An architecture for fast and robust memory-optimized OLTP

Kangnyeon Kim Tianzheng Wang Ryan Johnson Ippokratis Pandis*

University of Toronto *Cloudera
knkim, tzwang, ryan.johnson@cs.utoronto.ca ippokratis@cloudera.com

ABSTRACT

The emergence of systems with large main memories and massively parallel processors has triggered a resurgence of new high-performance transaction processing systems. Many of these systems adopt lightweight optimistic concurrency control schemes that are suitable only for a small fraction of transactional workloads. We examine existing systems and argue that—for better or for worse—it is their system architecture that largely dictates the choice of concurrency control they employ. Therefore, it is quite difficult, if not impossible, to adopt a different concurrency control scheme without significant changes to the rest of the system. The architecture also largely determines how the physical conflicts between concurrent threads are dealt, and how recovery is achieved.

We argue that transaction processing systems should be designed from the ground up with four basic requirements: to not heavily rely on partitioning; to provide flexible and robust concurrency control for the logical interactions between transactions; to address the physical interactions between threads in a scalable way; and to have a clear recovery methodology. We report on some early progress in designing a system from the ground up to support the requirements we lay out.

1. INTRODUCTION

Modern systems with large main memories and massively parallel processors have inspired a new breed of high-performance memory-optimized OLTP systems [8, 9, 10, 13, 18]. These systems leverage spacious main memory to fit the whole working set in DRAM with streamlined, memory-friendly data structures. Further, optimizations for multicore and multi-socket hardware allow a much higher level of parallelism compared to conventional database systems. With disk overheads and delays removed, transaction latencies drop precipitously and worker threads can usually execute transactions to completion without interruption. The result is a welcome reduction in contention at the logical level and less pressure on whatever concurrency control (CC) scheme might be in place. A less welcome result is an increasing pressure for scalable data structures and algorithms to cope with the growing number of worker threads that concurrently execute transactions and need to communicate.

Interactions at the logical level. Many designs exploit the reduction in the pressure on CC, by employing very optimistic and lightweight schemes, boosting even further the performance of these systems on suitable workloads. But, as is usually the case, it appears that database workloads stand ready to absorb any and all concurrency gains the memory-optimized systems have to offer. In particular, there is high demand for database systems that can readily serve heterogeneous database workloads, blending the gap between transaction and analytical processing. This trend is at least partly enabled by the improved concurrency and reduced contention offered by memory-optimized systems. Mixed workloads have two significant impacts on CC, however. First, the write/read ratio decreases from 1:2 (e.g. TPC-C) to 1:10 or less (e.g. TPC-E [17]), usually *by increasing the number of reads as the number of writes remains stable*. Second, workloads frequently include some fraction of large transactions that are *read-mostly rather than read-only*—a trend reflected in the TPC-E benchmark. Unfortunately, both of these workload properties result in larger effective concurrency control footprints, putting pressure on the CC scheme. Therefore, going forward and as the industry shifts to heterogeneous workloads served by memory-optimized engines, it is vital for them to employ effective and robust CC schemes.

We observe that the CC schemes currently in vogue with memory-optimized system are not robust under contention, particularly when short write-intensive transactions coexist with longer read-mostly transactions. For example, the two main families of approaches can be loosely classified as two-phase locking (2PL) and optimistic concurrency control (OCC). 2PL is common in traditional disk-oriented systems, and is often criticized because of high overheads, its policy of blocking transactions (leading to deadlocks and other scheduling problems), and a tendency to "lock up" (performance crash) once the aggregate transactional footprint grows too large, a state quickly attained when heavy read-mostly transactions enter the system. OCC, on the other hand, never blocks readers—and may not even block writers—thus avoiding most scheduling issues. Although they differ in details, the rising generation of memory-optimized systems almost universally adopts a form of OCC that is effectively single-versioned, with read footprint validation at pre-commit. Two systems that characteristically employ this type of OCC are

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2013.
7th Biennial Conference on Innovative Data Systems Research (CIDR'15) January 4-7, 2015, Asilomar, California, USA.
.

