Multi-Version Range Concurrency Control in Deuteronomy

Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang

Microsoft Research

{justin.levandoski, lomet, sudipta, rystutsm, ruiwang}@microsoft.com

ABSTRACT

The Deuteronomy transactional key value store executes millions of serializable transactions/second by exploiting multi-version timestamp order concurrency control. However, it has not supported range operations, only individual record operations (e.g., create, read, update, delete). In this paper, we enhance our multi-version timestamp order technique to handle range concurrency and prevent phantoms. Importantly, we maintain high performance while respecting the clean separation of duties required by Deuteronomy, where a transaction component performs *purely* logical concurrency control (including range support), while a data component performs data storage and management duties. Like the rest of the Deuteronomy stack, our range technique manages concurrency information in a latch-free manner. With our range enhancement, Deuteronomy can reach scan speeds of nearly 250 million records/s (more than 27 GB/s) on modern hardware, while providing serializable isolation complete with phantom prevention.

INTRODUCTION

1.1 Phantoms

Serializability with range protection is much harder to provide than other levels of isolation (say repeatable reads) because it requires the prevention of phantoms. An example of a phantom is a resource that does not exist when a transaction T begins execution and is not seen during T's execution. However, it is added to the database by transaction S, which serializes before T so that T should have seen it if T is to serialize after S.

To prevent phantoms, lock-based concurrency control locks more than just the records accessed by a transaction. But since phantoms, by definition, do not yet exist, the question becomes: "exactly what do we lock?" T needs a way to lock resources that not only include records that it knows about, but that include all possible records that do not yet exist but that, if they did, would be read by T. The conclusion is that locking individual records is insufficient to provide serializability.

There are a number of locking based solutions to the phantom problem. One way is to lock a resource that includes not only some current set of records but also any records that might be added to this set. Examples of this are table locks, page locks (where pages are associated in some way with a predicate as in a B-tree key range), and next key locking, which locks not only a record but also the range of keys between the record and its successor (see Weikum and Vossen [35] for a more complete discussion).

1.2 Versions and Timestamp Order

Deuteronomy uses a timestamp order multi-version concurrency control (MVCC) method [16]. The big plus for multi-version concurrency is that most read-write conflicts can be avoided. In particular, a transaction can frequently read a committed version that is earlier than a current version produced by an uncommitted transaction. This is an enormous advantage as read-write conflicts are by far the most common form of conflict since transaction read sets are typically much larger than write sets.

We recently showed how to use timestamp order (TO) concurrency control with multiple record versions to provide serializability when reads are restricted to individual records (but not record ranges) [16]. The basic idea is as follows. A transaction X is given a timestamp T when it begins execution, which also serves as its commit time. Transaction X uses T to identify the version that it should read, i.e. the latest version with a timestamp less than T. When X reads the record, it updates the "last read time" for that record (but only if it monotonically increases this "last read time" value). If X tries to read an uncommitted version that has a timestamp earlier than T, then X aborts. If X tries to write a version of a record, but that record has a "last read time" later than T, then X will abort (otherwise it would have written into another transaction's read set). X also aborts if it tries to write to a record that already has a prior active but uncommitted version (a write-write conflict). When X commits, all new versions it created are stamped with timestamp T. These steps ensure that timestamp order acts to serialize transactions.

There have been prior solutions to phantom protection in a multi-version setting, where the approach involves re-reading the range(s) at the end of the transaction so as to validate that other transactions have not changed it [5, 12]. Validation has two characteristics that we find worrisome. (1) Re-reading a range can be quite costly, requiring doubling the number of record accesses for validation. This might be fine for memory-only systems, but Deuteronomy only caches data in main memory, thus validation could involve extra I/O; this is a deal breaker for us. (2) A transaction's reads take place at its start time, but it is serialized as of its commit time. So any range read needs to be unchanged for the execution duration of the transaction, increasing the likelihood of abort as execution time increases.

1.3 Versions and Ranges

The Deuteronomy architecture, depicted in Figure 1, enables a clean separation of duties, where a transaction component (TC) performs *purely* logical concurrency control (but knows nothing about physical data storage), while a data component (DC) performs data storage and management duties but knows nothing about transactions [13, 23]. While Deuteronomy's DC supports key range scans using the high performance Bw-tree [15], until now the TC did not support serializability when key ranges were accessed since it did not protect against phantoms.

Proceedings of the VLDB Endowment, Vol. 8, No. 13
Copyright 2015 VLDB Endowment 2150-8097/15/09.

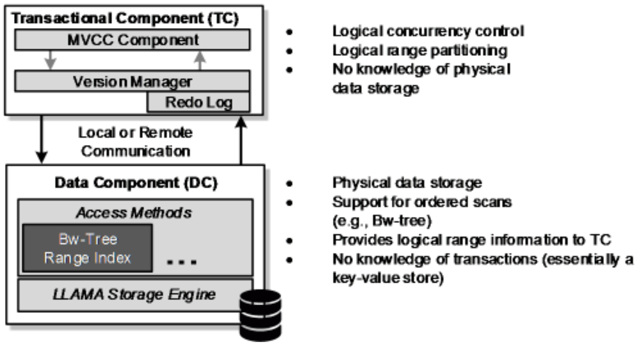

Transactional Component (TC)
MVCC Component
Version Manager
Redo Log
Local or Remote Communication
Data Component (DC)
Access Methods
Bw-Tree Range Index
...
LLAMA Storage Engine
• Logical concurrency control
• Logical range partitioning
• No knowledge of physical data storage
• Physical data storage
• Support for ordered scans (e.g., Bw-tree)
• Provides logical range information to TC
• No knowledge of transactions (essentially a key-value store)

Figure 1: Deuteronomy architecture: The TC performs purely logical concurrency control and receives logical range information from the DC. The DC is in charge of physical data storage and supports key-ordered scans.

Transactional concurrency control in the TC does not know anything about structure modifications done at the DC (e.g., B+-tree page splits). These are fully handled within the DC. If pagination information were exposed to the TC, this might require additional information propagation to keep this information up-to-date, adding both complexity and execution cost. For Deuteronomy we need a concurrency control method that is fully "logical". Deuteronomy's TC should have no knowledge of physical record placement. This, of course, rules out techniques based on pages.

Further, we want to avoid potentially expensive validation (e.g. re-reading a range) when transactions are executed serializably, which is our target isolation level, while also avoiding greatly increased abort rates. A prior paper [20] showed how logical ranges could be used in Deuteronomy when using pessimistic locking at the TC.

The technique in this paper exploits the idea of logical ranges but in a setting where we use timestamp order MVCC. This has never been done before. Hence we want to use versioned ranges, building on our prior timestamp order MVCC for individual record operations as described in Section 1.2.

1.4 Contributions

Achieving high performance requires that conflicting accesses be minimized. MVCC accomplishes this by eliminating most read-write conflicts. We extend our prior timestamp order MVCC scheme to include controlling accesses to ranges, thus providing the phantom protection required for full serializable transactions. The result is a Deuteronomy storage engine (TC plus DC) that exhibits great performance while supporting full serializable transactions. Our contributions in this work are summarized as follows:

1. Defining and managing logical range objects that are compatible with the Deuteronomy TC:DC logical and physical separation (Section 2).
2. Coordinating serializable transactions with write access to records of a range in a multi-version and concurrent way, treating ranges as part of a multi-granularity resource hierarchy, all without requiring a validation step (Section 3).
3. Effectively accessing and managing logical and versioned ranges without materializing any more records of the range than are requested by a client (Section 4).

We performed experiments to measure performance in a number of different dimensions (Section 5). We show that the range-enhanced Deuteronomy TC achieves fully serializable scan throughput of 43 million records/s (more than 4 GB/s): 43% faster than an approach that requires re-reading the range to validate the scan. Related work is covered in Section 6, while a discussion and conclusion is provided in Section 7 and 8, respectively.

2. LOGICAL RANGES

We now provide a brief overview of the Deuteronomy stack in order to understand the data flow within our TC architecture and how it interacts with the DC. We then discuss how to define and manage logical ranges in the Deuteronomy TC; this is important since we need to support purely logical concurrency control.

