Transaction Processing Techniques for Modern Hardware and the Cloud

Justin Levandoski Sudipta Sengupta Ryan Stutsman Rui Wang

Microsoft Research

Abstract

The Deuteronomy architecture provides a clean separation of transaction functionality (performed in a transaction component, or TC) from data storage functionality (performed in a data component, or DC). For the past couple of years, we have been rethinking the implementation of both the TC and DC in order for them to perform as well, or better than, current high-performance database systems. The result of this work so far has been a high-performance DC (the Bw-tree key value store) that currently ships in several Microsoft systems (the Hekaton main-memory database, Azure DocumentDB, and Bing). More recently, we have been completing the performance story by redesigning the TC. We are driven by two main design goals: (1) ensure high performance in various deployment scenarios, regardless of whether the TC and DC are co-located or distributed and (2) take advantage of modern hardware (multi-core and multi-socket machines, large main memories, flash storage). This paper summarizes the design of our new TC that addresses these two goals. We begin by describing the Deuteronomy architecture, its design principles, and several deployment scenarios (both local and distributed) enabled by our design. We then describe the techniques used by our TC to address transaction processing on modern hardware. We conclude by describing ongoing work in the Deuteronomy project.

1 Deuteronomy

The Deuteronomy architecture decomposes database kernel functionality into two interacting components such that each one provides useful capability by itself. The idea is to enforce a clean, layered separation of duties where a transaction component (TC) provides concurrency control and recovery that interacts with one or more data components (DC) providing data storage and management duties (access methods, cache, stability). The TC knows nothing about data storage details. Likewise, the DC knows nothing about transactional functionality — it is essentially a key-value store. The TC and DC may be co-located on the same machine or may be separated on different machines by a network. Regardless, the TC and DC interact through two main channels: (1) record operations, supporting the so-called CRUD (create, read, update, delete) interfaces, and (2) control operations that enforce the write-ahead log protocol and enable clean decomposition of components. Details of the Deuteronomy architecture have been discussed in prior work [10].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

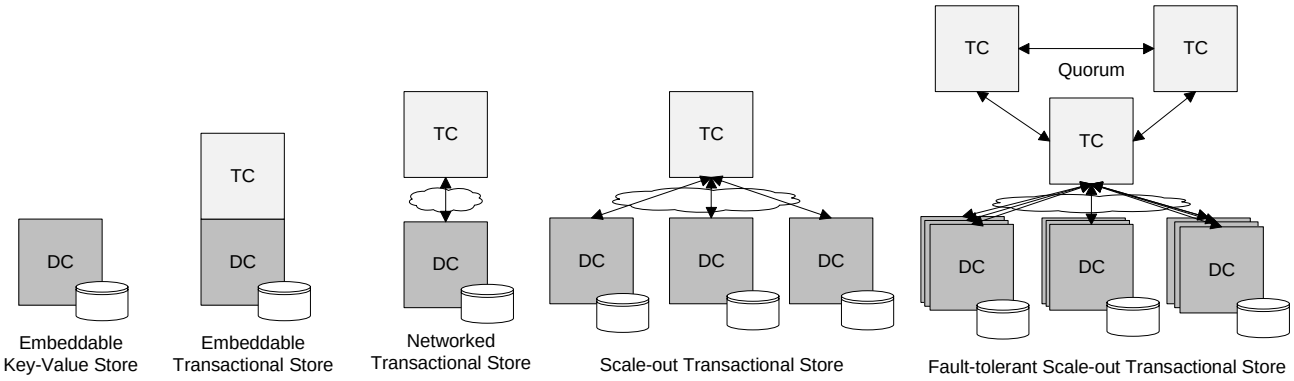

Figure 1: Different deployment scenarios (local and distributed) for Deuteronomy.

1.1 Flexible Deployment Scenarios

A salient aspect of Deuteronomy is that it enables a number of interesting and flexible deployment scenarios, as depicted in Figure 1. These scenarios span both distributed configurations and as well as cases where components coexist on the same machine. For instance, applications needing a high-performance key-value store might only embed the data component (e.g., the Bw-tree). A transaction component can be added on top to make it a high-performance embedded transactional key-value store. The TC and DC can also interact over a network when located on different machines. A TC can provide transaction support for several DCs (serving disjoint data records) in order to scale out. Finally, both the TC and DC can be replicated to form a fault-tolerant, scale-out transactional store. Today, such flexibility is especially important as applications span a wide array of deployments, from mobile devices, to single servers, all the way to on-premise clusters or cloud environments; this is certainly the case at Microsoft. Thus, systems can mix, match, and configure Deuteronomy's reusable components in a number of different ways to meet their data management needs.