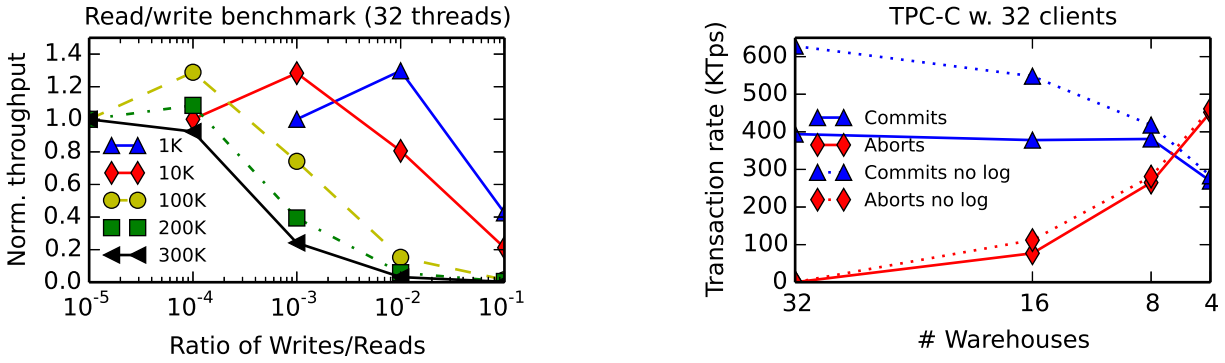

Figure 1: Performance of a memory-efficient OLTP engine with lightweight optimistic concurrency control, as the ratio of writes increases (left); and as the size of the database decreases (right).

Microsoft's Hekaton [10] and Silo [18]. This type of approach suffers badly in highly concurrent workloads because transactions must abort if any portion of their read footprint is overwritten before they commit. In Figure 1 we demonstrate how the performance of Silo, a representative of the camp of transaction processing engines with lightweight OCC, degrades as transactions have larger read footprint or when contention increases. (Section **??** has details about the experimental setup.) Figure 1(left) shows that it just takes 0.1% or 1% of the touched records to be updates for the transaction throughput to drastically drop. While Figure 1(right) shows that the abort rate grows quickly as the same number of threads operate on smaller TPC-C databases, thereby on higher contention.

Interactions at the physical level. But it is not only the interaction at the logical level that should be central to the design of a memory-optimized transaction processing engine. As commodity server hardware becomes increasingly parallel [1] many of the low-level issues (latching, thread scheduling, etc..) and design decisions—at the architecture level—need to be revisited. The form of logging used, the storage management architecture, and scheduling policies for worker threads can impose drastic constraints on which forms of CC can be implemented at all, let alone efficiently. It can be difficult or impossible to adopt a different CC scheme without significant changes to the rest of the system. For example, it was reported in [14] that the implementation effort required to add support for serializable snapshot isolation (SSI) in Postgres was very high, due to a lack of supporting infrastructure in the system. The point is not that such design choices should be avoided, but rather that they should be made only with a full awareness of the consequences for concurrency control.

Physical partitioning. Some systems sidestep the issues of logical and physical contention entirely—along with the accompanying implementation complexity—by adopting physical partitioning and a single-threaded transaction execution model [8, 9]. This execution model introduces a different set problems for mixed workloads and for workloads that are inherently difficult to partition. Given the developments in scaling-out the performance of distributed OLTP systems, especially for easy-to-partition workloads, e.g. [2], as well as for high availability and cost-effectiveness reasons, we predict that the successful architectures will combine scale-out solutions build on top of non-partitioning-based scale-up engines within each node. Therefore, we focus on the performance of non-partitioning-based engines.

Organization. In Section 2 we lay out the design principles that we believe are critical for transaction processing engines in the environment of highly-parallel servers with ample main memory. Next, on Section 3, we present a memory-optimized transaction processing architecture that carefully combines several techniques—epoch-based resource management, indirection arrays [16], anti-caching [3], and a specially-designed log manager—to enable robust CC, scalable thread interactions, and easy recovery. Section 4 underlines some weaknesses of a representative of the new breed of memory-optimized shared-everything transaction processing systems; and Section 6 concludes.