2.1 The Deuteronomy Stack

Figure 1 depicts the architecture and lists the basic functionality of both the transaction component and data component.

Transaction component (TC). The TC provides logical concurrency control and recovery. It consists of three main elements.

- *MVCC Component*: manages concurrency control within the TC using multi-version timestamp order (as summarized in Section 1.2). It uses a latch-free hash table to manage version information for individual records. This consists of a key, last read time, and the transaction id of each version writer. A *version offset* is also stored with the version, used to access the full record from the version manager.
- *Version manager*: manages the redo log (we use redo-only logging in Deuteronomy). All new writes are immediately written to a log buffer. Buffers are persisted once full, but they are retained in memory to serve as a version read cache until they are eventually recycled and reused. The version manager services log writes as well as version reads (using the version offsets in the MVCC table) that may be retrieved from either an in-memory log buffer, or through the DC, where it is then cached at the version manager.
- *Record manager*: sits atop the MVCC and version manager and is responsible for the user-facing API and running the transaction logic.

A detailed description of the TC and its techniques for achieving high performance are available in a previous paper [16]. This paper describes our method to support range concurrency control within the MVCC component in order to provide high performance serializable range scans without the need for validation.

Data component (DC). The data component manages physical data storage. We assume the DC supports efficient key-ordered scans. This work uses the Bw-tree [15] as our DC. The DC also uses its knowledge of key distribution to help the TC craft a set of logical ranges to optimize scan efficiency and reduce aborts. The rest of this section describes how to define and manage logical ranges.

2.2 Possible Logical Ranges

There are a number of ways to define ranges that do not require physical information. Below we give a brief assessment of some alternatives.

1. **Next key ranges:** Such ranges are very precise and might provide the highest concurrency since the ranges are very small. There are two downsides. (1) A large number of ranges need to be managed. (2) On insertion of a record, we need to check the "next key" to see how and whether the insertion can be done. Deuteronomy's separation of TC and DC means that the next key checking is very expensive. If the TC and DC are located on separate machines, this check involves a round trip over a network.
2. **Table objects:** Tables are usually included in multi-granularity locking implementations. The big difficulty with tables is that a table read access would impact every write access to any record in the table. Such an object would be very

heavily accessed and its elements would be under all but continuous update.

3. **Logical partitions:** A logical partition, e.g., defined by a partitioned key space, can be made small enough to avoid the concurrency bottleneck associated with table objects. If we bring knowledge of the partitioning to the TC early enough (e.g., during TC/DC initialization), we avoid having to access the DC frequently as required by next key ranges. We choose to work with logical partitions for these reasons. However, there are issues associated with logical partitions that we describe below.

2.3 Defining and Identifying Ranges

2.3.1 The Nature of Ranges

The introduction of any resource other than records, and that can denote some collection of records, introduces multi-granularity resource management. Multi-granularity locking has a long history [6], and we would like to leverage some of the thinking behind such a hierarchy. In particular, like multi-granularity locking, a higher level logical granule (similar to a page or table) should be identified by a unique resource identifier to provide high speed hashed access to the objects. To exploit much of the same mechanism that we had already built to handle record versions in the TC MVCC component [16], we designed a similar but separate latch-free hash table for range objects.

A critical aspect of any multi-granularity scheme is that we need to be able to determine, when presented with a record access, to which larger granule (range in our case) it belongs. While not strictly necessary, a nice simplification is to have our range objects be disjoint so that we need to track information for only one range object when a record in the range is accessed. For this reason, we chose to use a disjoint partitioning of the key space.

2.3.2 Range Definition Protocol

As indicated above when discussing next key ranges, it is important to ensure that the TC can acquire range information ahead of access time so that extra trips to the DC are not necessary when processing requests for records within the range. One simple technique might be to simply partition the key space into equal size pieces. This can be done independently of the key distribution.

While simple, we would like a more refined method of establishing range boundaries that at least takes into account key distributions. This avoids some very bad possible cases where all the records might reside in a single partition of the key space. To accomplish this, the TC asks the DC for partitioning information during the initialization phase in Deuteronomy. A hash based DC might respond with an equal "hashed key" space partition.

While the range definition protocol in no way requires a specific partitioning approach from the DC, system performance is enhanced when the DC is responsive to key distribution, and perhaps even key access distribution. As mentioned previously in Section 2.1, this work uses a DC based on the Bw-tree [15]. Using the Bw-tree, our DC provides partitioning information based on key distribution. Its protocol for defining logical key ranges is as follows.

1. The TC requests of the DC a partitioning of the keys that the DC is managing, telling the DC how many disjoint partitions it would like to have.
2. The DC accesses the root of the Bw-tree, and perhaps the level below the root, dividing the entries into the specified number of partitions based on the keys found in the internal index nodes of the B+-tree.
3. The DC responds to the TC, sending it this partitioning. The TC record manager (the user of the MVCC component) uses this partitioning to assign resource ids to the ranges. Subsequent concurrency control requests use these resource ids to identify the range resource that contains any given key.

Note that the MVCC component itself knows nothing about the range resources nor does it know what records should be associated with any of the range resources. This is known only by the record manager. So far as the MVCC manager is concerned, there is a resource hierarchy, but it knows only what is conveyed on any particular request, i.e., record key and/or range id. The record manager translates key ranges to range ids via a search of a table of sorted ranges. The MVCC component is described next.

2.4 MVCC Component

As mentioned previously, the Deuteronomy TC already has an MVCC component to manage concurrency for individual record operations [16]. It uses a latch-free hash table and accesses record versions managed by a version manager via logical offsets. These offsets reference versions either in the redo recovery log, or in a separate log structured read cache that contains records accessed originally from an associated DC. The left hand side of Figure 2 illustrates how this hash table is organized, we refer to this table as $MVCC_{rec}$ throughout the rest of the paper. Note that this data structure is designed to provide access to versions of a single record, and it exploits the fact that record updates are strictly ordered in the version list. Only one version is unambiguously the last version, and only the last version can be uncommitted.

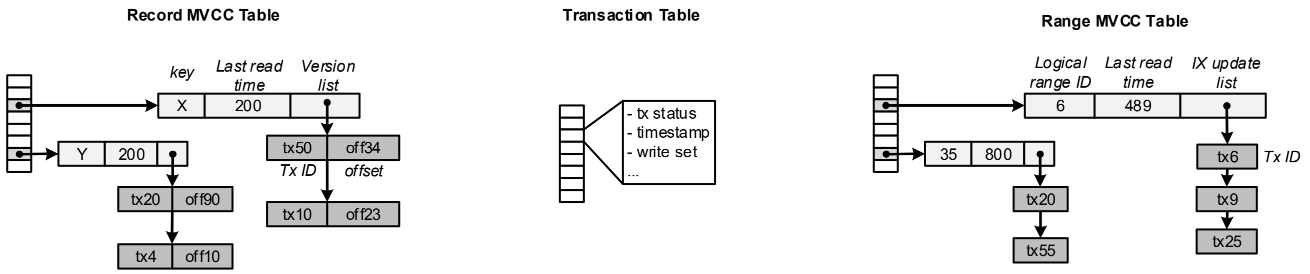
We have designed a separate hash table specifically for our range objects, as depicted on the right hand side of Figure 2, referred to as $MVCC_{ran}$ in the rest of the paper. While there are similarities in the design of this table, we separate the range table from the record table because of the way versions are handled and the fact that changes in ranges can commute, meaning that the range versions can be updated even when uncommitted updates are present. Multi-granularity for ranges permits range updates to be concurrent, with conflicting record updates identified when a corresponding common record is updated.

A range id is purely logical, with no instantiated version referenced by it. In fact, it is possible (and has not escaped our notice) that our strategy for logical ranges can be applied more generally to large objects with incremental updates (e.g., Binary Large Objects, or BLOBs). In any case, ranges are not instantiated in the $MVCC_{ran}$ table. Rather, a transactionally consistent range is assembled incrementally using both scan results from the DC and versions present in the TC (that are stable but have yet to be applied at the DC). We describe this process in Section 4.