1.2 The Current Deuteronomy Stack

After finishing an initial implementation of Deuteronomy that demonstrated its feasibility [10], we began working on a redesign of both the TC and DC to achieve performance competitive with the latest high performance systems. Our first effort resulted in a DC consisting of the Bw-tree latch-free access method [8] and LLAMA [7], a latch-free, log structured cache and storage manager. The result was a key-value store (DC) that executes several million operations per second that is now used as the range index method in SQL Server Hekaton [3] and the storage and indexing layer in other Microsoft products, e.g., Azure DocumentDB [4] and Bing [2]. This effort also showed that in addition to a TC/DC split, the DC could be further decomposed to maintain a hard separation between access methods and the LLAMA latch-free, log structured cache and storage engine, as depicted in Figure 2. With a DC capable of millions of operations per second the original TC became the system bottleneck. The rest of this article summarizes our effort to build a high-performance TC. We begin by discussing the high-level architecture and design principles in Section 2. We then provide a summary of the techniques that allow the TC to achieve high performance (Sections 3 to 5). We end by providing a summary of ongoing work in the Deuteronomy project in Section 6.

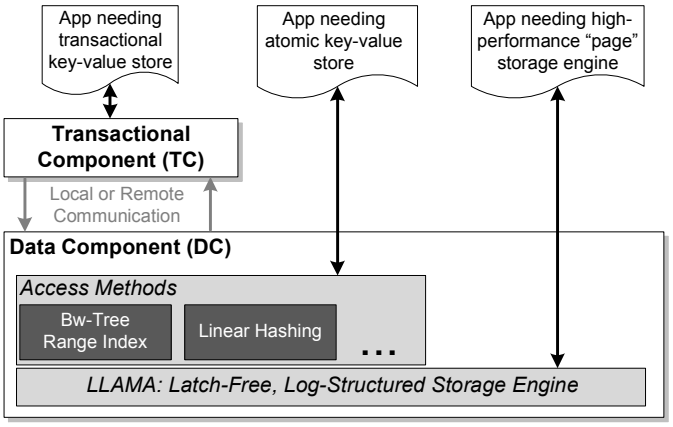

Figure 2: The Deuteronomy layered architecture. In addition to a TC/DC split, we further support a hard separation between access methods and a high-performance storage engine targeting modern hardware.

2 Transaction Component Design

2.1 Design Principles

As we outlined in the previous section, the Deuteronomy architecture enables a number of different deployment scenarios. Whether the TC and DC are co-located, or remote (or replicated), the design of our transaction component aims to achieve excellent performance in any scenario. We are driven by the following principles.

1. *Exploit modern hardware*. Modern machines, whether located on premise or in a data center, currently contain multiple CPU sockets, flash SSD, and large main memories. We aim to exploit the performance advantages of this modern hardware by using latch-free data structures, log structuring, and copy-on-write delta updates. These techniques avoid update-in-place and are well-suited to modern multi-core machines with deep cache hierarchies and low-latency flash storage. All components in the Deuteronomy stack (both TC and DC) take advantage of these techniques.

2. *Eliminate high latency from the critical paths*. DC access latency can limit performance, especially when the TC and DC are located on separate machines (or even sockets). This is particularly bad for hotspot data where the maximum update rate of 1/latency (independent of concurrency control approach) can severely limit performance. TC caching of hot records is essential to minimizing latency.

3. *Minimize transaction conflicts*. Modern multi-version concurrency techniques demonstrate the ability to enormously reduce conflicts. Deployed systems like Hekaton [3] have proven that MVCC performs well in practice. We also exploit MVCC in the TC in the form of multi-version timestamp ordering.

4. *Minimize data traffic between TC and DC*. Data transfers are very costly. Our "distributed" database kernel requires some data to be transferred between TC and DC. Our design aims to limit the TC/DC traffic as much as possible.

5. *Exploit batching*. Effective batching can often reduce the per "item" cost of an activity. We exploit batching when shipping data updates from the TC to the DC.

6. *Minimize data movement and space consumption*. Obviously, one wants only "necessary" data movement. We use pure redo recovery and place data in its final resting place immediately on the redo log within the TC, avoiding what is very frequently a major performance cost, while reducing memory footprint as well.

