An Empirical Evaluation of In-Memory Multi-Version Concurrency Control

Yingjun Wu
National University of Singapore
yingjun@comp.nus.edu.sg

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Jiexi Lin
Carnegie Mellon University
jiexil@cs.cmu.edu

Ran Xian
Carnegie Mellon University
rxian@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

Multi-version concurrency control (MVCC) is currently the most popular transaction management scheme in modern database management systems (DBMSs). Although MVCC was discovered in the late 1970s, it is used in almost every major relational DBMS released in the last decade. Maintaining multiple versions of data potentially increases parallelism without sacrificing serializability when processing transactions. But scaling MVCC in a multi-core and in-memory setting is non-trivial: when there are a large number of threads running in parallel, the synchronization overhead can outweigh the benefits of multi-versioning.

To understand how MVCC perform when processing transactions in modern hardware settings, we conduct an extensive study of the scheme's four key design decisions: concurrency control protocol, version storage, garbage collection, and index management. We implemented state-of-the-art variants of all of these in an in-memory DBMS and evaluated them using OLTP workloads. Our analysis identifies the fundamental bottlenecks of each design choice.

1. INTRODUCTION

Computer architecture advancements has led to the rise of multi-core, in-memory DBMSs that employ efficient transaction management mechanisms to maximize parallelism without sacrificing serializability. The most popular scheme used in DBMSs developed in the last decade is *multi-version concurrency control* (MVCC). The basic idea of MVCC is that the DBMS maintains multiple physical versions of each logical object in the database to allow operations on the same object to proceed in parallel. These objects can be at any granularity, but almost every MVCC DBMS uses tuples because it provides a good balance between parallelism versus the overhead of version tracking. Multi-versioning allows read-only transactions to access older versions of tuples without preventing read-write transactions from simultaneously generating newer versions. Contrast this with a single-version system where transactions always overwrite a tuple with new information whenever they update it.

What is interesting about this trend of recent DBMSs using MVCC is that the scheme is not new. The first mention of it appeared in a 1979 dissertation [38] and the first implementation started in 1981 [22] for the InterBase DBMS (now open-sourced as Firebird). MVCC is also used in some of the most widely deployed disk-oriented DBMSs today, including Oracle (since 1984 [4]), Postgres (since 1985 [41]), and MySQL's InnoDB engine (since 2001). But while there are plenty of contemporaries to these older systems that use a single-version scheme (e.g., IBM DB2, Sybase), almost every new transactional DBMS eschews this approach in favor of MVCC [37]. This includes both commercial (e.g., Microsoft Hekaton [16], SAP HANA [40], MemSQL [1], NuoDB [3]) and academic (e.g., HYRISE [21], HyPer [36]) systems.

Despite all these newer systems using MVCC, there is no one "standard" implementation. There are several design choices that have different trade-offs and performance behaviors. Until now, there has not been a comprehensive evaluation of MVCC in a modern DBMS operating environment. The last extensive study was in the 1980s [13], but it used simulated workloads running in a disk-oriented DBMS with a single CPU core. The design choices of legacy disk-oriented DBMSs are inappropriate for in-memory DBMSs running on a machine with a large number of CPU cores. As such, this previous work does not reflect recent trends in latch-free [27] and serializable [20] concurrency control, as well as in-memory storage [36] and hybrid workloads [40].

In this paper, we perform such a study for key transaction management design decisions in of MVCC DBMSs: (1) concurrency control protocol, (2) version storage, (3) garbage collection, and (4) index management. For each of these topics, we describe the state-of-the-art implementations for in-memory DBMSs and discuss their trade-offs. We also highlight the issues that prevent them from scaling to support larger thread counts and more complex workloads. As part of this investigation, we implemented all of the approaches in the **Peloton** [5] in-memory MVCC DBMS. This provides us with a uniform platform to compare implementations that is not encumbered by other architecture facets. We deployed Peloton on a machine with 40 cores and evaluate it using two OLTP benchmarks. Our analysis identifies the scenarios that stress the implementations and discuss ways to mitigate them (if it all possible).

2. BACKGROUND

We first provide an overview of the high-level concepts of MVCC. We then discuss the meta-data that the DBMS uses to track transactions and maintain versioning information.

2.1 MVCC Overview

A transaction management scheme permits end-users to access a database in a multi-programmed fashion while preserving the illu-

Proceedings of the VLDB Endowment, Vol. 10, No. 7

Table 1: MVCC Implementations – A summary of the design decisions made for the commercial and research MVCC DBMSs. The year attribute for each system (except for Oracle) is when it was first released or announced. For Oracle, it is the first year the system included MVCC. With the exception of Oracle, MySQL, and Postgres, all of the systems assume that the primary storage location of the database is in memory.

	Year	Protocol	Version Storage	Garbage Collection	Index Management
Oracle [4]	1984	MV2PL	Delta	Tuple-level (VAC)	Logical Pointers (TupleId)
Postgres [6]	1985	MV2PL/SSI	Append-only (O2N)	Tuple-level (VAC)	Physical Pointers
MySQL-InnoDB [2]	2001	MV2PL	Delta	Tuple-level (VAC)	Logical Pointers (PKey)
HYRISE [21]	2010	MVOCC	Append-only (N2O)	–	Physical Pointers
Hekaton [16]	2011	MVOCC	Append-only (O2N)	Tuple-level (COOP)	Physical Pointers
MemSQL [1]	2012	MVOCC	Append-only (N2O)	Tuple-level (VAC)	Physical Pointers
SAP HANA [28]	2012	MV2PL	Time-travel	Hybrid	Logical Pointers (TupleId)
NuoDB [3]	2013	MV2PL	Append-only (N2O)	Tuple-level (VAC)	Logical Pointers (PKey)
HyPer [36]	2015	MVOCC	Delta	Transaction-level	Logical Pointers (TupleId)

sion that each of them is executing alone on a dedicated system [9]. It ensures the atomicity and isolation guarantees of the DBMS.

There are several advantages of a multi-version system that are relevant to modern database applications. Foremost is that it can potentially allow for greater concurrency than a single-version system. For example, a MVCC DBMS allows a transaction to read an older version of an object at the same time that another transaction updates that same object. This is important in that execute read-only queries on the database at the same time that read-write transactions continue to update it. If the DBMS never removes old versions, then the system can also support "time-travel" operations that allow an application to query a consistent snapshot of the database as it existed at some point of time in the past [8].

The above benefits have made MVCC the most popular choice for new DBMS implemented in recent years. Table 1 provides a summary of the MVCC implementations from the last three decades. But there are different ways to implement multi-versioning in a DBMS that each creates additional computation and storage overhead. These design decisions are also highly dependent on each other. Thus, it is non-trivial to discern which ones are better than others and why. This is especially true for in-memory DBMSs where disk is no longer the main bottleneck.

In the following sections, we discuss the implementation issues and performance trade-offs of these design decisions. We then perform a comprehensive evaluation of them in Sect. 7. We note that we only consider serializable transaction execution in this paper. Although logging and recovery is another important aspect of a DBMS's architecture, we exclude it from our study because there is nothing about it that is different from a single-version system and in-memory DBMS logging is already covered elsewhere [33, 49].

2.2 DBMS Meta-Data

Regardless of its implementation, there is common meta-data that a MVCC DBMS maintains for transactions and database tuples.

Transactions: The DBMS assigns a transaction T a unique, monotonically increasing timestamp as its identifier (T_{id}) when they first enter the system. The concurrency control protocols use this identifier to mark the tuple versions that a transaction accesses. Some protocols also use it for the serialization order of transactions.

Tuples: As shown in Fig. 1, each physical version contains four meta-data fields in its header that the DBMS uses to coordinate the execution of concurrent transactions (some of the concurrency control protocols discussed in the next section include additional fields). The `txn-id` field serves as the version's write lock. Every tuple has this field set to zero when the tuple is not write-locked. Most DBMSs use a 64-bit `txn-id` so that it can use a single compare-and-swap (CaS) instruction to atomically update the value. If a transaction T with identifier T_{id} wants to update a tuple `A`, then the DBMS checks whether `A`'s `txn-id` field is zero. If it is, then DBMS will set the value of `txn-id` to T_{id} using a CaS instruction [27, 44].

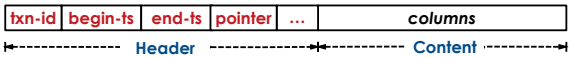


Figure 1: Tuple Format – The basic layout of a physical version of a tuple.

Any transaction that attempts to update `A` is aborted if this `txn-id` field is neither zero or not equal to its T_{id}. The next two meta-data fields are the `begin-ts` and `end-ts` timestamps that represent the lifetime of the tuple version. Both fields are initially set to zero. The DBMS sets a tuple's `begin-ts` to `INF` when the transaction deletes it. The last meta-data field is the `pointer` that stores the address of the neighboring (previous or next) version (if any).