2. DESIGN DIRECTIONS

In this section we briefly discuss desired properties of transaction processing system architectures. We primarily focus on three areas: the concurrency control mechanism that determines the interaction between concurrent transactions at the logical level; the mechanism that controls the interaction/communication of threads at the physical level; and recovery. As we already argued in Section 1, we are aiming for a scalable single-node design that achieves scale-up with minimal physical partitioning.

Append-only storage: Append-only storage allows drastic simplifications of both I/O patterns and corner cases in the code. When combined with a carefully designed log manager and indirection arrays (below), we are able to achieve a single-copy system where the log is a significant fraction of the database (with records graduating from the log to secondary storage only if they go a long time with no updates). The resulting system is also easier to recover, as both undo and redo are largely unnecessary (the log can be truncated at the first hole without losing any committed work).

Logging: System designers should also treat recovery as a first class citizen when designing a transaction processing system. Log managers are a well-known source of complexity and bottlenecks in a database system. Many of the recent

[1] Note that the upcoming generation of Intel server-grade processor, Haswell-EP, comes with up to 18 cores (and 36 hyperthreads) per socket.

proposals do not provide thorough design for recovery, at least not even close to the level of detail of [12]. For example, Silo's current code base does not even implement recovery.

The in-memory log buffer becomes a central point of communication that simplifies a variety of other tasks in the system, but which tends to kill scalability. Past work has attempted to optimize [7] or distribute [19] the log, with partial success and significant additional complexity. Systems such as H-Store [8] (and its commercial version, VoltDB) largely dispense with logging and rely instead on replication. These systems replace the log bottleneck with a transaction dispatch bottleneck. Silo avoids the logging bottleneck by giving up the traditional total order in transactions. Avoiding total ordering is efficient but prevents the system from using any but the simplest of CC schemes—there is no practical way to implement repeatable read or snapshot isolation, for example, and transactions *cannot see their own writes*.

We advocate a sweet spot between the extremes of fully coordinated logging (multiple synchronization points per transaction) and fully uncoordinated logging (no synchronization at all). A transaction with a reasonably small write footprint should expect to acquire a totally ordered commit timestamp, and reserve all needed space in the log, using a single global atomic operation. Such a system can scale to a few *millions* of commits per second [18]—the current world record in TPC-E is not quite 9ktps—while still preserving ordering information that enables advanced concurrency control schemes. We present just such a log manager in Section 3.

Concurrency control: Broadly speaking, there are two camps of CC methods: the pessimistic, e.g. two-phase locking (2PL), and the optimistic (OCC). As it has been shown in the past, e.g. [1], pessimistic methods theoretically beat optimistic ones under contention, assuming those pessimistic methods can be implemented with sufficiently low overhead relative to their optimistic counterparts. However, in practice this is not easy to achieve. For example, a study of the SHORE storage manager reports roughly 25% overhead for locking-based pessimistic methods [5]; that means that any OCC scheme with lower than 25% abort rate would outperform it.

Having said that, typical memory-optimized engines that employ lightweight OCC and running on modern commodity servers, leave some room for exploration. First, a contentious workload can easily drive abort rates to 25% or higher, opening the door for more heavyweight schemes. Second, modern systems achieve extraordinarily high throughput for short-running, partition-able transactions with predictable footprints), and it might be worth trading 25% of peak performance for more robust behavior in a wider spectrum of transactional and mixed workloads. In other words, it is easier to tolerate losing 15-20% of peak performance if net throughput is still several hundreds of thousands of transactions per second.

There are different flavor of optimistic, or opportunistic, CC. Many recent systems adopt a lightweight validation step at the end of the transaction, during pre-commit. Such kind of validation is very opportunistic, and somewhat brittle, leaving the system vulnerable in many workloads. Further, these optimistic schemes make no guarantee that it is safe to retry the transaction immediately after pre-commit fails, an issue that performance evaluations often sidestep by not retrying failed transactions at all. Ideally, a good CC scheme would provide a "safe restart" property [14] where the system guarantees that a transaction will not fail twice because of the same conflict. Regardless optimistic or pessimistic, the CC mechanism should not only have a low false positive rate detecting conflicts, but it should allow the system to detect the glaring conflict cases (cases where a transaction is destined to fail) as early as possible to minimize the amount of wasted work.