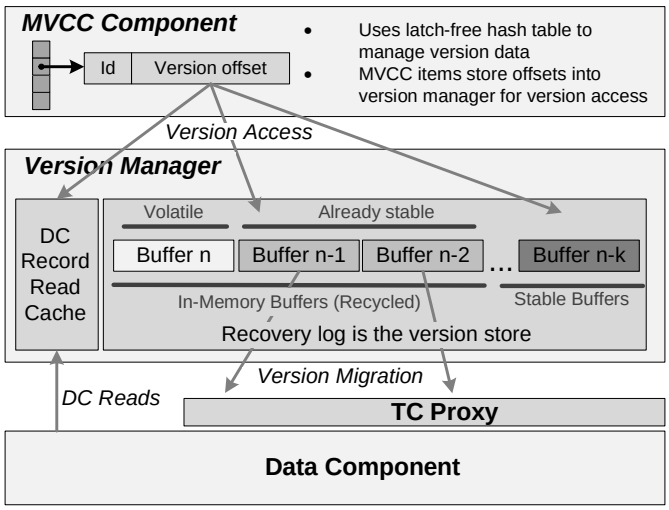

Figure 3: The Deuteronomy TC architecture and data flow. The TC consists of an multi-version concurrency control component and a version manager to manage the redo log and version cache. A TC Proxy is co-located with the DC to enable fast parallel replay of committed records to the database.

2.2 Transaction Component Architecture

Figure 3 presents the TC architecture, illustrating the flow of data within it and between TC and DC. The TC consists of three main components: (a) an MVCC component to manage the concurrency control logic; (b) a version manager to manage our redo log (also our version store) as well as cached records read from the DC; and (c) a TC Proxy that lives beside the DC and whose job is to submit committed operations to the DC, which maintains database state.

Caching Since we use MVCC, versions must be cached somewhere for concurrency control purposes. Versions resulting from updates are written into the redo recovery log. These recovery log versions are accessed via the MVCC component, which stores *version offsets* as part of its version entry and requests them through the version manager interface. The version manager uses the redo log as part of the TC record version cache. In-memory log buffers are written to stable storage and retained in memory to serve as a version cache until they are eventually recycled and reused.

Versions of data not recently updated need to be acquired from the DC. To make them accessible to MVCC, the version manager retains these versions in the read cache. Both read cache and recovery log buffers are subject to different forms of log structured cleaning (garbage collection). Thus, an MVCC request to the version manager could hit (1) the read cache; or (2) a log buffer (in-memory or stable).

To minimize data movement, we immediately post updates in their final resting place on the recovery log. Immediately logging updates means that uncommitted updates are on the log without any means of undoing them if their transaction is aborted. So we cannot post updates to the DC until we know that the containing transaction has committed. Updates from aborted transactions are simply never applied at the DC.

TC Proxy The TC proxy is a module that resides on the same machine as the DC. The TC Proxy receives log buffers from the version manager after the buffer is made stable and posts updates to the DC as appropriate. Since we use pure redo logging, these updates are blind, in that they do not require first reading a pre-image of the record for undo. Posting can only be done after the transaction responsible for the update has committed. This posting is part of log buffer garbage collection when the TC Proxy and DC are collocated with the TC. Otherwise cleaning occurs once the buffer has been received by a remote TC Proxy. Posting updates to the DC

is not part of the latency path of any operation and is done in the background. However, it is important for it to be somewhat timely, because it constrains the rate at which MVCC entries can be garbage collected.

3 Concurrency Control

The TC uses timestamp ordering (TO) as its concurrency control method. TO is a very old method [13] (including its multi-version variant); the idea is to assign a timestamp to a transaction such that all of its writes are associated with its timestamp, and all of its reads only "see" versions that are visible as of its timestamp. A correct timestamp order schedule of operations using the timestamps is then enforced. Transactions are aborted when the timestamp ordering cannot be maintained. While TO never took off in practice, recent work from us [9] and others [6, 12] has shown that TO performs well when all concurrency control metadata is kept in memory. In addition, a real advantage of TO is that it enables serializable isolation without needing a validation step at the end of the transaction [5].

We use a multi-version form of TO [1]. The TC tracks transactions via a transaction table. Each entry in the table denotes a transaction status, its transaction id, and a timestamp issued when the transaction starts. Version information for a record is managed by a latch-free MVCC hash table (see [9] for details). The entry for each version in the table is marked with the transaction id that created the version. This permits an easy check of the transaction information for each version, including its timestamp. The status in the transaction table entry indicates whether the transaction is active, committed, or aborted.