3. MULTI-GRANULARITY MVCC

3.1 Multi-granularity Resources

We borrow the idea of multi-granularity resources and their hierarchy from the world of pessimistic locking [6]. Table 1 illustrates the conflict matrix for classic multi-granularity locking. The inner sub-table highlights the various access (lock) modes that we support. We want to bring multi-granularity access management to the world of multi-version concurrency control. Our intent is to preserve the big gain of MVCC in greatly reducing read-write conflicts to drive down the conflict rate sufficiently so that abort can be used to deal with residual conflicts, not blocking. Thus, Table 1 applies to conflicts on a specific version. But given that we support multiple versions, as described below, concurrent access to ranges or records is frequently possible, even when modes conflict.



- Detects single-record concurrency conflict
- Version list for each key is in timestamp order
- At most one uncommitted update in the version list
- Versions contain tx id and offset into version store

- Consulted by both MVCC tables to check tx status and timestamp
- Consulted by range MVCC table to gather updates for a given IX

- Records range IX updates and detects conflicts
- IX list consists of tx id (owner of the IX update)
- Multiple uncommitted IX updates allowed in IX list
- Record-level conflicts detected in record MVCC table

Figure 2: The Deuteronomy TC record MVCC table, range MVCC table, and transaction table. The record MVCC table handles record-level conflicts, while the range MVCC table handles ranges.

Because we want to avoid blocking, we support range reads but not range writes. We reason that exclusive access to a range would, under heavy load, lead to too many aborts. However, a form of "read with intent to update" (denoted in multi-granularity locking with an SIX lock) is supported, meaning active scanners are allowed to update records as they go along. A benefit of not supporting exclusive access to ranges is that we can dispense with IS accesses entirely since they only conflict with the unsupported X mode range lock. Another mechanism is needed should exclusive range access be required.

As with MVCC with record versioning, range MVCC maintains a hash table, with each bucket providing access to the set of ranges mapped to the bucket by the hash function. For each range, we maintain a latch-free list of IX updates (essentially just the id of the transaction performing the update to the range) plus a "last read" time that provides a barrier preventing updates to a range being read with an earlier time, thus preventing phantoms. What is different from record updates is that there can be multiple IX updates, where the IX literally means the same thing as IX in the locking case, i.e. "Intention eXclusive". That is, one or more record updates can be made to the range by the transaction posting the IX.

3.2 Record Updating Protocol

Each single record update will be validated at the record MVCC hash table. Exactly as with multi-granularity locking, a transaction updating a record needs to first post an IX access to its range prior to updating the record. If the transaction previously posted an IX to the range, it does not have to post it again, so this part of the code path can be avoided. Posting the IX update first ensures that a range reader will conservatively see all updating transactions and their timestamps. This avoids a race condition in which a record is being updated with this update not known to a range reader.

After posting an IX access at the range object in the MVCC$_{\text{ran}}$ hash table, the update can be posted to the MVCC$_{\text{rec}}$ hash table entry for the record. It is the posting of the updated version at the MVCC$_{\text{rec}}$ entry that confirms (or not) that the update can be done serializably. This is the basic protocol for all record updates.

We maintain a write set for each transaction by chaining a transaction's MVCC$_{\text{rec}}$ version updates together and retain a pointer to these record versions in the transaction table (see the middle of Figure 2 for the basic information retained in the transaction table). This write set serves as a central location for us to find all updates relevant to a range object by simply using the transaction id in the IX entry to access the appropriate transaction table entry. Having a central location to find these updates is essential when servicing a range read request (see Section 4). Each of these write set entries includes the range ID associated with the record, as this information cannot be calculated inside the MVCC mechanism, but is only known to the record manager that is external to MVCC component.

3.3 Range Access Conflicts

3.3.1 IX - IX Interactions

The IX access at a range does NOT impede any other transactions from posting an IX access at the range. Just like in pessimistic multi-granularity locking, IX accesses are "compatible" and do not conflict. This differs from how record updates in the MVCC$_{\text{rec}}$ table are handled (i.e., record "X accesses"). Thus, a range object can have many IX accesses, multiple IX's can be for uncommitted transactions, and IX accesses need not be in strict timestamp order. Thus an IX access never precludes the posting of another IX access by a different transaction.

Table 1: Multi-granularity access modes and their conflicts. We support only modes IX, S, and SIX for ranges.

Mode	IS	IX	S	SIX	X
IS	N	N	N	N	Y
IX	N	N	Y	Y	Y
S	N	Y	N	Y	Y
SIX	N	Y	Y	Y	Y
X	Y	Y	Y	Y	Y

3.3.2 IX - Range Read Interactions

Range reads are handled in a way that is similar, but not identical to individual record reads. As with a record, we maintain a "last read" time for a range (see the MVCC$_{\text{ran}}$ table entry in Figure 2). IX accesses cannot be posted to the range if they are earlier than this "last read" time, otherwise the transaction would be posting a phantom update that a scanner (with a later timestamp) should have seen. Thus, while there are no write-write conflicts captured or detected in the MVCC$_{\text{ran}}$ table, read-write conflicts are detected, where the MVCC$_{\text{ran}}$ is what is being read (i.e., it has the equivalent of an S lock on it). This conflict is easily detected during an IX access, in the same way that an update conflict at an individual record read in the MVCC$_{\text{rec}}$ table is detected during a write access.

A read is more complicated at a range than it is at a record. A range read access must ensure that the reader transaction's timestamp is earlier than the IX accesses of all uncommitted transactions. Since there is no guaranteed order to the list of IX's associated with a range, the entire list of IXs needs to be examined for potential conflicts. This is different than an individual record read, which needs to only check the most recently posted version in a record's version list (in the MVCC_{rec} table) to see if the read is earlier than all uncommitted writes (of which there is at most one).

We can optimize read conflict analysis in the MVCC_{ran} table by noting that any read with a timestamp earlier than the "last read" time in the range entry cannot conflict with any uncommitted IX accesses, otherwise the reader that posted that "last read" time would not have been able to perform a scan at that time. We cannot use equality of read access time with "last read" time because of SIX accesses, which we discuss next.

3.3.3 SIX Accesses: Updates while Scanning

A fairly common situation is that a transaction wants to read a range, perhaps looking for an appropriate record to update, and then wants to update the record (in the same range). Pessimistic multi-granularity locking provided an SIX lock to deal with this case. An SIX lock cuts down on the number of items examined by the concurrency control scheme, since it enables the transaction to read the range without setting individual read locks on all records, and permits the transaction to update a record in the range by setting an X lock on its target record. For the same efficiency reasons, we also support the equivalent of SIX access in our MVCC approach.

Managing times is a little subtle for this case (see also the read optimization just presented in Section 3.3.2), but it is not difficult. When presented with an SIX access, the range item is accessed in the MVCC_{ran} table and conflict tests are made. The transaction time for SIX must be *later* than the "last read" time currently on the range item, as befits an IX access. Further this transaction time must be earlier than all uncommitted IX accesses, thus testing the SIX read ability for IX conflicts as would be done by a range reader. If both tests are passed, the range's "last read" time is set to the SIX time and an IX entry is posted to the range as well. The fact that the "last read" time and an IX time can be the same is the reason why we require range reads to be earlier than all uncommitted IXs, even if the read time EQUALS the current "last read" time.

3.4 Locks vs Versions

3.4.1 Pessimistic Concurrency Control

We use a hash table to access records (and their versions) in Deuteronomy's MVCC component. Using a hash table and recording accesses is also the classic way that lock managers work as well. Also, similar to locking, we know at the time of access whether an access succeeds or not. No validation is needed. This has traditionally been called pessimistic concurrency control.

3.4.2 The Difference between Versions and Locks

A big difference between locks and version management is that locks can be released (discarded) at the end of a transaction. In the Deuteronomy architecture, record version entries in the MVCC table (and the full record payloads themselves) must be kept in the TC until the updates have been applied at the DC (i.e., the data store). This is one of the extra costs of MVCC, and the hash table for MVCC is typically much larger than a corresponding lock manager table. Note, however, that we also use the versions as a TC cache as well [16].

