Scalable and Robust Snapshot Isolation for High-Performance Storage Engines

Adnan Alhomssi
Friedrich-Alexander-Universität Erlangen-Nürnberg
adnan.alhomssi@fau.de

Viktor Leis
Technische Universität München
leis@in.tum.de

ABSTRACT

MVCC-based snapshot isolation promises that read queries can proceed without interfering with concurrent writes. However, as we show experimentally, in existing implementations a single long-running query can easily cause transactional throughput to collapse. Moreover, existing out-of-memory commit protocols fail to meet the scalability needs of modern multi-core systems. In this paper, we present three complementary techniques for robust and scalable snapshot isolation in out-of-memory systems. First, we propose a commit protocol that minimizes cross-thread communication for better scalability, avoids touching the write set on commit, and enables efficient fine-granular garbage collection. Second, we introduce the Graveyard Index, an auxiliary data structure that moves logically-deleted tuples out of the way of operational transactions. Third, we present an adaptive version storage scheme that enables fast garbage collection and improves scan performance of frequently-modified tuples. All techniques are engineered to scale well on multi-core processors, and together enable robust performance for complex hybrid workloads.

PVLDB Reference Format:
Adnan Alhomssi and Viktor Leis. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. PVLDB, 16(6): 1426 - 1438, 2023.
doi:10.14778/3583140.3583157

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/leanstore/leanstore/tree/mvcc.

1 INTRODUCTION

MVCC. Despite the focus of many textbooks on Two-Phase Locking (2PL) and single-version storage, today most widely-used database systems rely on *Multi-Version Concurrency Control (MVCC)*. For example, this includes PostgreSQL, InnoDB, WiredTiger, Oracle, SQL Server, Vertica, HANA, Snowflake, and Redshift. The distinguishing idea of MVCC is to update tuples out-of-place and keep old versions visible to concurrent transactions. MVCC naturally allows implementing *Snapshot Isolation (SI)*: with SI, read transactions can proceed without acquiring read locks (2PL), without validating the read/write sets (OCC), and without writing tuple access timestamps (timestamp ordering). Indeed, without multiple versions and snapshotting, it is hard to imagine how large scans and updates can coexist without constantly interfering. MVCC-based SI therefore seems uniquely suitable for complex general-purpose workloads comprising both transactional data modifications and long-running analytical queries.

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583157

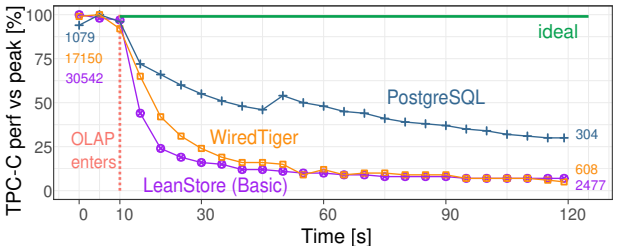


Figure 1: TPC-C performance collapses as long-running OLAP query occurs after 10 seconds (1 OLTP, 1 OLAP thread)

Challenges of Existing MVCC Designs. While snapshot isolation seems promising for hybrid workloads, general-purpose Hybrid Transaction/Analytical Processing (HTAP) in one system is still uncommon. Most organizations split their operational database from their data warehouse – even though both kinds of systems usually implement MVCC. One reason for this is physical storage: a row store is clearly more efficient for OLTP, and columnar storage is better for analytics. However, there is also a second reason for the split: current MVCC schemes are often unsuccessful in isolating the two workload classes from each other. In other words, OLAP may lead to performance collapse for OLTP and vice versa. Consider Figure 1, which shows the normalized TPC-C performance over time in three MVCC systems. Performance stays stable for 10s – until a long-running OLAP query enters. Although the query in this experiment does nothing and just sleeps – keeping the snapshot open – it causes OLTP performance to collapse. This is obviously unacceptable for mission-critical systems.

Commit Protocols for High-Performance Storage Engines. Existing disk-based storage engines lack not only robustness but also scalability. Common out-of-memory SI commit protocols, e.g., the ones used by PostgreSQL, InnoDB, and WiredTiger, construct a snapshot using a vector of concurrent transaction IDs. This design scales poorly with many threads as it has to fetch current transaction ID from other threads even if the transaction won't read versions written by them. It also leads to expensive visibility checks that make fine-granular garbage collection expensive. Porting in-memory commit protocols [16, 33, 39] into out-of-memory systems is also not viable because touching the write set again on commit can be expensive in out-of-memory workloads.