In order to keep the MVCC table from growing unbounded to the size of the entire database, we periodically perform garbage collection to remove entries. We compute the oldest active transaction (the one with the oldest timestamp), or OAT, which is used to determine which version information is safe to garbage collect from the table. We conservatively identify versions that are not needed by transactions or that are available from the DC. These items are (1) any updated version older than the version visible to the OAT; these versions are never needed again. (2) Any version visible to the OAT but with no later updates can also be discarded once it is known to have been applied at the DC. We are guaranteed to be able to retrieve such a record version from the DC.

4 Version Management

Providing fast access to versions is critical for high performance in Deuteronomy. The TC serves requests for its cached versions from two locations. The first is directly from in-memory recovery log buffers. The second is from a "read cache" used to hold hot versions that may not have been written recently enough to remain in the recovery log buffers.

4.1 Recovery Log Caching

The TC's MVCC approves and mediates all updates, which allows it to cache and index updated versions. The TC makes dual-use of the MVCC hash table to gain performance: the act of performing concurrency control naturally locates the correct version contents to be returned. When a transaction attempts an update, it is first approved by MVCC. If the update is permitted, it results in the new version being stored in a recovery log buffer within the version manager. Then, an entry for the version is created in the MVCC hash table that associates it with the updating transaction and contains an offset to the version in the recovery log. Later reads for that version that are approved by the MVCC directly find the data in memory using the version offset. Thus, in addition to concurrency control, the MVCC hash table serves as a version index, and the in-memory recovery buffers play the role of a cache.

Each updated version stored at the TC serves both as an MVCC version and as a redo log record for the transaction. The TC uses pure redo logging and does not include "before" images in these log records; this ensures that the buffers are dense with version records that make them efficient as a cache. Versions are written immediately to the recovery log buffer to avoid later data movement. Consequently, an update (redo log record) cannot be applied at the DC until its transaction is known to be committed, since there is no way to undo the update. The TC Proxy (§5) ensures this.

Recovery log buffers are written to stable storage to ensure transaction durability. The use of buffers as a main memory version cache means they are retained in memory even after they have been flushed to disk. Once a buffer is stable, it is given to the TC Proxy so that the records (that are from committed transactions) that it contains can be applied to the DC.

4.2 Read Cache

The recovery log acts as a cache for recently written versions, but some read-heavy, slow-changing versions are eventually evicted from the recovery log buffers when they are recycled. Similarly, hot read-only versions may pre-exist in the DC and are never cached in the recovery log. If reads for these hot versions were always served from the DC, TC performance would be limited by the round-trip latency to the DC. To prevent this, the TC's version manager keeps an in-memory read cache to house versions fetched from the DC. Each version that is fetched from the DC is placed in the read cache, and an entry is added to the MVCC table for it.

The read cache is latch-free and log-structured, similar to recovery log buffers. One key difference is that the read cache includes its own latch-free hash index structure. The MVCC table refers to versions by their log-assigned offsets, and this hash index allows a version to be relocated into the cache without having to update references to it. The TC uses this so that it can relocate hot versions into the cache from recovery log buffers when it must recycle the buffers to accommodate new versions.

Being log-structured, the read cache naturally provides a FIFO eviction policy. The cache is a large ring, and newly appended versions overwrite the oldest versions in the cache. So far, this policy has been sufficient for us; it works symbiotically with the TC's MVCC to naturally preserve hot items. This is because whenever a record is updated it is "promoted" to the tail of the record log buffers, naturally extending the in-memory lifetime of the record (though, via a new version). So far, our metrics indicate there is little incentive for more complex eviction polices.

Finally, the read cache aggressively exploits its cache semantics to eliminate synchronization to a degree that would otherwise be unsafe. For example, its hash index is lossy and updaters don't synchronize on update collisions. Also, readers may observe versions that are concurrently being overwritten. In these cases, the read cache detects (rather than prevents) the problem and treats the access as a cache miss. The concurrently developed MICA key-value store [11] uses similar techniques, though we were unaware of its design until its recent publication.

5 TC Proxy: Fast Log Replay at the DC

The TC Proxy's main job is to receive stable recovery log buffers from the TC and efficiently apply the versions within them to the DC using parallel log replay. The TC Proxy runs co-located with the DC. It enforces a well-defined contract that separates the TC and the DC sufficiently to allow the DC to run locally with the TC or on a remote machine. The TC Proxy receives recovery log buffers from the TC as they become stable, so the buffers act as a natural batch and make communication efficient even when over a network. Finally, it unbundles operations and applies them to the DC as appropriate.