3. CONCURRENCY CONTROL PROTOCOL

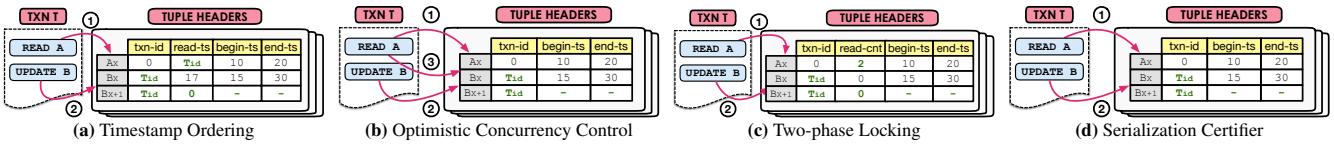
Every DBMS includes a *concurrency control protocol* that coordinates the execution of concurrent transactions [11]. This protocol determines (1) whether to allow a transaction to access or modify a particular tuple version in the database at runtime, and (2) whether to allow a transaction to commit its modifications. Although the fundamentals of these protocols remain unchanged since the 1980s, their performance characteristics have changed drastically in a multi-core and main-memory setting due to the absence of disk operations [42]. As such, there are newer high-performance variants that remove locks/latches and centralized data structures, and are optimized for byte-addressable storage.

In this section, we describe the four core concurrency control protocols for MVCC DBMSs. We only consider protocols that use tuple-level locking as this is sufficient to ensure serializable execution. We omit range queries because multi-versioning does not bring any benefits to phantom prevention [17]. Existing approaches to provide serializable transaction processing use either (1) additional latches in the index [35, 44] or (2) extra validation steps when transactions commit [27].

3.1 Timestamp Ordering (MVTO)

The MVTO algorithm from 1979 is considered to be the original multi-version concurrency control protocol [38, 39]. The crux of this approach is to use the transactions' identifiers (T_{id}) to precompute their serialization order. In addition to the fields described in Sect. 2.2, the version headers also contain the identifier of the last transaction that read it (`read-ts`). The DBMS aborts a transaction that attempts to read or update a version whose write lock is held by another transaction.

When transaction T invokes a read operation on logical tuple `A`, the DBMS searches for a physical version where T_{id} is in between the range of the `begin-ts` and `end-ts` fields. As shown in Fig. 2a, T is allowed to read version A_x if its write lock is not held by another active transaction (i.e., value of `txn-id` is zero or equal to T_{id}) because MVTO never allows a transaction to read uncommitted versions. Upon reading A_x, the DBMS sets A_x's `read-ts` field to T_{id} if its current value is less than T_{id}. Otherwise, the transaction reads an older version without updating this field.



(a) Timestamp Ordering **(b)** Optimistic Concurrency Control **(c)** Two-phase Locking **(d)** Serialization Certifier

Figure 2: Concurrency Control Protocols – Examples of how the protocols process a transaction that executes a READ followed by an UPDATE.

With MVTO, a transaction always updates the latest version of a tuple. Transaction T creates a new version B_{x+1} if (1) no active transaction holds B_x's write lock and (2) T_{id} is larger than B_x's `read-ts` field. If these conditions are satisfied, then the DBMS creates a new version B_{x+1} and sets its `txn-id` to T_{id}. When T commits, the DBMS sets B_{x+1}'s `begin-ts` and `end-ts` fields to T_{id} and `INF` (respectively), and B_x's `end-ts` field to T_{id}.

3.2 Optimistic Concurrency Control (MVOCC)

The next protocol is based on the optimistic concurrency control (OCC) scheme proposed in 1981 [26]. The motivation behind OCC is that the DBMS assumes that transactions are unlikely to conflict, and thus a transaction does not have to acquire locks on tuples when it reads or updates them. This reduces the amount of time that a transaction holds locks. There are changes to the original OCC protocol to adapt it for multi-versioning [27]. Foremost is that the DBMS does not maintain a private workspace for transactions, since the tuples' versioning information already prevents transactions from reading or updating versions that should not be visible to them.

The MVOCC protocol splits a transaction into three phases. When the transaction starts, it is in the *read phase*. This is where the transaction invokes read and update operations on the database. Like MVTO, to perform a read operation on a tuple `A`, the DBMS first searches for a visible version A_x based on `begin-ts` and `end-ts` fields. T is allowed to update version A_x if its write lock is not acquired. In a multi-version setting, if the transaction updates version B_x, then the DBMS creates version B_{x+1} with its `txn-id` set to T_{id}.

When a transaction instructs the DBMS that it wants to commit, it then enters the *validation phase*. First, the DBMS assigns the transaction another timestamp (T_{commit}) to determine the serialization order of transactions. The DBMS then determines whether the tuples in the transaction's read set was updated by a transaction that already committed. If the transaction passes these checks, it then enters the *write phase* where the DBMS installs all the new versions and sets their `begin-ts` to T_{commit} and `end-ts` to `INF`.

Transactions can only update the latest version of a tuple. But a transaction cannot read a new version until the other transaction that created it commits. A transaction that reads an outdated version will only find out that it should abort in the validation phase.

3.3 Two-phase Locking (MV2PL)

This protocol uses the two-phase locking (2PL) method [11] to guarantee the transaction serializability. Every transaction acquires the proper lock on the current version of logical tuple before it is allowed to read or modify it. In a disk-based DBMS, locks are stored separately from tuples so that they are never swapped to disk. This separation is unnecessary in an in-memory DBMS, thus with MV2PL the locks are embedded in the tuple headers. The tuple's *write lock* is the `txn-id` field. For the *read lock*, the DBMS uses a `read-cnt` field to count the number of active transactions that have read the tuple. Although it is not necessary, the DBMS can pack `txn-id` and `read-cnt` into contiguous 64-bit word so that the DBMS can use a single CaS to update them at the same time.

To perform a read operation on a tuple `A`, the DBMS searches for a visible version by comparing a transaction's T_{id} with the tuples' `begin-ts` field. If it finds a valid version, then the DBMS increments that tuple's `read-cnt` field if its `txn-id` field is equal to zero (meaning that no other transaction holds the write lock). Similarly, a transaction is allowed to update a version B_x only if both `read-cnt` and `txn-id` are set to zero. When a transaction commits, the DBMS assigns it a unique timestamp (T_{commit}) that is used to update the `begin-ts` field for the versions created by that transaction and then releases all of the transaction's locks.

The key difference among 2PL protocols is in how they handle deadlocks. Previous research has shown that the *no-wait* policy [9] is the most scalable deadlock prevention technique [48]. With this, the DBMS immediately aborts a transaction if it is unable to acquire a lock on a tuple (as opposed to waiting to see whether the lock is released). Since transactions never wait, the DBMS does not have to employ a background thread to detect and break deadlocks.

3.4 Serialization Certifier

In this last protocol, the DBMS maintains a serialization graph for detecting and removing "dangerous structures" formed by concurrent transactions [12, 20, 45]. One can employ certifier-based approaches on top of weaker isolation levels that offer better performance but allow certain anomalies.

The first certifier proposed was serializable snapshot isolation (SSI) [12]; this approach guarantees serializability by avoiding write-skew anomalies for snapshot isolation. SSI uses a transaction's identifier to search for a visible version of a tuple. A transaction can update a version only if the tuple's `txn-id` field is set to zero. To ensure serializability, the DBMS tracks *anti-dependency* edges in its internal graph; these occur when a transaction creates a new version of a tuple where its previous version was read by another transaction. The DBMS maintains flags for each transaction that keeps track of the number of in-bound and out-bound anti-dependency edges. When the DBMS detects two consecutive anti-dependency edges between transactions, it aborts one of them.

The serial safety net (SSN) is a newer certifier-based protocol [45]. Unlike with SSI, which is only applicable to snapshot isolation, SSN works with any isolation level that is at least as strong as `READ COMMITTED`. It also uses a more more precise anomaly detection mechanism that reduces the number of unnecessary aborts. SSN encodes the transaction dependency information into metadata fields and validates a transaction T's consistency by computing a low watermark that summarizes "dangerous" transactions that committed before the T but must be serialized after T [45]. Reducing the number of false aborts makes SSN more amenable to workloads with read-only or read-mostly transactions.

3.5 Discussion

These protocols handle conflicts differently, and thus are better for some workloads more than others. MV2PL records reads with its read lock for each version. Hence, a transaction performing a read/write on a tuple version will cause another transaction to abort if it attempts to do the same thing on that version. MVTO instead uses the `read-ts` field to record reads on each version. MVOCC does not update any fields on a tuple's version header during read/operations. This avoids unnecessary coordination between threads, and a transaction reading one version will not lead to an abort other transactions that update the same version. But MVOCC

requires the DBMS to examine a transaction's read set to validate the correctness of that transaction's read operations. This can cause starvation of long-running read-only transactions [24]. Certifier protocols reduce aborts because they do not validate reads, but their anti-dependency checking scheme may bring additional overheads.

There are some proposals for optimizing the above protocols to improve their efficacy for MVCC DBMSs [10, 27]. One approach is to allow a transaction to *speculatively read* uncommitted versions created by other transactions. The trade-off is that the protocols must track the transactions' read dependencies to guarantee serializable ordering. Each worker thread maintains a *dependency counter* of the number of transactions that it read their uncommitted data. A transaction is allowed to commit only when its dependency counter is zero, whereupon the DBMS traverses its dependency list and decrements the counters for all the transactions that are waiting for it to finish. Similarly, another optimization mechanism is to allow transactions to *eagerly update* versions that are read by uncommitted transactions. This optimization also requires the DBMS to maintain a centralized data structure to track the dependencies between transactions. A transaction can commit only when all of the transactions that it depends on have committed.