It is the dramatic reduction in conflicts via using versions that makes it possible to avoid blocking on conflicts and instead simply abort a transaction making a conflicting access. However, lack of blocking should not be confused with optimistic concurrency control. Indeed, one could abort transactions making a conflicting access when using two phase locking. This is not usually done because of conflict frequency. Using a multi-version approach dramatically reduces this frequency. Our view is that performance is improved when both validation and blocking are avoided.

4. SERVICING A RANGE REQUEST

The previous sections discussed how ranges are defined and how conflicts are detected and dealt with in the MVCC component. In this section, we describe how ranges of records are materialized as a result of a transaction issuing a range scan request.

4.1 Overview

Ranges are not stored as versioned objects within our MVCC_{ran} table. Nor are they managed as versioned objects by the Version Manager. Rather, range results are generated only when a transaction requests to read a range. We generate records of the range incrementally as a transaction processes the records in an order defined by the request. In this way, it is not required that the entire range be materialized within our transaction component, which would lead to burdensome memory overhead. Only the parts of the range that are incrementally read and processed by the transactional user are materialized.

A salient feature of the Deuteronomy components we have built is that while the TC uses multi-version concurrency control, the DC is a *single-version* data store (see [15]). We merely assume that the DC will contain only the latest version of a record sent to it from the TC. Further, these latest versions do not present a transactionally consistent view of the data. It is only when the DC data is combined with the appropriate versions in the version manager at the TC (i.e., identified by the IX updates) that it is possible to construct a transactionally consistent view of the data.

A transactionally consistent view of a record is trivial to construct. If a record is in the version store at the TC, we chose the version to read that is associated with the latest committed transaction ≤ the timestamp of the requesting transaction. If the record is not in the version store at the TC, then it has not been updated since it was stored in the DC and hence we request the record from the DC. In both cases, we protect the version we are reading by posting, if need be, a "last read" time that ensures this version can only be changed at a time that is later than our reader transaction (Section 3.3.2).

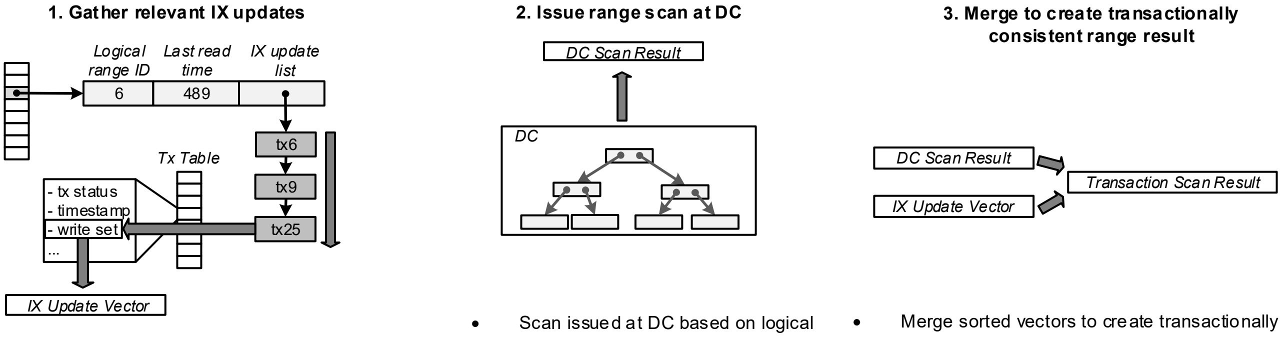
4.2 Transactionally Consistent Ranges

4.2.1 The Basic Idea

Figure 3 depicts the basic idea of how we generate transactionally consistent ranges. This process consists of the following steps.

1. Determine the range objects that intersect with the range requested by the user; this determines the exact set of range objects that play a role in the scan.
2. Find the IX entries for each such range whose transaction timestamps are ≤ the time of the range read. Find the updates for these transactions (using the transaction write set) that are part of the range. Filter this set of updates, selecting only the most recent updated versions, and discarding earlier ones.
3. Issue a range scan against the DC (e.g., the Bw-tree). The scan boundary is defined by the high and low keys of the logical range we are currently processing (or a tighter boundary if the user-provided keys are more constrained).
4. "Merge" the sorted list of record versions from step 2 with the scan result retrieved from the DC in step 3.

This processing is done incrementally, one logical range at a time. If the user-provided scan boundaries span multiple logical ranges, we repeat these steps for each logical range. See Section 4.4 for details.



- IX updates gathered from tx write sets
- Keys can be gathered from $MVCC_{rec}$ table entry
- Payloads (optional) read from Version Manager.
- Vector sorted by key

- Scan issued at DC based on logical range boundaries (or tighter boundaries if user-provided key range is more constrained).
- Returned result is ordered by key

- Merge sorted vectors to create transactionally consistent scan result.
- If entries are equal, favor the IX update over the DC scan result, since we are sure this is the version the transaction should see.

Figure 3: An outline of the steps required to service a range read request in Deuteronomy. A set of IX updates not yet applied at the DC are merged with the DC scan result to form a transactionally consistent range result. Scan results are incrementally built in manageable batches and returned to the transaction.

4.2.2 Gathering Records of a Range

We need to determine how to access versions of records relevant to the range requested by a user. This user-requested range is expressed in terms of a low key and perhaps optionally, a high key (both the low and high key can be open-ended as well). Since we incrementally deliver ranges, we start with the records that share the same logical range object as the low key, and access additional records as the range is progressively delivered to the user. Our strategy is thus to access relevant records that are included in one logical range object before proceeding to the next range object if that is required.

Transaction Component Records (Step 2): We query the $MVCC_{ran}$ table for the versions of records that are appropriate for the range. These are the versions for records that have been updated recently and should be visible to the scanning transaction. Recall that each logical range object has a list of IX postings that indicate what transaction has updated the range object. Each of these transactions must be committed for the range read request to have succeeded. Further each of the transactions has a timestamp that determines its place in the serializable schedule. For each IX access that precedes the read time requested, we scan the associated transaction's list of updates (using the write set for that transaction), looking for record versions that are associated with the range object, which is included in the MVCC entry. When we have gathered all such updates, we prune them so that only the latest update to the range object that is earlier than the read time remains in our key-sorted list of updates to the range object.

Data Component Records (Step 3): We access records from the DC, starting with the low key of the range that is of interest. This is either the low key from the user request, or the low key of the logical range object (when the user requested range intersects more than one range). Records from the DC are incrementally delivered (see Section 4.4).

4.2.3 The Merge (Step 4)

We now describe the basic idea of how to perform the merge between the TC and DC records. However, please see Section 4.3 for a subtle difficulty and Section 4.4 for the final answer.

The record versions from the DC represent the most recent versions of records posted to the DC. We expect them to usually be older than record versions we retrieve from the TC. From the TC, we retrieve precisely the updated record versions that we need for reading the range as of the time of the transactional reader. But these will usually be a small subset of the records of the range.

Our strategy then is to merge the sorted list of records in a range result from the DC with the sorted list of updated versions from the IXs for the range to bring the DC range up-to-date with (i.e., transactionally consistent with) the time of the reading transaction. Thus, whenever a version for a record occurs in both lists, we use the version from the TC. Otherwise (mostly), when a record version is present in only one of the lists, we use that version.

If a record has been recently deleted, it will have a "delete" marker from the TC. That delete stub will remove a version for the record that comes from the DC. If a record has been recently inserted, it will have a record from the TC, but not from the DC. If it has been recently updated, it may have a version from the DC, and from the TC, but we know that the TC version is correct for the time we need, so we use the TC version.

4.3 A Difficulty

4.3.1 Late Inserts

Deuteronomy is a non-blocking, latch-free asynchronous system. Thus many things can be going on concurrently. One of those things can interfere with the success of the merge as described in Section 4.2. The following steps, though uncommon, can occur to produce incorrect results.

