., a ratio of 1:X means the database is X times the memory size). For a 1:1 ratio, all accesses are in memory and performance stays relatively constant. For high workload skew, accesses mostly hit memory-resident records and performance for all ratios stays close to the in-memory case (the 1:1 ratio). As expected, for less skewed access performance starts to degrade as more accesses hit cold storage. When database size is 8x of memory, the system becomes heavily bottlenecked on updates of records on cold storage. Performance for other ratios degrades less severely since more records are memory-resident.

Figure 14 and 15 report the results of the YCSB read-heavy workload and read-only workloads. As expected, performance at the 1:1 ratio is better than the write-heavy workload due to less contention during updates (e.g., write-write conflicts). Performance for these workloads also degrades as access skew decreases and memory to database size ratios increase. For read-heavy workloads, we observe a smaller abort rates for transactions at higher skew rates compared to the write-heavy workloads. This is due to less conflict on the update path and leads to less dramatic performance drop-off at higher access skews. We note that similar trends for the YCSB workload were reported independently by [3], though absolute performance is not comparable due to different hardware configurations.

7. RELATED WORK

Buffer pool. Buffer pool caching is a tried-and-true method from traditional DBMS architectures that maps buffer pool pages to on-disk pages. This technique is not ideal for our scenario. Like most main-memory systems [6, 8], Hekaton does not use page-based indirection for efficiency reasons; there are only records and thus no buffer pool. In addition, buffer pool management incurs an unnecessarily large CPU overhead when the majority of accesses are to in-memory data [7], as is the case in our scenario.

Main memory OLTP engines. There has been much work recently exploring OLTP engine architectures optimized for main-memory access. Research prototypes in this space include H-Store [8, 19], HYRISE [6], and HyPer [9]. Commercial main-memory systems are currently on the market and include IBM's solidDB [15], Oracle's TimesTen [10], Microsoft's Hekaton [4], and VoltDB [20]. Most of these systems such as HyPer, solidDB, TimesTem, and VoltDB assume that the entire database fits in memory, completely dropping the concept of secondary storage. In this work, we diverge from this "memory-only" philosophy by considering a scenario where a main-memory optimized engine may occasionally access data in cold storage.

Cold data management in main-memory systems. A number of researchers have explored management of cold data within main-memory database systems. Our previous paper studied how to identify hot and cold data at record granularity [13]; this paper studies how to migrate and access cold data in a transactionally-consistent manner inside the Hekaton engine.

HyPer [9] is a main-memory hybrid OLTP and OLAP system. HyPer achieves high performance for OLTP workloads by partitioning tables. Partitions are further broken into "chunks" that are stored in "attribute vectors" with each attribute vector stored on a different virtual memory (VM) page. This approach enables the system to take VM snapshots for OLAP functionality. HyPer's cold-data management scheme [5] is capable of identifying cold transactional data at the VM page level, separating it from the hot data, and compressing it in a read-optimized format for OLAP queries. HyPer relies on virtual memory paging for databases with sizes larger than physical memory. In contrast, our work explores migrating and accessing cold data at record granularity.

Stoica et al [18] propose a cold data management approach similar to HyPer that separates hot and cold data into separate memory locations. The goal of this work is to place cold data in a memory location where it is most likely to be paged out by the operating system. Cold data is identified at the granularity of a record, similar to our approach in Siberia [13]. However, cold data migration and access is done at the granularity of a VM page, whereas we consider cold data access at record granularity.

The approach of clustering cold records on separate pages and relying on the operating system to page them in and out of memory is a poor fit for Hekaton. Hekaton indexes chain records together by pointers embedded in records. An index chain could include records both on hot and cold pages. Consequently, even an access of a hot record may have to pass through cold records on its way to the target record, forcing cold pages to be brought into memory. If a table has multiple indexes, clustering records on pages to minimize "just passing through" accesses is a challenging problem.

Calvin [21] is a distributed main-memory database system that uses a deterministic execution strategy. Calvin is capable of accessing data on secondary storage, however, it cannot simply schedule a new transaction while waiting for I/O. Instead, it employs a "warm up" phase before execution that attempts to retrieve (from disk) all records a transaction might access.

Anti-caching [3] is an approach for migrating and accessing cold data in H-Store. The basic idea is to move cold records to external storage leaving only a stub in memory. Indexes remain in memory so all index keys (both hot and cold) must stay in memory. Hot and cold records are identified using LRU but with sampling to reduce the overhead of maintaining LRU chains.