At least to our knowledge, there is no (publicly available) system that is both fast enough and has the appropriate infrastructure to support the implementation of robust CC schemes. The infrastructure matters terribly. It decides whether it is even possible to implement a particular CC scheme, and whether that would be practical. For example, the effort to enhance Postgres with serializable snapshot isolation (SSI) required a very large implementation effort, since the team had to integrate what it is essentially a lock manager [2]. And even then, the achieved performance was not impressive, due to severe bottlenecks in multiple parts of the system (the new lock manager and the existing log manager, in particular). Many of the design decisions we outline in Section 3 were specifically taken in order to allow more flexibility in implementing multiple CC schemes.

Physical layer: The interactions at the physical level are typically handled by a low-level component of any transaction processing system called the storage manager. The implementation of the storage manager is tightly-coupled to the CC scheme and that makes it difficult to modify or extend the CC of legacy systems.

In our opinion, one promising technique that provides desirable properties is the indirection array [4, 16]. For example, an indirection array reduces the amount of logging required for updates in append-only systems. In a record update only the corresponding entry in the array needs to be updated; without the indirection, creating a new version requires updating every reference to that record, which can be very expensive with tree-based secondary indexes update cascades to the root of every index that has a reference to the changed object, and result cascading updates that may propagate up to the root. Additionally, on record updates the system does not have to update every reference to that record, say from secondary indexes.

As a comparison, Silo does not employ indirection arrays but instead it performs in place updates; there is effectively a single committed version of an object with perhaps a private uncommitted copy, unless a special heavyweight snapshot mechanism is invoked. Hekaton is technically multi-versioned, but because of the CC scheme it uses, versions become unusable to non read-only transactions as soon as any overwrite commits. For all practical purposes, both systems are multi-versioned only for read-only transactions.

Indirection arrays are suitable for the physical implementation of CC for multi-versioned systems, as a single compare-and-swap (CAS) operation suffices to install a new version of an object. With indirection arrays even the anti-caching technique [3] is largely simplified. That is, in the vanilla anti-caching algorithm all the secondary indexes have to be updated, whenever a record is evicted from memory—with some considerable overhead. The indirection array gives a

[2] Once all this groundwork was done, extending Postgres to other CCs is relatively easy.

convenient place to replace an in-memory pointer with an on-disk pointer, and in some sense acts as a lightweight buffer pool.

There are also some low-level reasons, detailed in Section 3.4, for using indirection arrays. For example, space management becomes easier, and they admit a convenient analog to cache-friendly compact index structures such as CSB+Trees [15]. These benefits do not justify adopting indirection, but they become convenient to use once the indirection layer is in place for other reasons.

The flexibility in implementing various CC schemes is greatly enhanced if we can establish total ordering. One way to achieve total ordering, is through a centralized log, which can be used for establishing the transaction commit order. Therefore we opt using a centralized log, but we minimize the interaction with it to only a single communication per transaction. In contrast, Silo employs a epoch-based ordering that is only partial.

3. MEMORY-OPTIMIZED OLTP WITH ROBUST CONCURENCY CONTROL

In tis section we describe in more detail several key pieces of the system we are developing, with a focus on why we choose the design trade-offs we do.

3.1 Log manager

The log manager is a pivotal piece of most database engines. It provides—if anything does—a centralized point of coordination that other pieces of the system build off of and depend on. We have developed a logging scheme that generates a commit LSN for the transaction and reserves buffer space for the transaction's log records with a single global atomic operation. Achieving this required two key insights: first, the transaction can combine its log records into large blocks, avoiding the redundancy of writing individual log record headers and reducing the number of trips to the log. Second, the LSN space need not be contiguous as long as we can still convert easily from LSN to disk address and back.

The first property arises from our use of append-only storage. We achieve the second property by assigning each LSN to a "segment" and storing its segment number in the low order bits; the LSN's position on disk can be determined by looking up, and subtracting off, its segment's starting offset (read-only). Transactions race to "open" a new segment if they obtain an LSN past the end of the current one, and unlucky transactions holding a LSN in the gap between two segments simply discard it and request a new one. Segments can be 100GB or more in size, however, so overflows are quite rare. Once an LSN and segment have been assigned, the transaction verifies availability of space in the log's circular memory buffer (again, read-only); only in case the buffer is full will transactions have to block pending space, but disk arrays readily absorb the sequential write-only I/O stream.