1. There is no record version for a record K at the DC.
2. A range reader reads record versions from the TC version manager as of the time requested, and there is no record K in the TC either.
3. An insert operation creates a version for K.
4. The transaction for the insert is committed, and the version for K is eventually posted to the DC.
5. The range read now reaches the DC records that include the newly inserted record K. This record is returned to the range merge function.

Now we have a record from the DC that is too late to appear in the range, but we have no easy way of determining that. The real difficulty here is that we have no way of knowing whether ANY of the records from the DC that do not have matches from the TC are too late (i.e., should not be seen by the transaction) or not. That is, we cannot easily distinguish the too young records from the old records. Every unmatched record from the DC would need to be checked to see whether it was a late insert.

4.3.2 Timestamps to the Rescue

If we knew the transaction timestamp associated with DC records, then determining late inserts becomes trivial. Such a late insert will have a timestamp that is later than the timestamp of the reading transaction. Until this point, we did not have such timestamps at the DC.

In Deuteronomy, a DC must provide idempotent operations. In previous work we described one way to manage idempotence in the DC based on a B+-tree (the Bw-tree) [15, 16]. The idea is to use TC-provided sequence numbers (these are log sequence numbers, or LSNs) that uniquely identify each operation. The Bw-tree associates an LSN with each operation it receives. Occasionally, the TC sends a control operation to the Bw-tree (an "end of stable log" message, or EOSL) denoting that the Bw-tree has seen all operations with LSNs less than the given EOSL value. At this point the Bw-tree is able to consolidate LSN information, using a single page LSN (the highest LSN of all operations on the page). In addition to idempotence, the LSN was also used for log management at the TC, e.g. checkpointing and log truncation.

Given that our current incarnation of the TC uses timestamp ordering, we can use timestamps *instead* of LSNs, so long as we can relate timestamps to LSNs for checkpoint purposes. And we can return these idempotence markers (now timestamps) with the records read from the DC. One complication is when a record is updated multiple times within a transaction. We need to distinguish these updates and ensure that it is the last update that is posted as the result of the transaction. We add a sequence number to our timestamps to deal with this case.

As with LSNs, we do not need to retain a timestamp for every record at the DC. After no more updates can be made with earlier timestamps (previously lower LSNs), we can consolidate a page, providing the entire page with a single timestamp, as we did previously with a page LSN. That means that we would not be able to provide a precise timestamp for every record version from the DC, but it does permit us to easily identify old versions from the versions that result from late inserts. That is, the page timestamp is an upper bound on the record timestamps and hence identifies the records as earlier than any range requested by an active transaction.

Using timestamps instead of LSNs, we can now perform a merge that we know is correct. What we add is a test of each record's version that comes from the DC and that does not have a matching version from the TC. If its timestamp is less than the reading transaction's timestamp, we use it in the resulting range. Otherwise we drop it from the result.

An added bonus of using timestamps is that it enables us to optionally support temporal access methods in a very straightforward way. Many such access methods perform a time-split to separate historical versions from current data [2, 17, 18]. Such time-splits need transaction timestamps to work correctly.

4.4 Incremental Range Delivery

As mentioned previously, range records are not materialized in MVCC_{ran} tables. However, records that have been updated or read *individually* (using the MVCC_{rec} table) will be present within the Version Manager, and that is what permits us to merge DC and TC record versions. But we believe it is important to avoid materializing an entire range, which might contain a few records, but also might contain many thousands of records (or more).

Note that there are two range sizes that enter the picture.

1. The set of records between the user provided range boundaries (the low and high key) might well be small, but it also might be very large (or open-ended), encompassing several of our logical range objects that partition the entire set of keys.
2. The set of records for a single logical range object also can vary enormously in size. So even if the (low key, high key) range requested by a user is small, trying to cache the records of even one logical range object might be costly.

Ranges are represented logically in the MVCC_{ran} table (see Section 2) and are assembled incrementally when we need to deliver records to the user that requested them. We intersect the (low key, high key) range requested by the user with the range partition used for our logical range objects. We then access the DC for the result of this intersection one range partition at a time.

We perform one further breakdown of the records coming in batches from the DC. The Bw-tree returns batches of records, and it can usefully make a batch correspond to a page. So the unit of transfer that we use, and the unit that we materialize at the record manager, is this batch (page) of records. It is this sequence of page size chunks coming from the DC that are merged with the set of record versions for range objects that come from the TC.

We optimize the delivery of the DC record batch by providing a "box" (a "page size" storage allocation) into which the records are returned to the record manager. This avoids multiple allocations, e.g. one per record. It also provides excellent cache locality when we access the records of the batch at the record manager.

TC record versions for a range are materialized one logical range object at a time. If the user-requested range is contained within the key boundaries that define the logical range, then we only materialize records within the user-requested range boundaries.

Once a merge is done between the DC scan result and the relevant TC records, the incremental result is batched, again "in a box", and delivered to the user. We do not retain (cache) records of a range at the TC. Once ranges are delivered to the user, they are dropped. This design can be changed if cursor support is needed. But we would rather have that done outside of the TC.

5. EVALUATION

5.1 Purpose and Goals

Our goal in evaluation is to answer five key questions.

What is the total transaction throughput that Deuteronomy can deliver for transactions that perform scans? Section 5.2 shows that our range-based MVCC scales well on multicore, multisocket hardware to facilitate more than 2.25 million scans per second covering nearly 112 million records per second for the YCSB E workload.

What is the performance impact of adding scan support on point update and lookup operations? Section 5.3.1 shows that range MVCC support increases per-operation latency by about 80%.

What is the throughput cost for range concurrency control? How does performance of TC range scans compare with range scans issued directly against the DC? Section 5.3.2 shows that short serializable scans only decrease scan throughput by 20% compared to direct (un-serializable) scans issued against the DC. Longer scans are up to 35% slower.

OS	Windows® Server 2012
CPUs	4× Intel® Xeon® E5-4650L 32 total cores 64 total hardware threads
Memory	192 GB DDR3-1600 SDRAM
Storage	320 GB Flash SSD Effective read/write: 430/440 MB/s Effective read/write IOPS: 91,000/41,000

Table 2: Details of the system used for experiments.

How do varying workloads impact range scan performance? For example, are large scans that cross many range partitions efficient? Section 5.3.2 shows that long scans achieve scan rates of nearly 250 million records per second (or more than 27 GB/s of data).

Does the TC's use of logical ranges outperform other schemes? Section 5.5.1 shows that our approach provides 43% higher scan throughput than an approach that uses validation. Compared with Silo [34], a high performance integrated key-value store, our component based approach is 19 to 53% slower on short ranges, but avoids the performance penalty that Silo suffers under longer scans (see Section 5.5.2).

We want Deuteronomy to be competitive with main memory databases when the working set of an application fits in Deuteronomy main memory. Consequently, we focus evaluation on loads that highlight range concurrency control overheads rather than I/O device bandwidth limits. All experiments use our previously built DC (including Bw-tree and LLAMA, evaluated elsewhere [14, 15, 16]) and TC augmented with the new range support. All experiments also include the full cost of I/O for durability and all background "garbage collection" (e.g. for pointer stability for lock-free structures, MVCC version reclamation, etc.) needed for stable performance.

Our experimental machine set up is described in Table 2. Each of its four CPUs reside in separate NUMA nodes, which are organized as a ring. The machine has 32 cores total, and each core hosts two hardware threads. Additional hardware threads are used to perform updates against the DC, though the extra threads are lightly loaded since the benchmarks are scan intensive and perform few updates. Recovery logging and DC updates share a single commodity SSD.

5.2 Experimental Workload

For these experiments we use workloads similar to the YCSB benchmarks. For all experiments, the DC database is preloaded with 50 million 100-byte values. Our main workload is the YCSB E workload, which includes range scans. Client threads execute of a mix of 95% scan and 5% update operations. Each operation is executed in a separate transaction to create a transactional workload. Both the keys to update and the scan start keys are chosen randomly using a skewed Zipfian distribution (θ=0.877) that creates an "80-20" hot-cold mix. Scan transactions retrieve 1 to 100 values, with the scan length selected at uniform random. Points are averaged over 5 runs; variance is insignificant in all shown graphs.