Goals. The goals of this work are to (i) prevent OLTP performance from collapsing in the presence of long-running queries, (ii) ensure

scalability on multi-core processors, (iii) support out-of-memory workloads, and (iv) maintain the performance of long-running queries. To achieve this, we had to rethink all facets of MVCC.

Contribution 1: SI Commit Protocol (OSIC). As the foundation of our design, we present *Ordered Snapshot Instant Commit (OSIC)*, a commit protocol that brings the scalability and the cheap visibility checks of in-memory protocols to out-of-memory storage engines. OSIC achieves this through a fixed-size, per-thread data structure called *Commit Log*.

Contribution 2: Graveyard Index. OLTP workloads that exhibit a *queue*-like workload pattern such as TPC-C face performance degradation in the presence of long-running queries, as is shown in Figure 1. This happens due to the accumulation of logically-deleted tuples in the queue index; these tuples are visible to the long-running query and therefore cannot be garbage collected. To solve this problem, we propose the *Graveyard Index*, an auxiliary data structure that moves tombstones that are only visible to long-running queries out of the way of operational transactions.

Contribution 3: Adaptive Version Storage. MVCC performance crucially depends on where old versions are stored. In some situations, it is beneficial to have the most current version in the main index and older versions in a separate structure. In other situations, it would be better to keep all versions in the main index. We therefore propose an adaptive version storage scheme that automatically makes this choice in a workload-driven fashion. The two storage formats make garbage collection efficient and improve performance robustness for scans of frequently-updated tuples.

System Integration. We integrated our design into LeanStore [5, 34], a high-performance storage engine optimized for multi-core CPUs [6, 35] and NVMe SSDs [25]. We initially implemented an MVCC scheme that is comparable to existing state-of-the-art approaches, achieving high performance on TPC-C alone. However, as the *LeanStore (Basic)* line in Figure 1 shows, this is not robust even in the presence of a simple long-running OLAP scan. The design presented in this paper, in contrast, delivers robust and scalable performance in mixed and pure OLTP workloads. On a 64-core server, LeanStore achieves ≈2 million TPC-C transactions per second despite a concurrent long-running OLAP scan. The results were achieved under snapshot isolation and with logging [27] enabled.

2 BACKGROUND AND MOTIVATION

The Promise. The main reason for the wide adoption of systems with MVCC-based snapshot isolation is that they allow read operations to proceed without acquiring any locks. In other words: long-running read-only scan queries (e.g., a report computing the yearly revenue) should not compromise the performance of operational latency-critical transactions (e.g., an incoming order).

The Reality. In practice, existing MVCC system fail to fulfill this promise. The main problem is that update and delete operations create new tuple versions that may be physically but not logically visible to other transactions. If a long-running transaction is in the system, these tuple versions may "pile up" and cause additional, unnecessary work – which can only be solved using aggressive garbage collection (GC). A number of recent papers therefore propose variants of precise (fine-grained, interval-based) GC [13, 29, 33]. However, while these approaches help for many workloads, they fail to solve the problem in general. Indeed, even with fine-grained and aggressive GC, challenging OLTP workloads such as TPC-C still suffer from performance collapse when a long-running transaction enters the system.

Performance Collapse. Consider the performance of a mixed workload where one OLTP thread performs unmodified TPC-C transactions and one OLAP thread simply opens a transaction (snapshot) and then sleeps. Figure 1 shows normalized performance results for three MVCC implementations. Once a long-running transaction enters the system, the TPC-C throughput quickly drops and degrades over time. OLAP performance, as measured by the number of logically-visible tuples scanned per second, also deteriorates. The slope of the performance drop depends on the implemented garbage collection aggressiveness and precision. Systems that only garbage collect versions using a global minimum timestamp (high watermark) such as PostgreSQL, MySQL, and WiredTiger deteriorate faster than systems with interval-based GC such as SAP HANA [33], Steam [13], and vDriver [29]. In general, the performance of both types of transactions drops because they start to encounter and process index records that are not relevant to them because these records are either already deleted, out-dated, or recently-inserted but not yet visible to their snapshot.