An additional feature of the log is that transactions acquire a commit LSN before entering pre-commit, allowing validation of multiple transactions to proceed smoothly in parallel; depending on the outcome of pre-commit, a transaction either writes its log entries or a skip record aborts to the reserved log block.

Finally, because the shared counter is implemented as a wait-free linked list, the transaction can notify the log writer that its block is ready to flush by simply flagging its node as "dead" (a blind store). The log writer periodically scans the list and writes out all log blocks that precede the oldest "live" buffer allocation; transactions do not touch each others' nodes, and the log writer only reads them and flags dead nodes for the garbage collector.

3.2 Epoch-based resource management

We have developed a lightweight epoch management system that can track multiple timelines (of differing granularities) in parallel. A multi-transaction-scale epoch manager implements garbage collection of dead versions and deleted records, a medium-scale epoch implements a read-copy-update (RCU) mechanism that manages physical memory and data structure usage [**?**], and a very short timescale epoch manager tracks transaction IDS, which we recycle aggressively (see below).

The key to efficiency here is to avoid flagging stragglers unless it is absolutely necessary (because coordinating with non-responsive threads is very expensive). To avoid this, the system does not attempt to reclaim resource for epoch N until epoch $N+2$ begins. This way, potential stragglers have all of epoch $N+1$ to quiesce without penalty; however, epoch $N+3$ cannot begin until the last straggler from epoch N completes. This four-phase scheme communicates far less with stragglers than the traditional two-phase approach while maintaining the same worst-case timing bound. It allows us to track epochs at a very fine granularity when necessary.

3.3 Transaction management

Each transactions in the system is assigned a slot in a global transaction state table when it begins. This fixed-size table holds its begin time (which is the log's end LSN at the time it started), status, and end time (if applicable). Transaction ids are a combination of table offset and epoch, with an epoch manager to prevent entries from being recycled too soon. Update transactions write their XID into each version they create, change their status to pre-commit, acquire a commit LSN (or are given one by an impatient peer), and finally commit atomically by changing their status to "committed." A post-commit cleanup step involves replacing XID stamps with the transaction's commit LSN, at which point the state table entry is no longer needed can be recycled by the epoch manager. Other transactions that encounter an XID in a version can reliably verify its commit status and age by visiting the XID table, and—if necessary—will help a peer enter pre-commit by acquiring a log block on its behalf.

3.4 Indirection arrays

The indirection arrays used in ERMIA are very similar to the ones proposed in the literature. In short, all logical objects are identified by an object ID (OID) that maps to a slot in an OID array that contains the physical pointer. The pointer may reference disk, or a chain of versions stored in memory. As with Hekaton, uncommitted versions are never written to disk, but unlike Hekaton, we dispense with delta records (too expensive to apply) and use pure copy-on-write. New versions can be installed by an atomic compare-and-swap operation, and an uncommitted record at the head of the chain constitutes a write lock for CC schemes that care to track W-W conflicts (as most do).

3.5 Concurrency control

The system has been designed from the ground up to allow efficient implementations of a variety of concurrency control mechanisms. It can use read set validation (like SILO and Hekaton), but can also provide snapshot isolation to writers and even efficient implementations of serializable snapshot isolation that have been proposed recently [?]. We are also in the process of developing new CC schemes which promise lower abort rates than SSI with lower overhead and reduced implementation complexity. The other components in the system work together to make efficent CC possible: efficient (but optional) multi-versioning allowed by the indirection arrays, the total commit ordering afforded by the log, and the ability to determine easily the age of a version thanks to the transaction manager. The first (MVCC via indirection array) is available in other systems, but the other two are key enablers of the new CC schemes we are developing.