5.3 Scalability and Peak Throughput

Figure 4 shows YCSB E scan transaction throughput as the workload is scaled across multiple hardware threads and sockets. With all of the cores issuing transactions, Deuteronomy sustains 2.25 million scan transactions per second in steady state (250 million total records scanned per second) with an insignificant abort rate.

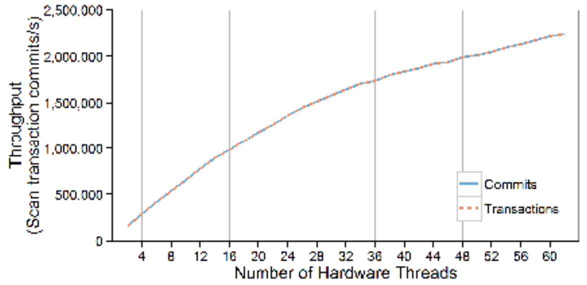


Figure 4: YCSB E scan transaction throughput as the number of client threads is varied.

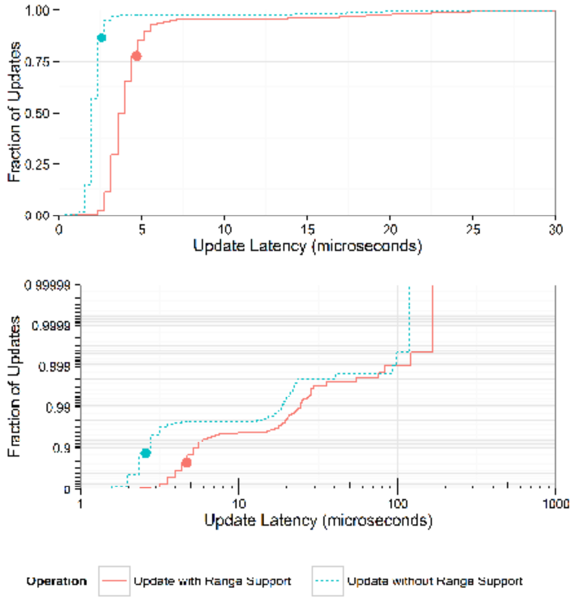


Figure 5: Update operation latency both with and without range support.

Performance scales well across all four CPU sockets until all of the cores are busy performing transactions. Overall, performance is limited by DRAM latency, primarily due to MVCC and range table accesses, read cache and log buffer accesses, and Bw-tree traversal and data page accesses. The large working set hampers the effectiveness of CPU caches and the DTLBs. We experimented with 1 GB super pages for large or relatively fixed-size structures like the recovery log buffers, the read cache, and the transaction table to reduce TLB pressure, but the limited number of TLB entries for super pages made them difficult to exploit.

5.4 Concurrency Control Overhead

5.4.1 Point Operation Latency

Figure 5 breaks down the overhead added due to range support. It shows (in linear and log scale) the cumulative fraction of single update operations that completed within a given time. The figure shows that range support increases update latency by about 80%. Average case latency is shown with a circle marker.

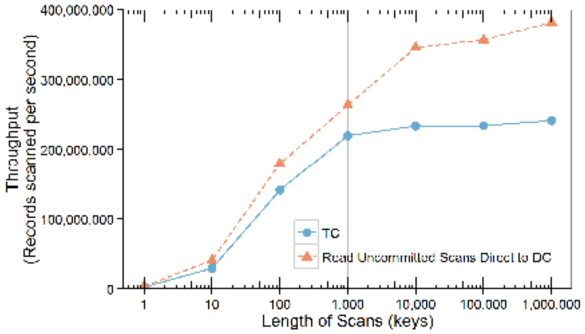


Figure 6: Aggregate scan throughput for varying scan lengths.

The first "shelve" in the graph (which crosses 10 µs) distinguishes updates whose versions must first be read from the DC from those whose version is already local at the TC. The second shelve (which crosses 100 µs) distinguishes DC read operations that execute quickly from those that have to perform delta page consolidations. Overall, the fast path operations are impacted most significantly, since consulting the range MVCC adds a few cache misses to a well-tuned path (without range support the original TC incurred as few as 6 cache misses per operation).

While the impact on latency is non-trivial, the impact on throughput is not large. We ran the OLTP style workload (YCSB with 84% reads, four operation transactions) used in our previous paper [16] with range support disabled and with it enabled. The test used the same thread management and data placement, and ran using 48 threads. Under these conditions, we achieved 1.28M tps without range support, and 1.25M tps with range support turned on. This is a difference of 2.3%. Under these conditions, supporting ranges has only a modest impact on overall performance.

5.4.2 Scan Throughput

Figure 6 shows the scan throughput for varying scan lengths and compares the TC's serializable scan performance to raw non-transactional DC scan performance. In this figure, the direct DC scans are effectively running "read uncommitted" since the DC does not provide any form of concurrency control. The serializable scans in this graph have an insignificant abort rate (not shown). This (and the remaining figures of the paper) are measured with 48 hardware threads executing the YCSB E workload.

Figure 6 shows cost of the TC's range concurrency control: range scan throughput is about 20 to 35% slower with concurrency control than when the scans are issued directly against the DC without concurrency control. Note that 250 million records/s draws data from the DC at 27 GB/s; the TC's serializable scans are fast enough to support even the most demanding applications even when records are fine-grained as in this benchmark.

Finally, Figure 6 also highlights the cost of traversing range boundaries. In this figure, a scan of length 1 million is likely to cross about 49 range boundaries, whereas a scan of length 10,000 is highly unlikely to cross any. The difference is nearly imperceptible for our default partitioning; the measured difference is well within the margin of error.

5.5 Choosing Partitions

Figure 7 explores the impact of partition granularity on overall scan performance. In this experiment, the number of logical range partitions is varied while worker threads execute scans that each collect 10,000 records. The scan-to-update ratio is also varied to explore the performance of workloads with updates.

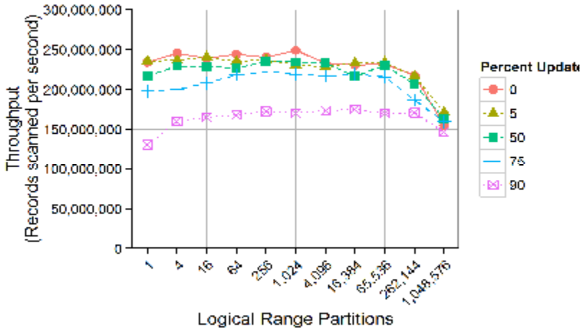


Figure 7: Aggregate scan throughput as logical range partition count is varied.

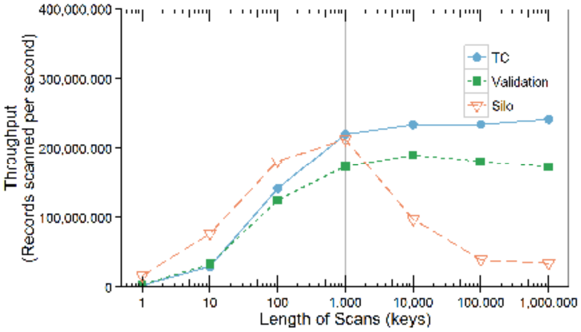


Figure 8: MVCC range scan throughput for various approaches.

Initially, scan throughput is improved by increasing the number of logical range partitions. Updates are fast and scan transaction aborts are insignificant at all points in this graph (hence, only the results for committed scan transactions are shown). So, the improved performance comes from the reduced number of versions chained in each partition's IX update list. Updates are spread across more partitions, and the cost of collecting and merging them is reduced as more partitions are introduced. However, above 64,000 partitions, scan performance begins to degrade. With a large number of partitions, the key space is already divided well enough to keep IX update lists small, and scans are incurring increased overhead by having to cross additional partition boundaries.

5.6 Performance Comparisons

5.6.1 Validation Strategy

Figure 8 compares the performance of Deuteronomy's logical range concurrency control approach with an approach that re-scans ranges at commit time to perform validation. The results show that, as expected, the validation scan almost exactly doubles the cost of scans (compared to the inconsistent scans of Figure 6). Overall, TC range scans are up to 43% faster, using our IX approach, than these same scans using validation. Further, our "validation" scan is simply a re-scan of the range, and does not do the comparison with the original range, so it illustrates only part of the validation cost and misses entirely validation failures. With full validation, the results would be much closer to Silo's.