2.1 Multi Version Storage

Indexes and Version Chains. Just like single-version systems, MVCC systems often use B-Tree indexes. Each index maps the user-specified attribute(s) to the version chain of the indexed tuple. The version chain maintains at least all versions of the tuple that might be required by any concurrent active transaction. The order of versions in the chain differs between systems: newest to oldest (N2O) order favors OLTP while oldest to newest (O2N) favors OLAP. In the rest of the paper, we assume N2O order – prioritizing the latency of operational transactions.

Full Copy vs. Delta. When a transaction updates a tuple, it inserts a new version at the beginning of the chain. The version can be either be a full copy or a *delta* that only contains the difference to the previous version. For instance, if we update only one 8-byte integer attribute in a 120-bytes tuple, we only have to mark the attribute we update and store the 8-byte XOR between current and next value. This, however, works only for updates that do not change the tuple size and comes at the price of greater complexity and less flexibility in manipulating the version chains. Readers cannot jump over versions and have to apply deltas in the right order to construct the correct version for their snapshot. Nevertheless, the large space savings of deltas for large tuples make their complexity worthwhile, and most modern designs therefore use deltas.

Tombstones. When a transaction deletes a key, it inserts a *tombstone* in the version chain to mark the tuple as deleted for future transactions. Once all concurrent transactions finish, the tuple becomes logically invisible for future transactions as well. Only at this point can the tombstone be physically removed from the index.

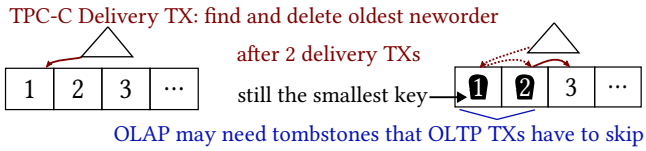
Skipping Versions. Generally, both OLTP and OLAP transactions rely on the same physical access paths. Therefore, on a low level, the index operations return the same version chain to both. A version chain that is relevant only to an old OLAP query – but totally obsolete for newer transactions – remains in the index for both.

Only after encountering a version chain, can a transaction use its snapshot information to determine which versions to retrieve or if this chain is even relevant to it, i.e., invisible in its snapshot. As we describe in the following, this additional skipping work lies at the heart of OLTP performance degradation.

2.2 OLTP Slowdown Analysis

Tombstone Skipping. OLTP performance crucially depends on indexing. After creating an index, we expect it to provide logarithmic access time. In MVCC systems, this assumption does not hold because of tombstones. Tombstones must remain in the index until the system can ensure that no transaction sees the deleted tuples. This can be become a problem in the presence of long-running transactions: physically, index range scans (and similar operations such as prefix lookups) return the same version chain to a long-running OLAP and to a recent OLTP transaction. Thus, an OLTP range scan has to skip these tombstones one by one.

The *neworder* Problem. The worst case happens when OLTP transactions themselves create the tombstones they have to skip. This might sound rare but is actually what happens in TPC-C in the presence of a long-running query (as in the experiment in Figure 1). The following figure illustrates where this happens in TPC-C:



TPC-C uses the *neworder* relation as a queue to store new orders that have not yet been shipped. The *delivery* transaction performs a range lookup on the *neworder* index to find the oldest order that should be shipped next – with the index operation returing a cursor to the smallest (leftmost) key and its version chain. From there, the concurrency control layer takes control and moves the cursor to the next right key one step at a time until it finds a tuple that is visible in its snapshot. Initially, the first *neworder* key the transaction encounters is a hit which results in a logarithmic runtime for the prefix look up. But after the transaction deletes the oldest *neworder* it has just found, it leaves a tombstone in the index because a concurrent old transaction still requires it. In the next delivery transaction, the prefix lookup will first encounter the very tombstone that was created in the last invocation. As more new orders are delivered and inserted in the index while the OLAP query is still running, the number of tombstones that OLTP transactions have to skip grows – leading to progressive deterioration.

Problem Generalization. The key takeaway is that the access pattern to the *neworder* relation resembles a queue. Transactions insert tuples at one end and remove them later on from the other. In existing systems, every transactional workload with such a pattern will face performance collapse when a long-running transaction enters the system. Furthermore, this problem happens not just with deletes but also with updates that change one of the key attributes because updates of the index key are typically implemented as a remove followed by an insert.