Both optimizations described above can reduce the number of unnecessary aborts for some workloads, but they also suffer from cascading aborts. Moreover, we find that the maintenance of a centralized data structure can become a major performance bottleneck, which prevents the DBMS from scaling towards dozens of cores.

4. VERSION STORAGE

Under MVCC, the DBMS always constructs a new physical version of a tuple when a transaction updates it. The DBMS's *version storage scheme* specifies how the system stores these versions and what information each version contains. The DBMS uses the tuples' `pointer` field to create a latch-free linked list called a *version chain*. This version chain allows the DBMS to locate the desired version of a tuple that is visible to a transaction. As we discuss below, the chain's HEAD is either the newest or oldest version.

We now describe these schemes in more detail. Our discussion focuses on the schemes' trade-offs for `UPDATE` operations because this is where the DBMS handles versioning. A DBMS inserts new tuples into a table without having to update other versions. Likewise, a DBMS deletes tuples by setting a flag in the current version's `begin-ts` field. In subsequent sections, we will discuss the implications of these storage schemes on how the DBMS performs garbage collection and how it maintains pointers in indexes.

4.1 Append-only Storage

In this first scheme, all of the tuple versions for a table are stored in the same storage space. This approach is used in Postgres, as well as in-memory DBMSs like Hekaton, NuoDB, and MemSQL. To update an existing tuple, the DBMS first acquires an empty slot from the table for the new tuple version. It then copies the content of the current version to the new version. Finally, it applies the modifications to the tuple in the newly allocated version slot.

The key decision with the append-only scheme is how the DBMS orders the tuples' version chains. Since it is not possible to maintain a latch-free doubly linked list, the version chain only points in one direction. This ordering has implications on how often the DBMS updates indexes whenever transactions modify tuples.

Oldest-to-Newest (O2N): With this ordering, the chain's HEAD is the oldest extant version of a tuple (see Fig. 3a). This version might not be visible to any active transaction but the DBMS has yet to reclaim it. The advantage of O2N is that the DBMS need not update the indexes to point to a newer version of the tuple whenever it is modified. But the DBMS potentially traverses a long version chain to find the latest version during query processing. This is slow because of pointer chasing and it pollutes CPU caches by reading unneeded versions. Thus, achieving good performance with O2N is highly dependent on the system's ability to prune old versions.

Newest-to-Oldest (N2O): The alternative is to store the newest version of the tuple as the version chain's HEAD (see Fig. 3b). Since most transactions access the latest version of a tuple, the DBMS does not have to traverse the chain. The downside, however, is that the chain's HEAD changes whenever a tuple is modified. The DBMS then updates all of the table's indexes (both primary and secondary) to point to the new version. As we discuss in Sect. 6.1, one can avoid this problem through an indirection layer that provides a single location that maps the tuple's latest version to physical address. With this setup, the indexes point to tuples' mapping entry instead of their physical locations. This works well for tables with many secondary indexes but increases the storage overhead.

Another issue with append-only storage is how to deal with non-inline attributes (e.g., BLOBs). Consider a table that has two attributes (one integer, one BLOB). When a transaction updates a tuple in this table, under the append-only scheme the DBMS creates a copy of the BLOB attributes (even if the transaction did not modify it), and then the new version will point to this copy. This is wasteful because it creates redundant copies. To avoid this problem, one optimization is to allow the multiple physical versions of the same tuple to point to the same non-inline data. The DBMS maintains reference counters for this data to ensure that values are deleted only when they are no longer referenced by any version.

4.2 Time-Travel Storage

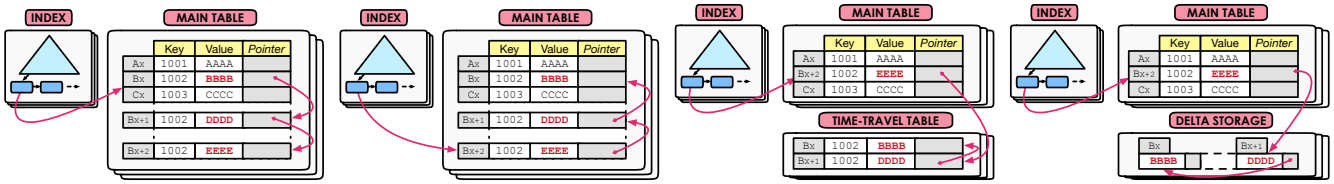
The next storage scheme is similar to the append-only approach except that the older versions are stored in a separate table. The DBMS maintain a *master* version of each tuple in the main table and multiple versions of the same tuple in a separate time-travel table. In some DBMSs, like SQL Server, the master version is the current version of the tuple. Other systems, like SAP HANA, store the oldest version of a tuple as the master version to provide snapshot isolation [29]. This incurs additional maintenance costs during GC because the DBMS copies the data from the time-travel table back to the main table when it prunes the current master version. For simplicity, we only consider the first time-travel approach where the master version is always in the main table.

To update a tuple, the DBMS first acquires a slot in the time-travel table and then copies the master version to this location. It then modifies the master version stored in the main table. Indexes are not affected by version chain updates because they always point to the master version. As such, it avoids the overhead of maintaining the database's indexes whenever a transaction updates a tuple and is ideal for queries that access the current version of a tuple.

This scheme also suffers from the same non-inline attribute problem as the append-only approach. The data sharing optimization that we describe above is applicable here as well.

4.3 Delta Storage

With this last scheme, the DBMS maintains the master versions of tuples in the main table and a sequence of *delta versions* in a separate *delta storage*. This storage is referred to as the *rollback segment* in MySQL and Oracle, and is also used in HyPer. Most existing DBMSs store the current version of a tuple in the main table. To update an existing tuple, the DBMS acquires a continuous space from the delta storage for creating a new delta version. This delta



(a) Append-only (O2N) **(b)** Append-only (N2O) **(c)** Time-travel Storage **(d)** Delta Storage

Figure 3: Version Storage – This diagram provides an overview of how the schemes organize versions in different data structures and how their pointers create version chains in an in-memory MVCC DBMS. Note that there are two variants of the append-only scheme that differ on the ordering of the version chains.

version contains the original values of modified attributes rather than the entire tuple. The DBMS then directly performs in-place update to the master version in the main table.

This scheme is ideal for `UPDATE` operations that modify a subset of a tuple's attributes because it reduces memory allocations. This approach, however, leads to higher overhead for read-intensive workloads. To perform a read operation that accesses multiple attributes of a single tuple, the DBMS has to traverse the version chain to fetch the data for each single attribute that is accessed by the operation.

4.4 Discussion

These schemes have different characteristics that affect their behavior for OLTP workloads. As such, none of them achieve optimal performance for either workload type. The append-only scheme is better for analytical queries that perform large scans because versions are stored contiguously in memory, which minimizes CPU cache misses and is ideal for hardware prefetching. But queries that access an older version of a tuple suffer from higher overhead because the DBMS follows the tuple's chain to find the proper version. The append-only scheme also exposes physical versions to the index structures, which enables additional index management options.

All of the storage schemes require the DBMS to allocate memory for each transaction from centralized data structures (i.e., tables, delta storage). Multiple threads will access and update this centralized storage at the same time, thereby causing access contention. To avoid this problem, the DBMS can maintain separate memory spaces for each centralized structure (i.e., tables, delta storage) and expand them in fixed-size increments. Each worker thread then acquires memory from a single space. This essentially partitions the database, thereby eliminating centralized contention points.

5. GARBAGE COLLECTION

Since MVCC creates new versions when transactions update tuples, the system will run out of space unless it reclaims the versions that are no longer needed. This also increases the execution time of queries because the DBMS spends more time traversing long version chains. As such, the performance of a MVCC DBMS is highly dependent on the ability of its *garbage collection* (GC) component to reclaim memory in a transactionally safe manner.

The GC process is divided into three steps: (1) detect expired versions, (2) unlink those versions from their associated chains and indexes, and (3) reclaim their storage space. The DBMS considers a version as expired if it is either an invalid version (i.e., created by an aborted transaction) or it is not visible to any active transaction. For the latter, the DBMS checks whether a version's `end-ts` is less than the T_{id} of all active transactions. The DBMS maintains a centralized data structure to track this information, but this is a scalability bottleneck in a multi-core system [27, 48].

An in-memory DBMS can avoid this problem with coarse-grained *epoch-based* memory management that tracks the versions created by transactions [44]. There is always one active epoch and an FIFO queue of prior epochs. After some amount of time, the DBMS moves the current active epoch to the prior epoch queue and then creates a new active one. This transition is performed either by a background thread or in a cooperative manner by the DBMS's worker threads. Each epoch contains a count of the number of transactions that are assigned to it. The DBMS registers each new transaction into the active epoch and increments this counter. When a transaction finishes, the DBMS removes it from its epoch (which may no longer be the current active one) and decrements this counter. If a non-active epoch's counter reaches zero and all of the previous epochs also do not contain active transactions, then it is safe for the DBMS to reclaim expired versions that were updated in this epoch.