In fact, this represents a best case for validation based on re-reading the range, since the DC caches scan results between the initial scan and the validation scan. For large scans that cannot be cached the validation approach doubles the use of I/O bandwidth, which we expect to be one of the scarcest system resources.

5.6.2 Comparison with Silo

Finally, Figure 8 also shows Deuteronomy's TC scan performance compared to Silo [34]. Silo's tightly-coupled and optimistic, validation-based approach works well on short ranges, but breaks down as scans get long. For long ranges, Silo explicitly tracks each key read as well as the version of each node seen in a tree in order to ensure that concurrent updates have not affected the scan at commit time. In addition to the basic bookkeeping cost, large scans are more likely to abort due to intervening updates as they cover more keys and remain uncommitted for longer intervals.

Figure 8 shows this affect. On short scans the TC is 19 to 53% slower than Silo, but Silo's optimistic approach breaks down on large scans, and its performance falls significantly. Scan performance in Deuteronomy is stable and predictable independent of scan length.

6. RELATED WORK

Concurrency control methods have existed for as long as there have been database systems. There is too much literature to make an adequate survey here. We focus more narrowly on phantom prevention—while acknowledging that it too has a very long history.

6.1 Multi-granularity Locking

Multi-granularity locking was introduced in System R [1, 6]. This simple idea was a powerful one, as it permitted a resource higher in the granularity hierarchy to be locked without the need to lock each of its included lower level resources. Hierarchies typically included at least tables and their contained records. Frequently, physical pages were also included in the hierarchy.

There is a tension between reduced locking overhead and the granularity of the resource. The larger the granule, the larger the number of accesses that compete to use the resource. The smaller the granule, the more locks are needed when the resource requested is itself large.

One way of dealing with this tension is to support more levels in the hierarchy. For example, pages can be included between tables and records. The difficulty here is that one cannot control access to pages if the concurrency control method knows nothing about page boundaries, as is the case for the Deuteronomy TC. Thus, locking pages effectively forces transactional concurrency control to intrude into access method page management.

An additional problem is lock manager contention that can add substantial overhead and restrict scalability when supporting, e.g., IS mode locks. In many approaches, readers need to post an IS lock. Johnson et al [9] avoid this problem via a form of lock inheritance that avoids many interactions with the lock manager. In contrast, we avoid most of this contention by not supporting IS mode locks, which are unnecessary since we do not support X mode locks on higher level resources. Our contention is further reduced via the use of latch-free data structures in our version manager (which serves also as our "lock" manager).

6.2 Next Key Locking

Another idea that started with System R is called next key locking. The idea is that a lock on a key identifying a record in a table locks not only the record but the key space between that record and the next (or previous) record with its key [1, 6, 20, 21, 27]. When a record is to be insert into a table, the next key is checked. If it is locked (in some manner) then the insertion may be blocked.

Next key locking has the advantage of being fine grained (each adjacent pair of keys defines a range resource) and also be logical and hence not depend on the physical attributes such as how records are assigned to pages. The difficulty is that, when doing an insert, one needs to check a lock on the "next key". In Deuteronomy, where data may be remote from the concurrency control function, discovering the next key adds significant overhead to the insert operation, e.g., in the form of a round-trip traversal to a remote DC.

6.3 Multi-version Methods

When full versioning in a database exists [19, 25], and even temporary versioning (as is done for snapshot isolation in Oracle, SQL Server, and others), those versions can support serializable read transactions. The difficulty has always been to deal with and serialize read-write transactions correctly. Usually, one finds only snapshot isolation being supported, which avoids the need to validate read sets.

Concurrency control to provide serializable transactions using multiple versions has been explored as well. For instance, Cahill et al explore adding a lock manager in Postgres that otherwise used versions to support only snapshot isolation [3]. Jensen et al. and Lomet et al. explored using a lock manager to access multiple versions [7, 22]. A major difficulty was the inherent overhead of locks which were proxies for the versions, and blocking that was used in the face of conflicts. The result was more concurrency when supporting serializable transactions, but much greater overheads.

6.4 Recent Systems

Newly implemented systems have recognized that concurrency control overhead and blocking behavior are serious impediments to achieving high performance. Thus we have seen the exploration of a number of new non-blocking approaches. These approaches are pursued in the context of main memory databases that remove secondary storage and user latencies.

6.4.1 Serial Execution

One of the most straightforward ways to removing blocking is to run transactions serially one after the other. VoltDB [32] and its academic precursor HStore [31] provide serializable transactions by executing transactions in serial order to completion. The initial version of the HyPer main-memory database also executed transactions sequentially [11]. Calvin [33] is a partitioned distributed system that orders transaction execution deterministically to avoid cross-partition contention. Transactions in Calvin run serially on each partition.

6.4.2 Optimistic Methods

The Hekaton main-memory OLTP engine uses an optimistic multi-version concurrency control technique [5, 12]. The basic idea is as follows: records are multi-versioned and versions have disjoint valid time ranges. Transactions read records as of a logical read time, while updates create new versions. At the end of execution, transactions are given a unique timestamp denoting the commit time of the transaction; this is also the write time for any new versions created by the transaction. For isolations levels stronger than snapshot, this technique requires validating read sets, and in the case of serializable isolation, also requires re-doing all scans in order to perform phantom detection. Since Hekaton's target is snapshot isolation, this scheme works well. The goal in Deuteronomy is to provide efficient execution of transactions running in full serializable isolation. We deem concurrency control schemes that require post-processing validation too high a price to pay to achieve serializability. In Section 5.5, we showed that our approach is up to 43% more efficient than a validation-based approach. This is a best-case scenario for validation, since the workload was read-only (no aborts) and memory bound. The situation would be much worse if validation required extra I/O to perform the second scan. Since Deuteronomy is not a memory-only engine, we want to avoid validation at all costs.

6.4.3 Other Systems

Silo is a high-performance transaction processing engine targeting multi-core machines with large main memories [34]. Transactions in Silo keep write sets private during processing. During commit processing, a transaction installs its write set in the underlying data structure (the Masstree range index [24]). During this time, it temporarily locks (latches) the record while installing its write set, aborting if it detects a conflict. Silo provides serializable transactions. It detects phantoms by versioning the leaf nodes of the Masstree. Range scans record the version of the leaf pages they encounter during a scan, and re-check the versions during commit to ensure the versions are the same. This scheme couples concurrency control with data storage; Deuteronomy, on the other hand, performs purely logical concurrency control, including range concurrency.

HyPer recently explored the use of timestamp order concurrency control in main-memory database systems and showed that this technique performs well in modern database architectures [26]. HyPeR recently added support for serializable transactions based on multi-versioning, precision locking, and validation [28].

VLL [30] is a lightweight locking technique designed for main-memory database system. VLL supports range concurrency by defining range boundaries using prefixes of key space. While this technique is purely logical, nonetheless unlike Deuteronomy, VLL is a (pessimistic) locking scheme.

7. DISCUSSION

7.1 Serializability

Serializability has an extra cost that manifests itself in two ways. (1) Either concurrency is reduced, or (2) execution path is increased. This pushes users toward lower levels of isolation. This is unfortunate as it is serializability that provides the illusion to applications that they have the entire system to themselves: if a transaction executes correctly in a single user setting, it will execute correctly in a fully concurrent setting. Our goal is serializability with a sufficiently small penalty, both in performance and in concurrency, as to enable it to be the default for users.

The most challenging aspect, as we focus on in this paper, is preventing phantoms when a transaction reads sets of data. Without an index, the usual approach is to evaluate predicates during a table scan. Preferable when possible, is to replace the table scan with a key range scan as then only the key range needs to be read and protected. This has been the focus of the current paper.