2.3 OLAP Slowdown Analysis

The *warehouse* Problem. There are also situations where concurrent OLTP transactions negatively impact the performance of OLAP. The challenge is scanning tuples that concurrent OLTP transactions update frequently. Because of the N2O version order, OLTP transactions append new versions at the beginning of the chain – thereby increasing the distance to the version visible to the query. In our running TPC-C+Scan example we see this pattern with the *warehouse* relation. The relation stores statistics (the *balance* attribute) for every warehouse and is therefore updated very frequently. Prior work [13, 33] has recognized and addressed this problem in main-memory systems. An observation that helps to solve the problem is that at any point in time, the number of different versions that must exist cannot exceed the number of concurrent transactions in the system. However, none of the existing *out-of-memory* systems integrate these eager and precise techniques for reasons that we explain in the following.

2.4 SI Commit Protocols

Commit Protocols. An important part of any MVCC scheme is the mechanism for quickly determining which tuples are visible to which transactions. The complexity and cost of such visibility checks depend primarily on the commit protocol. The commit protocol defines the visibility rules as it is responsible for making a transaction's changes atomically visible to future transactions on commit. It also has to guarantee the consistency of transaction snapshots. Despite being such a crucial part of any MVCC implementation, the literature rarely describes out-of-memory commit protocols in detail. In the following discussion, which is largely based on careful reading of source code, we describe the two dominant variants, highlighting their scalability characteristics, cost of visibility checks, and garbage collection.

Out-of-Memory SI Commit Protocols. In WiredTiger [3], PostgreSQL [47], and MySQL InnoDB [7], a transaction starts by drawing a transaction ID (TXID) from a global atomic counter. Then, it publishes its in-progress TXID using an atomic per-transaction counter. A transaction T1's snapshot is then created by building a ReadView (RV) vector that holds the TXIDs of all concurrent transactions that are smaller than T1's TXID. If the system is running 10 transactions in parallel, then the RV will hold up to 10 TXIDs. Each stored tuple holds the TXID of the transaction that wrote to it last (plus a logical pointer to the previous version). A tuple is visible to a transaction RV if the following two conditions (PostgreSQL also needs an extra lookup in CLOG) hold: (i) Tuple's TXID is not in the RV, i.e., is not written by a concurrent transaction. (ii) Tuple's TXID is smaller than the maximum TXID in the RV, i.e., is not written by a future transaction that started later. This protocol achieves instant commits without touching the write set (all tuples that T1 modified) again on commit. This is a decisive criterion for buffer-managed systems because marking the tuples as committed would otherwise require traversing the indexes again and/or paying the I/O cost for possibly evicted page in steal systems. However, a visibility check can cost up to #T of comparison where T is the number of concurrent threads or transactions. Cross-transaction visibility checks cost even more because each comparison could result in a cache miss. This makes any form of precise garbage collection costly in

Table 1: Conceptual comparison of commit protocols

	trad. DBMS*	in-mem. DBMS**	OSIC (LeanStore)
post-commit	$\Theta(1)$	Θ(write set)	$\Theta(1)$
snapshotting	$\Theta(T)$	$\Theta(1)$	$\Omega(1), O(T \log T)$
visibility check	$\Omega(1), O(T)$	$\Theta(1)$	$\Theta(1)$
memory usage	$\Theta(T^2)$	$\Theta(T)$	$\Theta(T^2)$

(*) PostgreSQL/InnoDB/WiredTiger (**) Hekaton/HANA/Hyper
T = #Threads or #Concurrent Transactions

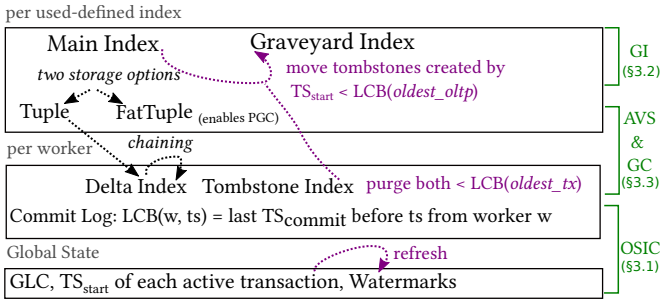


Figure 2: System overview with the proposed techniques

out-of-memory systems. Moreover, the cost of acquiring a snapshot increases linearly with the number of concurrent transactions even if the transaction will only read tuples written by a single committed transaction. This appears to be a known scalability limitation that, for example, the PostgreSQL community has acknowledged [23].