There are two GC implementations for a MVCC that differ on how the DBMS looks for expired versions. The first approach is *tuple-level* GC wherein the DBMS examines the visibility of individual tuples. The second is *transaction-level* GC that checks whether any version created by a finished transaction is visible. One important thing to note is that not all of the GC schemes that we discuss below are compatible with every version storage scheme.

5.1 Tuple-level Garbage Collection

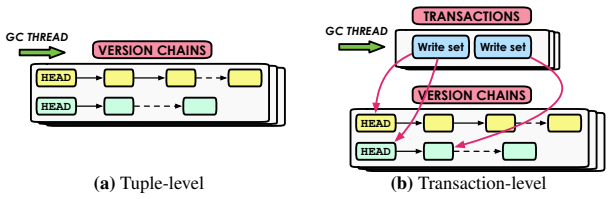
With this approach, the DBMS checks the visibility of each individual tuple version in one of two ways:

Background Vacuuming (VAC): The DBMS uses background threads that periodically scan the database for expired versions. As shown in Table 1, this is the most common approach in MVCC DBMSs as it is easier to implement and works with all version storage schemes. But this mechanism does not scale for large databases, especially with a small number of GC threads. A more scalable approach is where transactions register the invalidated versions in a latch-free data structure [27]. The GC threads then reclaim these expired versions using the epoch-based scheme described above. Another optimization is where the DBMS maintains a bitmap of dirty blocks so that the vacuum threads do not examine blocks that were not modified since the last GC pass.

Cooperative Cleaning (COOP): When executing a transaction, the DBMS traverses the version chain to locate the visible version. During this traversal, it identifies the expired versions and records them in a global data structure. This approach scales well as the GC threads no longer needs to detect expired versions, but it only works for the O2N append-only storage. One additional challenge is that if transactions do not traverse a version chain for a particular tuple, then the system will never remove its expired versions. This problem is called "dusty corners" in Hekaton [16]. The DBMS overcomes this by periodically performing a complete GC pass with a separate thread like in VAC.

5.2 Transaction-level Garbage Collection

In this GC mechanism, the DBMS reclaims storage space at transaction-level granularity. It is compatible with all of the version storage schemes. The DBMS considers a transaction as expired when the versions that it generated are not visible to any active transaction. After an epoch ends, all of the versions that were generated by the transactions belonging to that epoch can be safely



(a) Tuple-level **(b)** Transaction-level

Figure 4: Garbage Collection – Overview of how to examine the database for expired versions. The tuple-level GC scans the tables' version chains, whereas the transaction-level GC uses transactions' write-sets.

removed. This is simpler than the tuple-level GC scheme, and thus it works well with the transaction-local storage optimization (Sect. 4.4) because the DBMS reclaims a transaction's storage space all at once. The downside of this approach, however, is that the DBMS tracks the read/write sets of transactions for each epoch instead of just using the epoch's membership counter.

5.3 Discussion

Tuple-level GC with background vacuuming is the most common implementation in MVCC DBMSs. In either scheme, increasing the number of dedicated GC threads speeds up the GC process. The DBMS's performance drops in the presence of long-running transactions. This is because all the versions generated during the lifetime of such a transaction cannot be removed until it completes.

6. INDEX MANAGEMENT

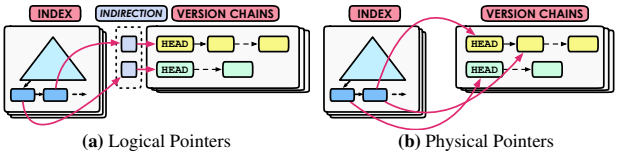
All MVCC DBMSs keep the database's versioning information separate from its indexes. That is, the existence of a key in an index means that some version exists with that key but the index entry does not contain information about which versions of the tuple match. We define an *index entry* as a key/value pair, where the *key* is a tuple's indexed attribute(s) and the *value* is a pointer to that tuple. The DBMS follows this pointer to a tuple's version chain and then scans the chain to locate the version that is visible for a transaction. The DBMS will never incur a false negative from an index, but it may get false positive matches because the index can point to a version for a key that may not be visible to a particular transaction.

Primary key indexes always point to the current version of a tuple. But how often the DBMS updates a primary key index depends on whether or not its version storage scheme creates new versions when a tuple is updated. For example, a primary key index in the delta scheme always points to the master version for a tuple in the main table, thus the index does not need to be updated. For append-only, it depends on the version chain ordering: N2O requires the DBMS to update the primary key index every time a new version is created. If a tuple's primary key is modified, then the DBMS applies this to the index as a `DELETE` followed by an `INSERT`.

For secondary indexes, it is more complicated because an index entry's keys and pointers can both change. The two management schemes for secondary indexes in a MVCC DBMS differ on the contents of these pointers. The first approach uses *logical pointers* that use indirection to map to the location of the physical version. Contrast this with the *physical pointers* approach where the value is the location of an exact version of the tuple.

6.1 Logical Pointers

The main idea of using logical pointers is that the DBMS uses a fixed identifier that does not change for each tuple in its index entry. Then, as shown in Fig. 5a, the DBMS uses an indirection layer that maps a tuple's identifier to the HEAD of its version chain. This avoids the problem of having to update all of a table's indexes to point to a new physical location whenever a tuple is modified



(a) Logical Pointers **(b)** Physical Pointers

Figure 5: Index Management – The two ways to map keys to tuples in a MVCC are to use logical pointers with an indirection layer to the version chain HEAD or to use physical pointers that point to an exact version.

(even if the indexed attributes were not changed). Only the mapping entry needs to change each time. But since the index does not point to the exact version, the DBMS traverses the version chain from the HEAD to find the visible version. This approach is compatible with any version storage scheme. As we now discuss, there are two implementation choices for this mapping:

Primary Key (PKey): With this, the identifier is the same as the corresponding tuple's primary key. When the DBMS retrieves an entry from a secondary index, it performs another look-up in the table's primary key index to locate the version chain HEAD. If a secondary index's attributes overlap with the primary key, then the DBMS does not have to store the entire primary key in each entry.

Tuple Id (TupleId): One drawback of the PKey pointers is that the database's storage overhead increases as the size of a tuple's primary key increases, since each secondary index has an entire copy of it. In addition to this, since most DBMSs use an order-preserving data structure for its primary key indexes, the cost of performing the additional look-up depends on the number of entries. An alternative is to use a unique 64-bit tuple identifier instead of the primary key and a separate latch-free hash table to maintain the mapping information to the tuple's version chain HEAD.

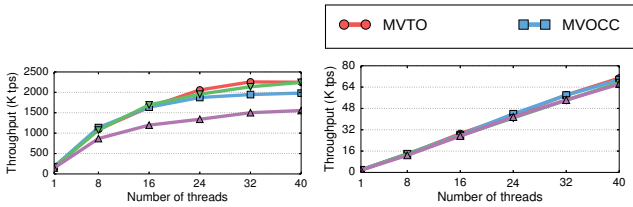
6.2 Physical Pointers

With this second scheme, the DBMS stores the physical address of versions in the index entries. This approach is only applicable for append-only storage, since the DBMS stores the versions in the same table and therefore all of the indexes can point to them. When updating any tuple in a table, the DBMS inserts the newly created version into all the secondary indexes. In this manner, the DBMS can search for a tuple from a secondary index without comparing the secondary key with all of the indexed versions. Several MVCC DBMSs, including MemSQL and Hekaton, employ this scheme.

6.3 Discussion

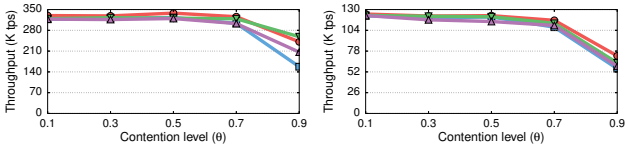
Like the other design decisions, these index management schemes perform differently on varying workloads. The logical pointer approach is better for write-intensive workloads, as the DBMS updates the secondary indexes only when a transaction modifies the indexes attributes. Reads are potentially slower, however, because the DBMS traverses version chains and perform additional key comparisons. Likewise, using physical pointers is better for read-intensive workloads because an index entry points to the exact version. But it is slower for update operations because this scheme requires the DBMS to insert an entry into every secondary index for each new version, which makes update operations slower.

One last interesting point is that index-only scans are not possible in a MVCC DBMS unless the tuples' versioning information is embedded in each index. The system has to always retrieve this information from the tuples themselves to determine whether each tuple version is visible to a transaction. NuoDB reduces the amount of data read to check versions by storing the header meta-data separately from the tuple data.



(a) Short Transactions (#Ops=1) **(b)** Long Transactions (#Ops=100)

Figure 6: Scalability Bottlenecks – Throughput comparison of the concurrency control protocols using the read-only YCSB workload with different number of operations per transaction.



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 7: Transaction Contention – Comparison of the concurrency control protocols (40 threads) for the YCSB workload with different workload/contention mixtures. Each transaction contains 10 operations.