We had previously implemented an MVCC mechanism that provided serializability for all cases except for dealing with ranges [16]. This provided both our starting point and a very large challenge. As a starting point, we had achieved very high performance with a very low abort rate, which was great. However, this meant that any added overhead or lost concurrency when dealing with ranges risked having a large relative performance impact.

7.2 Concurrency and Aborts

When a timestamp is defined at transaction end, the longer the transaction executes, the more conflicts it is likely to encounter since a version is subject to conflict from access time to transaction end. A start time timestamp avoids this difficulty, as done with our timestamp order approach. And, indeed, once a range object is successfully read (the read is posted as of the transaction's start time) its read time will ensure that the incremental reading of records at the range cannot be subject to interference.

Concurrent write transactions early in the range reader's execution may be forced to abort. Or it may need to abort because of an uncommitted update of a record in the range that was posted prior to the range read. But writers that start after the range reader are not interfered with as they update records with versions that will come after the range reader's transactions. Hence these writers will neither themselves abort because an earlier range version is being read nor cause abort difficulties for the earlier range reader.

The "magic" of MVCC is that range reads can be concurrent with record updating. This is exactly what is achieved with snapshot isolation, in exactly the same way- the range reader reads the earlier version while updaters modify later versions. There is some concurrency lost when serializability is the isolation level. But much of the MVCC concurrency gain from snapshot isolation can be realized also with MVCC serializable concurrency control.

7.3 Overhead and Contention

We cap the resource hierarchy at two levels. Every level of a multi-granularity resource hierarchy increases overhead for simple record updates by potentially requiring a check of the resources in the hierarchy path leading to a record.

We targeted ranges that are a modest fraction of the "table size", on the order of a few tenths of one percent. Modest size ranges work to spread out IX postings and hence avoid hot spots in our MVCC hash table. Modest range size reduces the potential that any given write will impact and perhaps conflict with concurrent range reads. This also reduces the overhead of conflict checking since the IX accesses are spread out over a larger number of ranges.

This existence of multiple versions and use of MVCC reduces the access contention from locks. Low transaction latency also reduces contention, which results from Deuteronomy treating a transaction as committed once its commit record is in the volatile recovery buffer (often called "fast commit") [4]. We wait for transaction durability only for notifying clients. Contention on physical resources can also be serious, including access to the recovery log [10]. This contention is reduced via our latch-free access to these resources [16], particularly the use of the FAI atomic instruction,

7.4 Pessimism vs Optimism

One can debate whether transactions only accessing record level resources should use optimistic vs pessimistic concurrency control. Record level conflicts tend to be quite low. However, range objects have a much higher "conflict profile". This reduces greatly the desirability of optimistic methods, even with MVCC.

The higher potential for conflict with ranges means that between the original access of records in a range and subsequent commit time validation, there is a greatly increased likelihood of validation failing. Pessimistic concurrency control "validates" at access time. This coupled with the TO approach means that later writers (with later timestamps) are not impacted at all. And earlier writers are already partially executed. So a successful posting of a range access has only modest impact on writers that come to the resource later, regardless of their timestamps.

The pessimistic approach does have "up front" overhead, which can be seen in the comparison with Silo in section 5.6.2. However, as also seen there, it avoids the problems that occur with optimistic methods as scans get longer.

7.5 Performance

The performance results we report here are for serializable transactions. The high transaction throughput and the low abort rate for supporting serializable isolation validate our design decisions, and provide additional confirmation that a carefully architected and modular system can achieve terrific performance.

8. Conclusion

This paper presented a high performance range concurrency control technique that extends multi-version timestamp ordering to support range resources and fully supports phantom prevention. We described a protocol to define a set of logical ranges that is compatible with the Deuteronomy TC:DC logical and physical partitioning. This protocol uses the inner leaf nodes of a range-based DC to provide the TC with a balanced logical partitioning of the key space. The TC uses this key partitioning to define a set of logical range objects that are used in a multi-granularity resource hierarchy in our multi-version timestamp order scheme. These range resources are used to both detect serializable conflicts in the timestamp order, as well as help range readers construct a transactionally consistent range scan result. Our technique incrementally returns range results to the user in manageable batches, thus only materializing relevant record results. Our evaluation shows that this concurrency control scheme reaches scan speeds of nearly 250 million records/s (more than 27 GB/s) on modern hardware, while providing serializable isolation complete with phantom prevention. The approach is 43% faster than an approach that relies on re-reading the range scan for validation.

9. REFERENCES

[1] M. M. Astrahan et al. System R: Relational Approach To Database Management. ACM TODS 1(2): 97-137, 1976.

[2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. VLDBJ 5(4): 264-275, 1996.

[3] M. J. Cahill, U. Rohm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. In Sigmod, 2008, pp. 729 – 738.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, D. A. Wood. Implementation Techniques for Main Memory Database Systems. SIGMOD, 1984, pp. 1-8

[5] C. Diaconu et al. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In SIGMOD, 2013, pp. 1243-1254.

[6] J. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger. Granularity of Locks in a Shared Data Base. In VLDB, 1975, pp. 428-451.

[7] C. S. Jensen and D. B. Lomet. Transaction Timestamping in (Temporal) Databases. In VLDB, 2001, pp. 441-450.

[8] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. TKDE 11(1): 36-44, 1999.

[9] R. Johnson, I. Pandis, A. Ailamaki: Improving OLTP Scalability using Speculative Lock Inheritance. PVLDB 2(1): 479-489, 2009.

[10] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, A. Ailamaki. Scalability of Write-Ahead Logging on Multicore and Multisocket Hardware. VLDBJ 21(2): 239-263, 2012.

[11] A. Kemper and T. Nuemann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In ICDE, 2011, pp. 195-206.

[12] P-A Larson et al: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309, 2011.

[13] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In CIDR, 2011, pp. 123–133.

[14] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. PVLDB 6(10): 877-888, 2013.

[15] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In ICDE 2013: 302-313.

[16] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High Performance Transactions in Deuteronomy. In CIDR, 2015.

[17] D. B. Lomet and B. Salzberg. Access Methods for Multiversion Data. In SIGMOD, 1989. pp. 315-324.

[18] D. B. Lomet and F Nawab. High Performance Temporal Indexing on Modern Hardware. To appear in ICDE, 2015.

[19] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y Zhu. Immortal DB: Transaction Time Support for SQL Server. In SIGMOD, 2005, pp. 939-941.

[20] D. B. Lomet and M. F. Mokbel. Locking Key Ranges with Unbundled Transaction Services. PVLDB 2(1): 265-276, 2009.

[21] D. B. Lomet. Key Range Locking Strategies of Improved Concurrency. In VLDB, 1993, pp. 655 – 664.

[22] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-Version Concurrency via Timestamp Range Conflict Management. In ICDE, 2012, pp. 714-725.

[23] D. Lomet, A. Fekete, G. Weikum, M. Zwilling. Unbundling Transaction Services in the Cloud. In CIDR, 2009, 123–133.

[24] Y. Mao, E. Kohler, R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In EuroSys, 2012, pp. 183-196.

[25] E. McKenzie and R. Snodgrass. Extending the Relational Algebra to Support Transaction Time. In SIGMOD, 1987, pp. 467-478.

[26] H. Mühe, S. Wolf, A. Kemper, and T. Neumann. An Evaluation of Strict Timestamp Ordering Concurrency Control for Main-Memory Database Systems. In IMDM Workshop, 2013, 74-85.

[27] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operation on B-Tree Indexes. In VLDB, 1990, pp. 392-405.

[28] T. Neumann, T. Mühlbauer, A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In SIGMOD, 2015, pp. 677-689.

[29] D. P. Reed. Implementing Atomic Actions on Decentralized Data. ACM TOCS 1(1): 3-23, 1983.

[30] K. Run et al. VLL: A Lock Manager Redesign for Main Memory Database Systems. VLDBJ: 1-25, 2015.

[31] M. Stonebraker et al. The End of an Architectural Era: (It's Time for a Complete Rewrite). In VLDB, 2007, pp. 1150-1160.

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

[33] A. Thomson et al. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In SIGMOD, 2012, pp. 1-12.

[34] S. Tu et al. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*, 2013, pp. 18-32.

[35] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.