In-Memory SI Commit Protocols. In main-memory DBMS like HANA [33], Hekaton [16], Hyper [39], each transaction draws a start timestamp from a global atomic counter. Each tuple stores the start timestamp of the transaction that wrote to it until that transaction commits, but then replaces it with the commit timestamp. At commit time, the transaction draws a commit timestamp from the same global counter. A commit atomically marks the current transaction as committed and publishes its commit timestamp using a central data structure. Consequently, it goes over its write set and installs the commit timestamp. The visibility can then be determined as follows. If the tuple already holds the commit timestamp, then it is visible iff the commit timestamp is smaller than the reading transaction start timestamp. Otherwise, it checks the status of the transaction that left its start timestamp in the tuple. The other transaction could be either in-progress or committed but has not managed to install the new timestamp in its write set. In the former case, the tuple remains invisible. In the later case, one has to read the commit timestamp from the central data structure and compare it with the reading start timestamp. Oftentimes, the commit timestamp is already installed and visibility checks will only require cheap integer comparisons.

Best of Both Worlds Commit Protocol. Table 1 compares the approaches in terms of the work done post-commit, building a snapshot, visibility checks, and memory usage. In-memory protocols outperform out-of-memory ones in every dimension except for post-commit processing: (i) They build snapshots in constant time regardless of the number of concurrent transactions; (ii) Visibility checks are as cheap as two integer comparisons, so we can efficiently implement precise garbage collection techniques. However, they require re-visiting all the tuples that the transaction wrote to install the commit timestamp. In out-of-memory systems, this requirement is prohibitively expensive. One has to traverse the indexes again as virtual memory pointers may not be valid in buffer-managed systems. And in steal systems, the pages may already be evicted. In Section 3.1, we propose a commit protocol for LeanStore that minimizes cross-thread communication (hence the lower and upper bound) and brings cheap visibility checks without touching the write set as in in-memory systems.

3 ROBUST OUT-OF-MEMORY MVCC

Overview. Figure 2 shows an overview of the design and the proposed techniques and data structures. The bottom of the figure shows OSIC, an out-of-memory SI commit protocol that allows one to efficiently determine which set of transactions is visible to concurrent OLTP and OLAP transactions and provides better threads scalability by minimizing cross-thread communication while building its snapshot (Section 3.1). The top of the figure shows an auxiliary *Graveyard Index* that keeps tombstones away from the OLTP hot access path (Section 3.2). In between, two complementary version storage formats are shown that limit version chain length while keeping garbage collection efficient (Section 3.3). Section 3.4 discusses durability and recovery, and Section 3.5 explains how all components interact.

Underlying Storage Engine. Before we dive into the details of each technique, let us shortly mention some assumptions. Our scheme is geared towards out-of-memory buffer-managed engines where relations and indexes are B-Trees. We also assume that the write set of a transaction can be larger than main-memory (steal configuration). Logging is distributed across threads, each having its write ahead log [27].

3.1 Snapshot Ordering and Instant Commit

OSIC Overview. The *Ordered Snapshot Instant Commit (OSIC)* commit protocol provides snapshot isolation. As in main-memory systems, an OSIC snapshot is determined by a single integer, namely the start timestamp of the transaction. As in out-of-memory systems, transactions commit instantly without having to touch their write sets a second time on commit. OSIC makes visibility checks as cheap as a single integer comparison, and allows *ordering* transactions according to their recentness using their start timestamps. This enables efficient precise garbage collection in disk-based settings. Moreover, OSIC only requires cross-thread communication when reading recently committed tuples from another worker.