7. EXPERIMENTAL ANALYSIS

We now present our analysis of the transaction management design choices discussed in this paper. We made a good faith effort to implement state-of-the-art versions of each of them in the Peloton DBMS [5]. Peloton stores tuples in row-oriented, unordered in-memory heaps. It uses libcuckoo [19] hash tables for its internal data structures and the Bw-Tree [32] for database indexes. We also optimized Peloton's performance by leveraging latch-free programming techniques [15]. We execute all transactions as stored procedures under the `SERIALIZABLE` isolation level. We configured Peloton to use the epoch-based memory management (see Sect. 5) with 40 ms epochs [44].

We deployed Peloton on a 4-socket Intel Xeon E7-4820 server with 128 GB of DRAM running Ubuntu 14.04 (64-bit). Each socket contains ten 1.9 GHz cores and 25 MB of L3 cache.

We begin with a comparison of the concurrency control protocols. We then pick the best overall protocol and use it to evaluate the version storage, garbage collection, and index management schemes. For each trial, we execute the workload for 60 seconds to let the DBMS to warm up and measure the throughput after another 120 seconds. We execute each trial five times and report the average execution time. We summarize our findings in Sect. 8.

7.1 Benchmarks

We next describe the workloads that we use in our evaluation.

YCSB: We modified the YCSB [14] benchmark to model different workload settings of OLTP applications. The database contains a single table with 10 million tuples, each with one 64-bit primary key and 10 64-bit integer attributes. Each operation is independent; that is, the input of an operation does not depend on the output of a previous operation. We use three workload mixtures to vary the number of reads/update operations per transaction: (1) **read-only** (100% *reads*), (2) **read-intensive** (80% *reads*, 20% *updates*), and (3) **update-intensive** (20% *reads*, 80% *updates*). We also vary the number of attributes that operations read or update in a tuple. The operations access tuples following a Zipfian distribution that is controlled by a parameter (θ) that affects the amount of contention (i.e., skew), where θ=1.0 is the highest skew setting.

TPC-C: This benchmark is the current standard for measuring the performance of OLTP systems [43]. It models a warehouse-centric order processing application with nine tables and five transaction types. We modified the original TPC-C workload to include a new table scan query, called `StockScan`, that scans the `Stock` table and counts the number of items in each warehouse. The amount of contention in the workload is controlled by the number of warehouses.

7.2 Concurrency Control Protocol

We first compare the DBMS's performance with the concurrency control protocols from Sect. 3. For serialization certifier, we implement SSN on top of snapshot isolation (denoted as SI+SSN) [45]. We fix the DBMS to use (1) append-only storage with N2O ordering, (2) transaction-level GC, and (3) logical mapping index pointers.

Our initial experiments use the YCSB workload to evaluate the protocols. We first investigate the bottlenecks that prevent them from scaling. We then compare their performance by varying workload contention. After that, we show how each protocol behaves when processing heterogeneous workloads that contain both read-write and read-only transactions. Lastly, we use the TPC-C benchmark to examine how each protocol behaves under real-world workloads.

Scalability Bottlenecks: This experiment shows how the protocols perform on higher thread counts. We configured the read-only YCSB workload to execute transactions that are either *short* (one operation per transaction) or *long* (100 operations per transaction). We use a low skew factor (θ=0.2) and scale the number of threads.

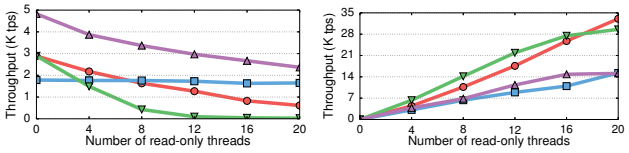
The short transaction workload results in Fig. 6a show that all but one of the protocols scales almost linearly up to 24 threads. The main bottleneck for all of these protocols is the cache coherence traffic from updating the memory manager's counters and checking for conflicts when transactions commit (even though there are no writes). The reason that SI+SSN achieves lower performance is that it maintains a centralized hash table for tracking transactions. This bottleneck can be removed by pre-allocating and reusing transaction context structures [24]. When we increase the transaction length to 100 operations, Fig. 6b shows that the throughput of the protocols is reduced by $\sim$30$\times$ but they scale linearly up to 40 threads. This is expected since the contention on the DBMS's internal data structures is reduced when there are fewer transactions executed.

Transaction Contention: We next compare the protocols under different levels of contention. We fix the number of DBMS threads to 40. We use the read-intensive and update-intensive workloads with 10 operations per transaction. For each workload, we vary the contention level (θ) in the transactions' access patterns.

Fig. 7a shows the DBMS's throughput for the read-intensive workload. When θ is less than 0.7, we see that all of the protocols achieve similar throughput. Beyond this contention level, the performance of MVOCC is reduced by $\sim$50%. This is because MVOCC does not discover that a transaction will abort due to a conflict until after the transaction has already executed its operations. There is nothing about multi-versioning that helps this situation. Although we see the same drop for the update-intensive results when contention increases in Fig. 7b, there is not a great difference among the protocols except MV2PL; they handle write-write conflicts in a similar way and again multi-versioning does not help reduce this type of conflicts.

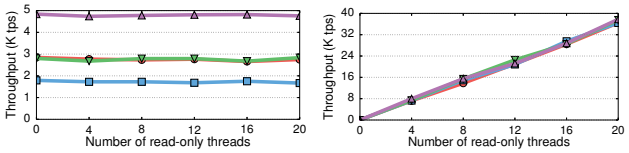
Heterogeneous Workload: In this next experiment, we evaluate a heterogeneous YCSB workload that is comprised of a mix of read-write and read-only `SERIALIZABLE` transactions. Each transaction contains 100 operations each access a single independent tuple.

The DBMS uses 20 threads to execute the read-write transactions and we vary the number of threads that are dedicated to the read-only queries. The distribution of access patterns for all operations



(a) Read-Write Throughput **(b)** Read-Only Throughput

Figure 8: Heterogeneous Workload (without `READ ONLY`) – Concurrency control protocol comparison for YCSB (θ=0.8). The read-write portion executes a update-intensive mixture on 20 threads while scaling the number of read-only threads.



(a) Read-Write Throughput **(b)** Read-Only Throughput

Figure 9: Heterogeneous Workload (with `READ ONLY`) – Concurrency control protocol comparison for YCSB (θ=0.8). The read-write portion executes a update-intensive mixture on 20 threads while scaling the number of read-only threads.

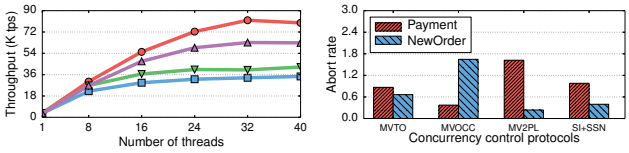
use a high contention setting (θ=0.8). We execute this workload first where the application does not pre-declare queries as `READ ONLY` and then again with this hint.

There are several interesting trends when the application does not pre-declare the read-only queries. The first is that the throughput of read-write transactions drops in Fig. 8a for the MVTO and MV2PL protocols as the number of read-only threads increases, while the throughput of read-only transactions increases in Fig. 8b. This is because these protocols treat readers and writers equally; as any transaction that reads or writes a tuple blocks other transactions from accessing the same tuple, increasing the number of read-only queries causes a higher abort rate for read-write transactions. Due to these conflicts, MV2PL only completes a few transactions when the number of read-only threads is increased to 20. The second observation is that while MVOCC achieves stable performance for the read-write portion as the number of read-only threads increases, their performance for read-only portion are lower than MVTO by 2× and 28×, respectively. The absence of read locks in MVOCC results in the starvation of read-only queries. The third observation is that SI+SSN achieves a much higher performance for read-write transactions. This is because it reduces the DBMS's abort rate due to the precise consistency validation.

The results in Fig. 9 show that the protocols perform differently when the workload pre-declares the read-only portion of the workload. The first observation is that their read-only throughput in Fig. 9b is the same because the DBMS executes these queries without checking for conflicts. And in Fig. 9a we see that their throughput for read-write transactions remains stable as the read-only queries are isolated from the read-write transactions, hence executing these read-only transactions does not increase data contention. SI+SSN again performs the best because its reduced abort rate; it is 1.6× faster than MV2PL and MVTO. MVOCC achieves the lowest performance because it can result in high abort rate due to validation failure.

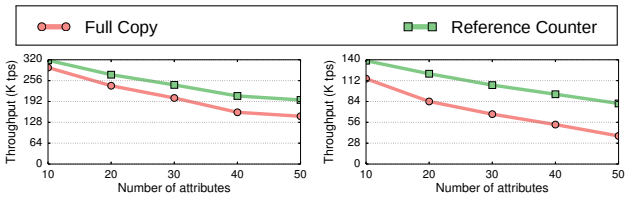
TPC-C: Lastly, we compare the protocols using the TPC-C benchmark with the number of warehouses set to 10. This configuration yields a high-contention workload.

The results in Fig. 10a show that MVTO achieves 45%–120% higher performance compared to the other protocols. SI+SSN also yields higher throughput than the rest of the protocols because it detects anti-dependencies rather than blindly aborting transactions



(a) Throughput **(b)** Abort Rate

Figure 10: TPC-C – Throughput and abort rate comparison of the concurrency control protocols with the TPC-C benchmark.



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 11: Non-Inline Attributes – Evaluation of how to store non-inline attributes in the append-only storage scheme using the YCSB workload with 40 DBMS threads and varying the number of attributes in a tuple.