Anti-caching is built for the H-store single-threaded execution model that executes one transaction at a time. A thread executing transaction T1 that encounters a disk-resident record goes into a "pre-pass" phase that speculatively runs the transaction in attempt to find all disk-resident records it might need (without issuing the I/O). The transaction then aborts and the thread goes on to execute other transactions while waiting for asynchronous I/Os to retrieve and install records for T1. T1 is restarted after its data is brought into memory. Anti-caching has two serious drawbacks.

Limited space savings. Indexes and index keys for cold records remain in memory. For tables with multiple indexes, especially if they have multi-column keys, this may severely limit the space savings. Furthermore, the LRU chains also consume valuable space. We opt to completely separate both hot and cold data; only hot records remain in memory, while cold records (keys and payloads) are kept in cold storage.

Repeated execution. If any of the records required by a transaction reside in the cold store, the transaction goes through a pre-pass execution to determine what records to bring in. Once the records have been read in, the transaction is restarted. Unfortunately, *a single pre-pass execution may not be enough***.** Consider a query that joins three tables R, S, and T and the join structure is a chain (R →S → T). This may require up to three pre-pass execution. The first execution causes missing R records to be brought in. In the second we have the join keys from R but some required S records may be missing. In the third execution some T records may be missing. Finally, in the fourth execution all the required records are

in memory and the query finally executes successfully. This wastes a lot of resources and leads to unpredictable performance. Our design does not suffer from this problem: execution of a transaction resumes as soon as the required record has been brought into memory.

8. CONCLUSION

This paper described the Siberia architecture and functionality that enables migrating, reading, and updating data on cold storage in the Hekaton main-memory database engine. Our approach completely separates hot and cold data into an in-memory hot store and persistent cold store (e.g., flash or disk). Siberia does not require a transactional cold store, thus its design is flexible enough to work with a wide array of cold storage implementations. Siberia is space-efficient; besides compact Bloom filters, no information about cold data is kept in memory. Siberia dovetails with Hekaton's optimistic multi-version concurrency control scheme and allows transactions to read and update data in both hot and cold stores without restriction. This functionality is enabled by our update memo design that temporarily stores timestamp information for records recently inserted into or deleted from the cold store. The update memo completely removes extraneous accesses to cold storage by ensuring that Hekaton postprocessing and validation of cold data is done completely in memory. Experiments on Siberia implemented in Hekaton reveal that for cold data acccess rates appropriate for main-memory optimized databases, we incur an acceptable 7-14% throughput loss.

9. REFERENCES

[1] K. Alexiou, D. Kossmann, P.Å. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB 6(14)*: 1714-1725 (2013).

[2] J. Dean. Challenges in Building Large-Scale Information Retrieval Systems. In WSDM 2009 (keynote). Slides available at: `http://research.google.com/people/jeff/WSDM09-keynote.pdf`

[3] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB* 6(14): 1942-1953 (2013).

[4] 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*, 2013.

[5] F. Funke, A. Kemper, and T. Neumann. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. PVLDB 5(11): 1424– 1435 (2012).

[6] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB* 4(2): 105-116 (2010).

[7] S. Harizopoulos, D.J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, 2008.

[8] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB* 1(2): 1496-1499 (2008).

[9] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[10] T. Lahiri, M.A. Neimat, and S. Folkman. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Engineering Bulletin 36(2):* 6-13 (2013).

[11] P.Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, M. Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5(4): 298-309 (2011).

[12] J. Lee, M. Muehle, N., May, F. Faerber, V. Sikka, H. Plattner, J. Krueger, and M. Grund. High-Performance Transaction Processing in SAP HANA. *IEEE Data Engineering Bulletin 36(2):* 28-33 (2013).

[13] J. Levandoski, P.Å. Larson, and R.Stoica. Classifying Hot and Cold Data in a Main Memory OLTP Engine. In *ICDE,* 2013.

[14] J. Levandoski, D. B. Lomet, and S Sengupta, The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE,* 2013.

[15] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, K. Vakkila. IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability. *IEEE Data Engineering Bulletin 36(2):* 14-20 (2013).

[16] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA,* 2002.

[17] L. Sidirourgos and P.Å. Larson, Adjustable and Updatable Bloom Filters. *Available from the authors*.

[18] R. Stoica and A. Ailamaki. Enabling Efficient OS Paging for Main-memory OLTP Databases. In *DaMon*, 2013.

[19] M. Stonebraker et al. The End of an Architectural Era (Its Time for a Complete Rewrite). In *VLDB*, 2007.

[20] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Engineering Bulletin 36(2):* 21-27 (2013).

[21] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*, 2012.

[22] SQLIO Disk Benchmark Tool: `http://aka.ms/Naxvpm`

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.