Preliminaries. In our design, every transaction draws two timestamps from the same Global Logical Clock (GLC): at the start (TS_{start}) and just before committing (TS_{commit}). The GLC is implemented as an atomic 64-bit integer. Furthermore, we follow the *first-writer wins* rule, which forces a transaction to abort if it attempts to update a tuple for which the most recent version is invisible. This rule results in recoverable schedules and provides snapshot isolation without the need for a validation phase at commit. Worker

threads process transactions in parallel. Each worker thread in our system is assigned a unique and fixed Worker ID and processes transactions one after another. $W(Tx)$ denotes the worker ID of a transaction Tx, $W(v)$ is the worker ID of the transaction that wrote version v, and $\#W$ is the total number of workers. Based on our processing model, OSIC exploits the following important invariant:

> **Transitive Commit Invariant.** If a transaction on worker w committed before timestamp ts, then so did all transactions that started before timestamp ts on worker w.

Snapshots. A Snapshot determines the set of committed transactions whose versions are considered visible to the current transaction. This set is a function of the start timestamp TS_{start} and does not change after the start timestamp was drawn. As in main-memory systems, it includes all transactions that committed before transaction start, i.e., $Snapshot(ts) = \{T | TS_{commit}(T) < ts\}$. OSIC differs from other protocols in how it determines whether a version is visible for a particular snapshot.

Write Set Stamping. While processing a transaction, we know the start timestamp but not yet its commit timestamp. For future transactions to determine whether the written version is visible, in-memory systems [16, 32, 33, 39] install the commit timestamp in the full write set while committing. When a transaction Tx reads a version v, it simply checks if $TS_{commit}(v) < TS_{start}(Tx)$. In out-of-memory systems, this approach would result in a second index traversal for every write operation because one cannot simply dereference a pointer. Furthermore, the tuples might already have been evicted to storage – e.g., occurring during bulk loading.

Indirection Approach. One approach to avoid write set stamping would be to maintain an in-memory mapping data structure between the commit and start timestamps of every transaction. After all, main-memory systems have to maintain something similar for in-flight transactions until the commit timestamp is installed. A read transaction would then retrieve the commit timestamp from the shared map instead of reading it from the tuple. The downsides of this shared mapping data structure are additional memory consumption and the potential contention on it.

OSIC Main Idea. Knowing the exact commit timestamp is, however, not the only way to determine whether the writing transaction belongs to the reader snapshot. To decide whether a version is visible, we instead rely on the Transitive Commit Invariant. It is enough to know the commit timestamp of the last completed transaction by the same worker that wrote the version before we acquire the snapshot. More formally, let *LCB(w, ts)* ("Last Committed Before") denote the last commit timestamp by a transaction on worker w before a given start timestamp ts. A version by worker w written by a transaction with $TS_{start} = vts$ is visible to a snapshot with $TS_{start} = ts$ iff $LCB(w, ts) > vts$. Consequently, to determine visibility for all tuples and transactions all we have to do is maintain the LCB for the start timestamp of every in-progress transaction. This reduces the number of transactions we have to keep state for: from all transactions to a maximum of $\#W^2$ where #W is the number of workers. With this established, we implement a simple data structure called *Commit Log* based on a fixed-size per-worker ordered array. It retains the commit timestamps of recently committed transactions as long as needed to answer LCB queries.

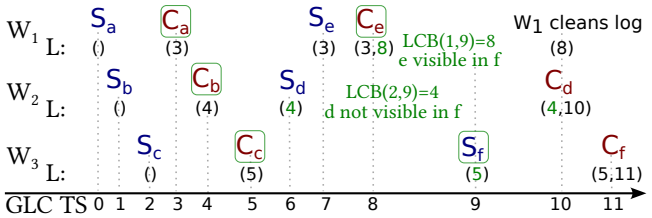


Figure 3: OSIC example. At each timestamp, a (S)tart or (C)ommit happens at a worker W_i. TX f sees a,b,c,e but not d.

Example. In Figure 3, we illustrate our protocol through an example with three workers (W1, W2, W3). Every increment of the clock (GLC) corresponds to the start or commit timestamp of a transaction that belongs to one of the workers. Whenever a transaction commits, the processing worker inserts the commit timestamp in its own commit log (L). Because the inserted timestamps are monotonically increasing in each log, the worker simply appends the new timestamp. Start timestamps are not needed in the commit log but determine the snapshot. In the figure, the first three transactions are trivial and just read an empty database. Transactions d and e both include a, b, and c in their snapshot. Starting at timestamp 9, transaction f includes all transactions from W1 that started before LCB(1,9)=8 (a and e), but only b from W2 because LCB(2,9)=4. Even after d commits at timestamp 10, LCB(2,9) will return 4, ensuring that d remains invisible for f. After d commits at timestamp 10, W1 can remove the entry 3 from its log because all future transactions will include all transactions from W1 up to 8.