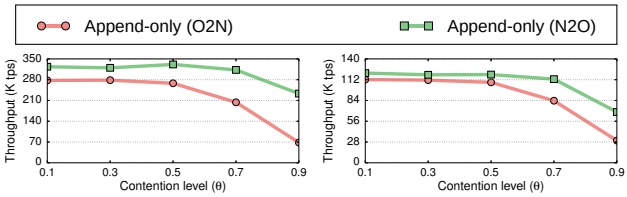
through OCC-style consistency validation. MVOCC incurs wasted computation because it only detects conflicts in the validation phase. A more interesting finding in Fig. 10b is that the protocols abort transactions in different ways. MVOCC is more likely to abort `NewOrder` transactions, whereas the `Payment` abort rate in MV2PL is 6.8× higher than `NewOrder` transactions. These two transactions access the same table, and again the optimistic protocols only detect read conflicts in `NewOrder` transactions in the validation phase. SI+SSN achieves a low abort rate due to its anti-dependency tracking, whereas MVTO avoids false aborts because the timestamp assigned to each transaction directly determines their ordering.

7.3 Version Storage

We next evaluate the DBMS's version storage schemes. We begin with an analysis of the storage mechanisms for non-inline attributes in append-only storage. We then discuss how the version chain ordering affects the DBMS's performance for append-only storage. We next compare append-only with the time-travel and delta schemes using different YCSB workloads. Lastly, we compare all of the schemes again using the TPC-C benchmark. For all of these experiments, we configured the DBMS to use the MVTO protocol since it achieved the most balanced performance in the previous experiments.

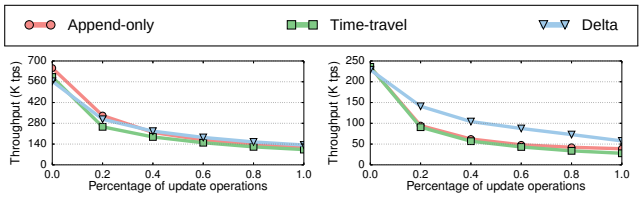
Non-Inline Attributes: This first experiment evaluates the performance of different mechanisms for storing non-inline attributes in append-only storage. We use the YCSB workload mixtures in this experiment, but the database is changed to contain a single table with 10 million tuples, each with one 64-bit primary key and a variable number of 100-byte non-inline `VARCHAR` type attributes. We use the read-intensive and update-intensive workloads under low contention (θ=0.2) on 40 threads with each transaction executing 10 operations. Each operation only accesses one attribute in a tuple.

Fig. 11 shows that maintaining reference counters for unmodified non-inline attributes always yields better performance. With the read-intensive workload, the DBMS achieves ∼40% higher throughput when the number of non-inlined attributes is increased to 50 with these counters compared to conventional full-tuple-copy scheme. This is because the DBMS avoids redundant data copying for update operations. This difference is more prominent with the update-intensive workload where the results in Fig. 11b show that the performance gap reaches over 100%.



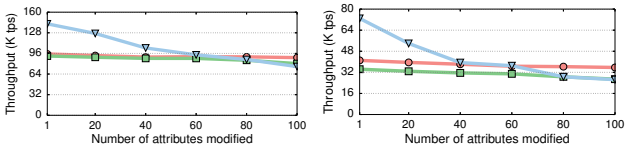
(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 12: Version Chain Ordering – Evaluation of the version chains for the append-only storage scheme using the YCSB workload with 40 DBMS threads and varying contention levels.



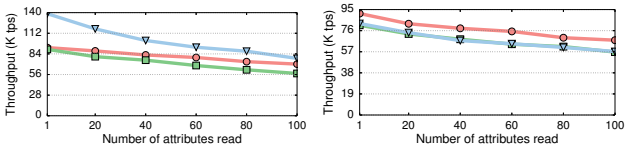
(a) 10 Attributes **(b)** 100 Attributes

Figure 13: Transaction Footprint – Evaluation of the version storage schemes using the YCSB workload (θ=0.2) with 40 DBMS threads and varying the percentage of update operations per transaction.



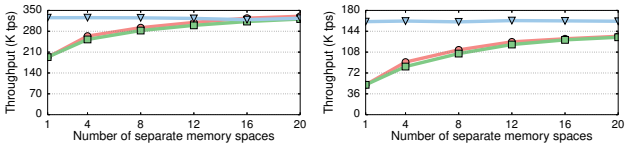
(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 14: Attributes Modified – Evaluation of the version storage schemes using YCSB (θ=0.2) with 40 DBMS threads and varying the number of the tuples' attributes that are modified per update operation.



(a) One Modified Attribute per Update **(b)** 100 Modified Attributes per Update

Figure 15: Attributes Accessed – Evaluation of the version storage schemes using YCSB (θ=0.2) with 40 DBMS threads and varying the number of the tuples' attributes that are accessed per read operation.

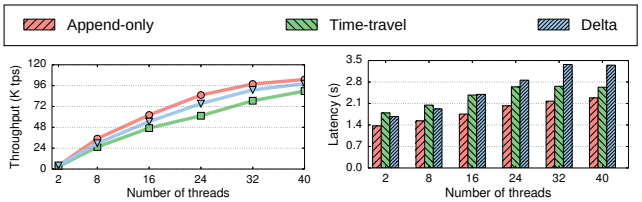


(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 16: Memory Allocation – Evaluation of the memory allocation effects to the version storage schemes using the YCSB workload with 40 DBMS threads and varying the number of separate memory spaces.

Version Chain Ordering: The second experiment measures the performance of the N2O and O2N version chain orderings from Sect. 4.1. We use transaction-level background vacuuming GC and compare the orderings using two YCSB workload mixtures. We set the transaction length to 10. We fix the number of DBMS threads to 40 and vary the workload's contention level.

As shown in Fig. 12, the N2O ordering always performs better than O2N in both workloads. Although the DBMS updates the indexes' pointers for each new version under N2O, this is overshadowed by the cost of traversing the longer chains in O2N. Increasing the length of the chains means that transactions take longer to execute, thereby increasing the likelihood that a transaction will conflict with another one. This phenomenon is especially evident with the measurements under the highest contention level (θ=0.9), where the N2O ordering achieves 2.4–3.4× better performance.



(a) Throughput **(b)** Scan Latency

Figure 17: TPC-C – Throughput and latency comparison of the version storage schemes with the TPC-C benchmark.

Transaction Footprint: We next compare the storage schemes when we vary the number of attributes in the tuples. We use the YCSB workload under low contention (θ=0.2) on 40 threads with each transaction executing 10 operations. Each read/update operation only accesses/modifies one attribute in the tuple. We use append-only storage with N2O ordering. For all the version storage schemes, we have allocated multiple separate memory spaces to reduce memory allocation overhead.

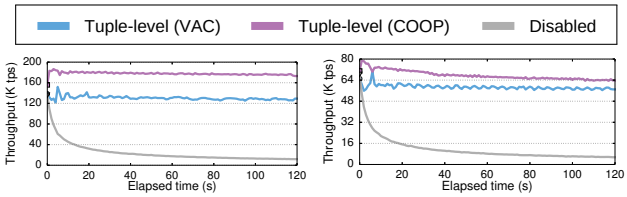
As shown in Fig. 13a, the append-only and delta schemes achieve similar performance when the table has 10 attributes. Likewise, the append-only and time-travel throughput is almost the same. The results in Fig. 13b indicate that when the table has 100 attributes, the delta scheme achieves ∼2× better performance than append-only and time-travel schemes because it uses less memory.

Attributes Modified: We now fix the number of attributes in the table to 100 and vary the number of attributes that are modified by transactions per update operation. We use the read-intensive and update-intensive workloads under low contention (θ=0.2) on 40 threads with each transaction executing 10 operations. Like the previous experiment, each read operation accesses one attribute.

Fig. 14 shows that the append-only and time-travel schemes' performance is stable regardless of the number of modified attributes. As expected, the delta scheme performs the best when the number of modified attributes is small because it copies less data per version. But as the scope of the update operations increases, it is equivalent to the others because it copies the same amount of data per delta.

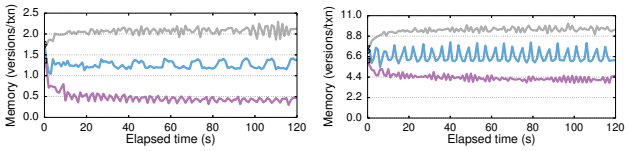
To measure how modified attributes affect reads, we vary the number of attributes accessed per read operation. Fig. 15a shows that when updates only modify one (random) attribute, increasing the number of read attributes largely affects the delta schemes. This is expected as the DBMS has to spend more time traversing the version chains to retrieve targeted columns. The performance of append-only storage and time-travel storage also degrades because the inter-socket communication overhead increases proportionally to the amount of data accessed by each read operation. This observation is consistent with the results in Fig. 15b, where update operations modify all of the tuples' attributes, and increasing the number of attributes accessed by each read operation degrades the performance of all the storage schemes.

Memory Allocation: We next evaluate how memory allocation affects the performance of the version storage schemes. We use the YCSB workload under low contention (θ=0.2) on 40 threads. Each transaction executes 10 operations that each access only one attribute of a tuple. We change the number of separate memory spaces and measure the DBMS's throughput. The DBMS expands each memory space in 512 KB increments.