The TC proxy must provide two things to achieve good overall performance. First, log records must be processed as few times as possible. The pure redo design requirement complicates this. Not all of the records

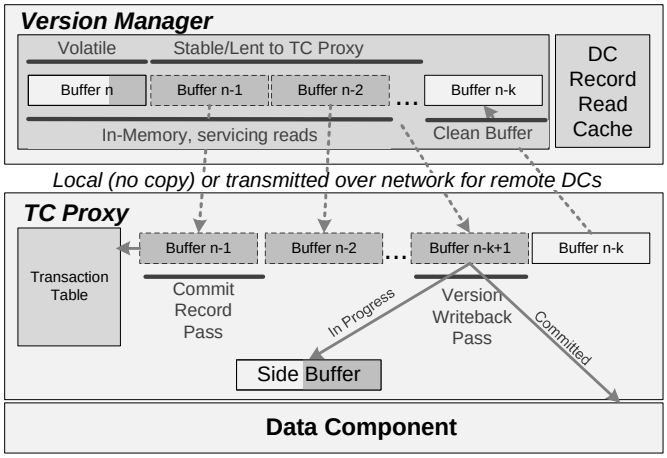

Figure 4: The TC Proxy receives full, stable log buffers from the TC. An eager pass updates transaction statuses; a lazy pass applies committed operations to the DC.

in the received buffers are from committed transactions, so they may have to be applied after a delay. Second, log replay must be parallelized across multiple cores to prevent it from becoming a bottleneck in write-heavy workloads.

Ideally, the TC Proxy would only encounter records of committed transactions when it processes a recovery log buffer. Then, all operations could be applied as they were encountered. To make that ideal scenario "almost true", buffer processing is delayed. Figure 4 depicts this. The TC Proxy immediately scans arriving buffers for commit records and updates its own version of the transaction table to track which transactions are committed (as indicated by encountering their commit records). A second scan applies all of the operations of transactions known to be committed. Operations of aborted transactions are discarded, and operations whose transaction outcome is not yet known are relocated into a side buffer.

This works well when very few operations have undecided outcomes. Delaying the replay scan can minimize the number of operations that must be relocated into a side buffer. In practice, the side buffer space requirement for storing the undecided transaction operations has been tiny, but will vary by workload. A side buffer operation is applied to the DC (or discarded) whenever its transaction's commit (or abort) record is encountered.

The TC Proxy parallelizes log replay by dispatching each received log buffer to a separate hardware thread. Log records are applied to the DC out of log order. The LSN of the commit record for each version is used for idempotence checking, and versions that have already been superseded by newer versions at the DC are discarded. That is, the LSN is used to ensure that the version associated with a particular key progresses monotonically with respect to log order. In the current experiments, this parallel replay naturally expands to use all the cores of a socket under write-heavy loads.

6 Ongoing Work

Our future plans for Deuteronomy involve a number of interesting directions:

- **Range concurrency control**. The TC currently supports serializable isolation for single-record operations, but not phantom avoidance for range scans. We plan to integrate range support into our multi-version timestamp order concurrency control scheme while still maintaining high performance.

- **Scale out**. A number of performance experiments have shown that the TC is capable of handling millions

of transactions per second [9]. A current major thrust of our work is to scale out the number of DCs beneath a TC. In this setting, each DC manages a disjoint set of records while transactional management for all records (across all DCs) is managed by a single TC (see the "scale out transactional store" deployment in Figure 1).

- **Fault tolerance**. Fault tolerance is extremely important, especially if the TC and DC are embedded in a cloud-based system running on commodity machines. We are actively working on replication methods such that a TC or DC can actively fail over efficiently with little downtime.

7 Acknowledgements

This is joint work with David Lomet. We would like to also thank the various product groups at Microsoft (SQL Server, Azure DocumentDB, and Bing) for helping to improve the Deuteronomy stack.

References

[1] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] Bing. **`https://www.bing.com`**.

[3] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, 2013.

[4] Azure DocumentDB. **`https://www.documentdb.com`**.

[5] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4):286–297, 2012.

[6] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *ICDE*, pages 580–591, 2014.

[7] Justin Levandoski, David Lomet, and Sudipta Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10):877–888, 2013.

[8] Justin Levandoski, David Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, 2013.

[9] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High Performance Transactions in Deuteronomy. In *CIDR*, 2015.

[10] Justin Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction Support for Cloud Data. In *CIDR*, pages 123–133, 2011.

[11] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, pages 429–444, 2014.

[12] Henrik Mühe, Stephan Wolf, Alfons Kemper, and Thomas Neumann. An Evaluation of Strict Timestamp Ordering Concurrency Control for Main-Memory Database Systems. In *IMDM*, pages 74–85, 2013.

[13] David P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, M.I.T., Cambridge, MA, 1978.