3.6 Recovery

Recovery is straightforward because the log contains only committed work. OID arrays are the only real source of complexity, as they are volatile in-memory structures that make it possible to find all other objects in the system. It turns out that logical objects (records) are physically logged, while physical data (allocator state and OID array contents) use logical logging. OID arrays are themselves objects stored in a master OID array, but they are updated in place to avoid overloading the log, with changes replayed by a log analysis step that reads only log block headers. This analysis step is very fast, because the skipped-over log payloads account for 90% or more of the total log. In order to support efficient recovery, system transactions occasionally checkpoint the OID arrays using a fuzzy checkpointing mechanism to minimize the impact on user transactions. Because the log is the database, recovery only needs to rebuild the OID arrays in memory; anti-caching will take care of loading the actual data, though background pre-loading is highly recommended to minimize cold start effects.

4. EVALUATION

We want to show that the recent proposals for main-memory OLTP solutions with light-weight optimistic concurrency control schemes, such as Silo, can perform well and are suitable for only on a limited subset of transactional applications, primarily due to limitations imposed by their (tightly-integrated) concurrency control mechanism. We are not attempting a detailed evaluation, as this would be more suitable for a longer document.

We use a 4-socket server with 6-core Intel Xeon E7-4807 processors, for a total of 24 physical cores and 48 hyper-threads. The machine has 64GB of RAM. For our comparison we use Silo. Silo's SLAB allocator is given 32GB of memory and its logging is disabled. We use the TPC-C database with 32 Warehouses (scaling factor of 32). We use the Stock table for the transactions. The Stock table has 100K*Warehouses (3.2M) records. Each transaction randomly peaks a number of records to read and a smaller fraction of records to update. We run each experiment for 60secs.

Figure 1 (left) shows the performance of Silo when we increase the ratio of writes in transactions of increasingly larger footprint. It just takes 0.1% or 1% of writes for the performance to drop by almost an order of magnitude. Figure 1 (right) shows the performance of Silo when 32 clients run the TPC-C benchmark in an increasingly smaller database. At 4 Warehouses the abort increases almost exponentially.

Figure 2 shows the performance of Silo as the number of concurrent threads increases, for an increasing ratio of writes, when each transaction performs 100K random reads. The noticeable trend is the quickly increasing rate of aborts with the number of concurrent threads. Things get even worse as we increase the read footprint of the transactions. For example, Figure 3 shows the performance of Silo as the number of concurrent threads increases, for an increasing ratio of writes, when each transaction is three times larger, performing 300K random reads.

5. RELATED WORK

In terms of concurrency control, one of the most important studies has been [1]. This modeling study shows that if the overhead of pessimistic two-phase locking can be comparable to the overhead of optimistic methods then the pessimistic one is superior. The same study shows that it is beneficial to abort transactions that are going to abort as soon as possible. That is corroborated by other studies as well, e.g. [14]. We follow the findings, trying to detect conflicts early.

Many of the memory-optimized systems adopt lightweight optimistic concurrency control schemes that are suitable only for a small fraction of transactional workloads. The designs can be categorized in three categories: non-partitioning- and partitioning-based systems and clustered solutions. Silo's [18] employs a light-weight optimistic concurrency control that performs validations at pre-commit. That, as we showed in Section 4, performs well only in a limited set of workloads. Microsoft Hekaton [4] employs similar multi-versioning CC [10]. It is worth mentioning that Hekaton also uses a technique similar to the indirection map, which we also use.

H-Store (and its commercial version, VoltDB) is a characteristic partitioning-based system [8]. H-Store physically partitions each database to as many instances as the number of available processors, and each processor executes each transaction in serial order without interruption. Problems raise when the system has to execute mutli-site transactions, transactions that touch data from two or more separate database instances. Hyper [9] follows H-Store's single-threaded execution principle. To scale up to multi-cores they employ the hardware transactional memory capabilities of the latest generation of processors [11]. DORA [13] employs logical partitioning that avoids limitations of physically partitioned systems. But DORA uses Shore-MT's codebase [6], which is a scalable but disk-optimized storage manager with significantly bloated codebase. Hence, its performance lacks in comparison with the memory-optimized proposals.

6. CONCLUSION

In this paper we underlined the weaknesses of recent MVCC-based transaction processing system proposals, and presented a novel system with much more robust performance due to its concurrency control. The point we want to make is that concurrency control is a fundamental component of any transaction processing system, and that it cannot be baked in after the fact to system. Instead, transaction processing system designers should think of concurrency control from the beginning as the decision about the concurrency control mechanism will dictate most of the design decision of all the other components of the systems.