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 18: Tuple-level Comparison (Throughput) – The DBMS's throughput measured over time for YCSB workloads with 40 threads using the tuple-level GC mechanisms.



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 19: Tuple-level Comparison (Memory) – The amount of memory that the DBMS allocates per transaction over time (lower is better) for YCSB workloads with 40 threads using the tuple-level GC mechanisms.

Fig. 16 shows that the delta storage scheme's performance is stable regardless of the number of memory spaces that the DBMS allocates. In contrast, the append-only and time-travel schemes' throughput is improved by 1.6–4× when increasing the number of separate memory spaces from 1 to 20. This is because delta storage only copies the modified attributes of a tuple, which requires a limited amount of memory. Contrast to this, the other two storage schemes frequently acquire new slots to hold the full copy of every newly created tuple version, thereby increasing the DBMS's memory allocation overhead.

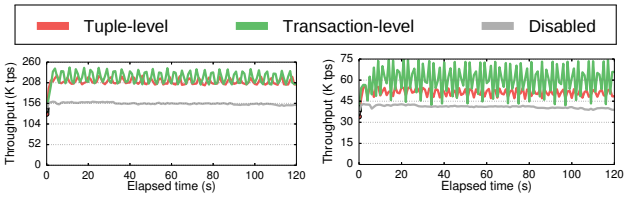
TPC-C: Lastly, we compare the schemes using TPC-C. We set the number of warehouses to 40, and scale up the number of threads to measure the overall throughput and the `StockScan` query latency.

The results in Fig. 17a show that append-only storage achieves comparatively better performance than the other two schemes. This is because this scheme can lead to lower overhead when performing multi-attribute read operations, which are prevalent in the TPC-C benchmark. Although the delta storage scheme allocates less memory when creating new versions, this advantage does not result in a notable performance gain as our implementation has optimized the memory management by maintaining multiple spaces. Time-travel scheme suffers lower throughput as it does not bring any benefits for read or write operations. In Fig. 17b, we see that the append-only and time-travel schemes are better for table scan queries. With delta storage, the latency of the scan queries grows near-linearly with the increase of number of threads (which is bad), while the append-only and time-travel schemes maintain a latency that is 25–47% lower when using 40 threads.

7.4 Garbage Collection

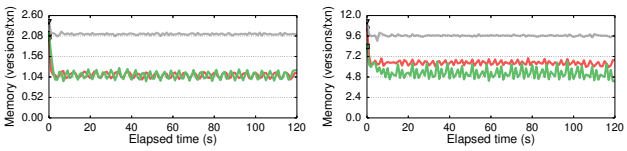
We now evaluate the GC mechanisms from Sect. 5. For these experiments, we use the MVTO concurrency control protocol. We first compare background versus cooperative cleaning in tuple-level GC. We then compare tuple-level and transaction-level approaches.

Tuple-level Comparison: We use the update-intensive workload (10 operations per transaction) with low and high contentions. The DBMS uses append-only storage with O2N ordering, as COOP only works with this ordering. We configure the DBMS to use 40 threads for transaction processing and one thread for GC. We report both the throughput of the DBMS over time as well as the amount of new memory that is allocated in the system. To better understand the impact of GC, we also execute the workload with it disabled.



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 20: Tuple-level vs. Transaction-level (Throughput) – Sustained throughput measured over time for two YCSB workloads (θ=0.8) using the different GC mechanisms.



(a) Read-Intensive (R/W=80/20%) **(b)** Update-Intensive (R/W=20/80%)

Figure 21: Tuple-level vs. Transaction-level (Memory) – The amount of memory that the DBMS allocates per transaction over time (lower is better) for two YCSB workloads (θ=0.8) using the different GC mechanisms.

The results in Fig. 18 show that COOP achieves 45% higher throughput compared to VAC under read-intensive workloads. In Fig. 19, we see that COOP has a 30–60% lower memory footprint per transaction than VAC. Compared to VAC, COOP's performance is more stable, as it amortizes the GC overhead across multiple threads and the memory is reclaimed more quickly. For both workloads, we see that performance declines over time when GC is disabled because the DBMS traverses longer version chains to retrieve the versions. Furthermore, because the system never reclaims memory, it allocates new memory for every new version.

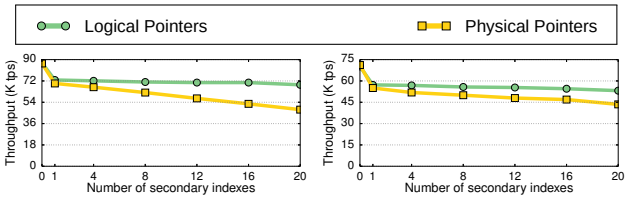
Tuple-level vs. Transaction-level: We next evaluate the DBMS's performance when executing two YCSB workloads (high contention) mixture using the tuple-level and transaction-level mechanisms. We configure the DBMS to use append-only storage with N2O ordering. We set the number of worker threads to 40 and one thread for background vacuuming (VAC). We also execute the same workload using 40 threads but without any GC.

The results in Fig. 20a indicate that transaction-level GC achieves slightly better performance than tuple-level GC for the read-intensive, but the gap increases to 20% in Fig. 20b for the update-intensive workload. Transaction-level GC removes expired versions in batches, thereby reducing the synchronization overhead. Both mechanisms improve throughput by 20–30% compared to when GC is disabled. Fig. 21 shows that both mechanisms reduce the memory usage.

7.5 Index Management

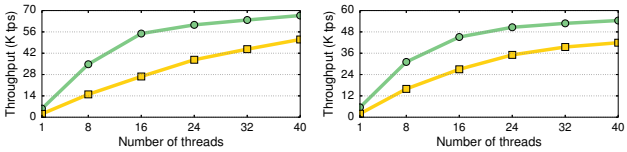
Lastly, we compare the index pointer schemes described in Sect. 6. The main aspect of a database that affects the DBMS's performance with these schemes is secondary indexes. The DBMS updates pointers any time a new version is created. Thus, we evaluate the schemes while increasing the number of secondary indexes in the database with the update-intensive YCSB workload. We configure DBMS to use the MVTO concurrency control protocol with append-only storage (N2O ordering) and transaction-level COOP GC for all of the trials. We use append-only storage because it is the only scheme that supports physical index pointers. For logical pointers, we map each index key to the HEAD of a version chain.

The results in Fig. 22b show that under high contention, logical pointer achieves 25% higher performance compared to physical pointer scheme. Under low contention, Fig. 22a shows that the



(a) Low contention (θ=0.2) **(b)** High contention (θ=0.8)

Figure 22: Index Management – Transaction throughput achieved by varying the number of secondary indexes.



(a) Low contention (θ=0.2) **(b)** High contention (θ=0.8)

Figure 23: Index Management – Throughput for update-intensive YCSB with eight secondary indexes when varying the number of threads.

performance gap is enlarged to 40% with the number of secondary indexes increased to 20. Fig. 23 further shows the advantage of logical pointers. The results show that for the high contention workload, the DBMS's throughput when using logical pointers is 45% higher than the throughput of physical pointers. This performance gap decreases in both the low contention and high contention workloads with the increase of number of threads.

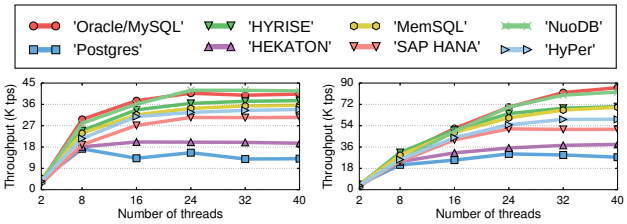
8. DISCUSSION

Our analyses and experiments of these transaction management design schemes in MVCC DBMSs produced four findings. Foremost is that the version storage scheme is one of the most important components to scaling an in-memory MVCC DBMS in a multi-core environment. This goes against the conventional wisdom in database research that has mostly focused on optimizing the concurrency control protocols [48]. We observed that the performance of append-only and time-travel schemes are influenced by the efficiency of the underlying memory allocation schemes; aggressively partitioning memory spaces per core resolves this problem. Delta storage scheme is able to sustain a comparatively high performance regardless of the memory allocation, especially when only a subset of the attributes stored in the table is modified. But this scheme suffers from low table scan performance, and may not be a good fit for read-heavy analytical workloads.

We next showed that using a workload-appropriate concurrency control protocol improves the performance, particularly on high-contention workloads. The results in Sect. 7.2 show that the protocol optimizations can hurt the performance on these workloads. Overall, we found that MVTO works well on a variety of workloads. None of the systems that we list in Table 1 adopt this protocol.

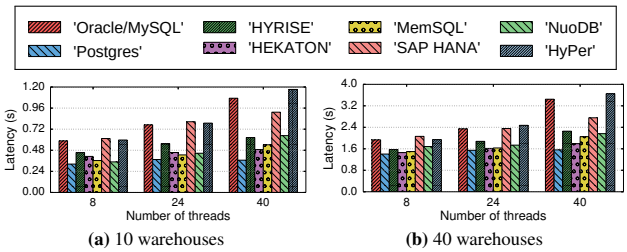
We also observed that the performance of a MVCC DBMS is tightly coupled with its GC implementation. In particular, we found that a transaction-level GC provided the best performance with the smallest memory footprint. This is because it reclaims expired tuple versions with lower synchronization overhead than the other approaches. We note that the GC process can cause oscillations in the system's throughput and memory footprint.