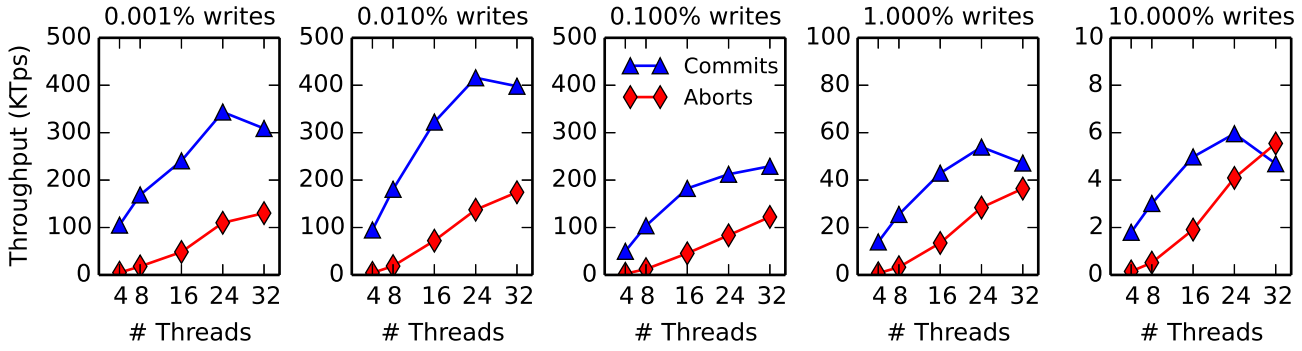

Figure 2: Rate of committed and aborted transactions for Silo as the number of concurrent threads increases, for an increasing ratio of writes. Each transaction performs 100K random reads.

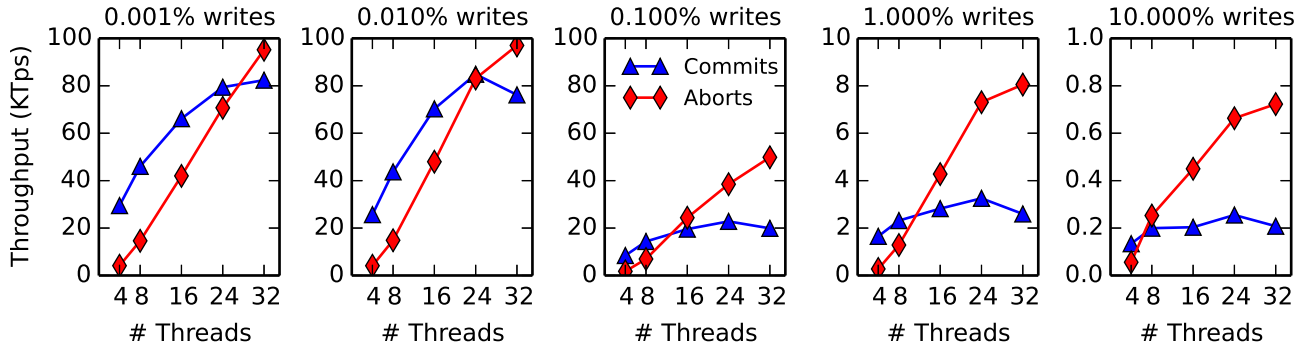

Figure 3: Rate of committed and aborted transactions for Silo as the number of concurrent threads increases, for an increasing ratio of writes. Each transaction performs 300K random reads.

References

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM TODS*, 12(4):609–654, 1987.

[2] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with ramp transactions. In *SIGMOD*, 2014.

[3] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *PLVDB*, 2013.

[4] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, 2013.

[5] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[6] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, 2009.

[7] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3, 2010.

[8] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 2008.

[9] A. Kemper and T. Neumann. HyPer – a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

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

[11] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

[12] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.

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

[14] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PLVDB*, 5(12), 2012.

[15] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.

[16] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making updates disk-I/O friendly using SSDs. *PVLDB*, 6, 2013.

[17] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP benchmarks - the obsolete, the ubiquitous, the unexplored. In *EDBT*, 2013.

[18] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[19] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 2014.