Lastly, we found that the index management scheme can also affect the DBMS's performance for databases with many secondary indexes are constructed. The results in Sect. 7.5 show that logical pointer scheme always achieve a higher throughput especially when processing update-intensive workloads. This corroborates other reports in industry on this problem [25].



(a) 10 warehouses **(b)** 40 warehouses

Figure 24: Configuration Comparison (Throughput) – Performance of the MVCC configurations from Table 1 with the TPC-C benchmark.



(a) 10 warehouses **(b)** 40 warehouses

Figure 25: Configuration Comparison (Scan Latency) – Performance of the MVCC configurations from Table 1 with the TPC-C benchmark.

To verify these findings, we performed one last experiment with Peloton where we configured it to use the MVCC configurations listed in Table 1. We execute the TPC-C workload and use one thread to repeatedly execute the `StockScan` query. We measure the DBMS's throughput and the average latency of `StockScan` queries. We acknowledge that there are other factors in the real DBMSs that we are not capturing in this experiment (e.g., data structures, storage architecture, query compilation), but this is still a good approximation of their abilities.

As shown in Fig. 24, the DBMS performs the best on both the low-contention and high-contention workloads with the Oracle/MySQL and NuoDB configurations. This is because these systems' storage schemes scale well in multi-core and in-memory systems, and their MV2PL protocol provides comparatively higher performance regardless of the workload contention. HYRISE, MemSQL, and HyPer's configurations yield relatively lower performance, as the use of MVOCC protocol can bring high overhead due to the read-set traversal required by the validation phase. Postgres and Hekaton's configurations lead to the worst performance, and the major reason is that the use of append-only storage with O2N ordering severely restricts the scalability of the system. This experiment demonstrates that both concurrency control protocol and version storage scheme can have a strong impact on the throughput.

But the latency results in Fig. 25 show that the DBMS's performance is the worst with delta storage. This is because the delta storage has to spend more time on traversing version chains so as to find the targeted tuple version attribute.

9. RELATED WORK

The first mention of MVCC appeared in Reed's 1979 dissertation [38]. After that, researchers focused on understanding the theory and performance of MVCC in single-core disk-based DBMSs [9, 11, 13]. We highlight the more recent research efforts.

Concurrency Control Protocol: There exist several works proposing new techniques for optimizing in-memory transaction processing [46, 47]. Larson et al. [27] compare pessimistic (MV2PL) and optimistic (MVOCC) protocols in an early version of the Microsoft Hekaton DBMS [16]. Lomet et al. [31] proposed a scheme that uses ranges of timestamps for resolving conflicts among transactions, and

Faleiro et al. [18] decouple MVCC's concurrency control protocol and version management from the DBMS's transaction execution. Given the challenges in guaranteeing MVCC serializability, many DBMSs instead support a weaker isolation level called snapshot isolation [8] that does not preclude the write-skew anomaly. Serializable snapshot isolation (SSI) ensures serializability by eliminating anomalies that can happen in snapshot isolation [12, 20]. Kim et al. [24] use SSN to scale MVCC on heterogeneous workloads. Our study here is broader in its scope.

Version Storage: Another important design choice in MVCC DBMSs is the version storage scheme. Herman et al. [23] propose a differential structure for transaction management to achieve high write throughput without compromising the read performance. Neumann et al. [36] improved the performance of MVCC DBMSs with the transaction-local storage optimization to reduce the synchronization cost. These schemes differ from the conventional append-only version storage scheme that suffers from higher memory allocation overhead in main-memory DBMSs. Arulraj et al. [7] examine the impact of physical design on the performance of a hybrid DBMS while running heterogeneous workloads.

Garbage Collection: Most DBMSs adopt a tuple-level background vacuuming garbage collection scheme. Lee et al. [29] evaluate a set of different garbage collection schemes used in modern DBMSs. They propose a new hybrid scheme for shrinking the memory footprint in SAP HANA. Silo's epoch-based memory management approach allows a DBMS to scale to larger thread counts [44]. This approach reclaims versions only after an epoch (and preceding epochs) no longer contain an active transaction.

Index Management: Recently, new index data structures have been proposed to support scalable main-memory DBMSs. Lomet et al. [32] introduced a latch-free, order preserving index, called the Bw-Tree, which is currently used in several Microsoft products. Leis et al. [30] and Mao et al. [34] respectively proposed ART and Masstree, which are scalable index structures based on tries. Instead of examining the performance of different index structures, this work focuses on how different secondary index management schemes impact the performance of MVCC DBMSs.

10. CONCLUSION

We presented an evaluation of the design decisions of transaction management with in-memory MVCC. We described the state-of-the-art implementations for each of them and showed how they are used in existing systems. We then implemented them in the Peloton DBMS and evaluated them using OLTP workloads to highlight their trade-offs. We demonstrated the issues that prevent a DBMS from supporting larger CPU core counts and more complex workloads.

Acknowledgements: This work was supported (in part) by the National Science Foundation (CCF-1438955) and the Samsung Fellowship Program. We also thank Tianzheng Wang for his feedback.

11. REFERENCES

[1] MemSQL. http://www.memsql.com.
[2] MySQL. http://www.mysql.com.
[3] NuoDB. http://www.nuodb.com.
[4] Oracle Timeline. http://oracle.com.edgesuite.net/timeline/oracle/.
[5] Peloton. http://pelotondb.org.
[6] PostgreSQL. http://www.postgresql.org.
[7] J. Arulraj and *et al.* Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. SIGMOD, 2016.
[8] H. Berenson and *et al.* A Critique of ANSI SQL Isolation Levels. SIGMOD'95.
[9] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *CSUR*, 13(2), 1981.
[10] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR*, 2011.
[11] P. A. Bernstein and *et al.* *Concurrency Control and Recovery in Database Systems*. 1987.
[12] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. SIGMOD, 2008.
[13] M. J. Carey and W. A. Muhanna. The Performance of Multiversion Concurrency Control Algorithms. *TOCS*, 4(4), 1986.
[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.
[15] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted To Know About Synchronization But Were Afraid To Ask. In *SOSP*, 2013.
[16] C. Diaconu and *et al.* Hekaton: SQL Server's Memory-Optimized OLTP Engine. SIGMOD, 2013.
[17] K. P. Eswaran and *et al.* The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11), 1976.
[18] J. M. Faleiro and D. J. Abadi. Rethinking Serializable Multiversion Concurrency Control. VLDB, 2014.
[19] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI*, 2013.
[20] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. *TODS*, 30(2), 2005.
[21] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. VLDB, 2010.
[22] A. Harrison. InterBase's Beginnings. http://www.firebirdsql.org/en/ann-harrison-s-reminiscences-on-interbase-s-beginnings/.
[23] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional Update Handling in Column Stores. SIGMOD, 2010.
[24] K. Kim, T. Wang, J. Ryan, and I. Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. SIGMOD, 2016.
[25] E. Klitzke. Why uber engineering switched from postgres to mysql. https://eng.uber.com/mysql-migration/, July 2016.
[26] H.-T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *TODS*, 6(2), 1981.
[27] P.-Å. Larson and *et al.* High-Performance Concurrency Control Mechanisms for Main-Memory Databases. VLDB, 2011.
[28] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krueger, and M. Grund. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2), 2013.
[29] J. Lee and *et al.* Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. SIGMOD, 2016.
[30] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. ICDE, 2013.
[31] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-Version Concurrency via Timestamp Range Conflict Management. ICDE, 2012.
[32] D. B. Lomet, S. Sengupta, and J. J. Levandoski. The Bw-Tree: A B-tree for New Hardware Platforms. ICDE, 2013.
[33] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking Main memory OLTP Recovery. ICDE, 2014.
[34] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, 2012.
[35] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB'90.
[36] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. SIGMOD, 2015.
[37] A. Pavlo and M. Aslett. What's Really New with NewSQL? *SIGMOD Rec.*, 45(2):45–55, June 2016.
[38] D. P. Reed. Naming and Synchronization in a Decentralized Computer System. *Ph.D. dissertation*, 1978.
[39] D. P. Reed. Implementing Atomic Actions on Decentralized Data. *TOCS*, 1983.
[40] V. Sikka and *et al.* Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. SIGMOD, 2012.
[41] M. Stonebraker and L. A. Rowe. The Design of POSTGRES. SIGMOD, 1986.
[42] M. Stonebraker and *et al.* The End of an Architectural Era: (It's Time for a Complete Rewrite). VLDB, 2007.
[43] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf, June 2007.
[44] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*, 2013.
[45] T. Wang, R. Johnson, A. Fekete, and I. Pandis. Efficiently Making (Almost) Any Concurrency Control Mechanism Serializable. *arXiv:1605.04292*, 2016.
[46] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *SIGMOD*, 2016.
[47] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*, 2016.
[48] X. Yu and *et al.* Staring Into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. VLDB, 2014.
[49] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*, 2014.