Synchronization and Correctness. Snapshots must include all transactions with smaller commit timestamps and exclude the ones with larger timestamps. Without correct synchronization, it could happen that transaction f queries LCB(1,9) from W1 after e draws 8 from GLC but before e inserts it into its commit log. Thus, f would miss e, an already committed transaction, in its snapshot. To prevent race conditions and maintain scalability, each commit log is protected by its own mutex. The committing transaction exclusively locks its log before drawing the commit timestamp and releases the lock only after inserting the commit timestamp. This way, in our example, transaction f will have to wait on the W1 lock if e has drawn 8 but has not inserted it yet. Excluding future transactions is guaranteed as well because the GLC assigns them larger timestamps and LCB will skip the newly inserted one. In our example in Figure 3, LCB(2,9) returns the same 4 excluding transaction d whether we evaluate it at timestamp 9 or 10.

Minimizing Cross-Thread Communication. A snapshot's visibility is determined by its start timestamp. The LCB of other workers will return the same value whether it is queried at the beginning of the transaction or at a later point in time. Because evaluating LCB is not for free due to cross-thread communication, it makes sense to evaluate it for a certain worker only when we first encounter a version written by that worker. In addition, the returned value can be cached in case we read another version written by the same worker. Listing 1 shows pseudo code for version visibility checks under snapshot and read committed isolation.

Limiting The Commit Log Size. The maximum size of a commit log is equal to the number of workers #W. Thus, before a transaction

Listing 1: Pseudo Code for OSIC Tuple Visibility Check

```
w_i: current transaction worker id
tx_ts_start: current transaction start timestamp
sc: thread-local snapshot cache // init with zeros
   [w_i -> (cache_ts_start, w_i_ts_commit)]
bool isVisible(tuple_w_i, tuple_ts_start)
   if (tuple_w_i == w_i)
      return true // workers see their own changes
   if (read committed) // get latest snapshot
      tx_ts_start = GLC.load()
   if (sc[tuple_w_i].w_i_ts_commit > tuple_ts_start)
      return true // cache hit
   if (sc[tuple_w_i].cache_ts_start < tx_ts_start)
      // compute LCB and update cache
      last_commit_ts = LCB(tuple_w_i, tx_ts_start)
      sc[tuple_w_i] = (tx_ts_start, last_commit_ts)
   return sc[tuple_w_i].w_i_ts_commit > tuple_ts_start
```

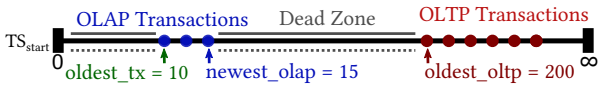


Figure 4: Separate Watermarks for OLTP and OLAP

starts, it checks if it has enough space in its fixed-size commit log for a new timestamp. If the log is full, we remove redundant entries, keeping only entries that are the LCB of an active transaction, which is implemented as follows. We collect the TS_{start} from all concurrent threads, then we find and mark the LCB commit entry in the full array for every collected timestamp. The remaining unmarked entries can be then removed from the array. Because OLTP transactions refresh their snapshot often, only few recent commit log entries will remain. Using 8-byte timestamps, even with 1024 threads the total memory consumption of all commit logs will not exceed $1024^2 * 8 = 8$MiB.

Separate High Watermarks for OLTP and OLAP. To speed up garbage collection, most MVCC systems use a single high watermark that tracks the set of transactions that are visible to *all* active transactions, i.e., the oldest snapshot. In our design and in many main-memory systems, this watermark corresponds to the oldest (smallest) TS_{start} of all current active transactions. While the single high watermark technique is cheap and easy to implement, it is imprecise and can only remove the tail of a version chain. However, by distinguishing OLTP from OLAP transactions, we can gain more pruning power using watermarks as is illustrated in Figure 4. When a long-running OLAP query prevents the oldest_tx watermark from advancing, the oldest_oltp likely keeps advancing, and we can use it to identify deleted tuples ("tombstones") that are not relevant for future OLTP transactions. We will explain in more detail how to use this idea in Section 3.2. Moreover, in Section 3.3 we use the gap between (newest_olap, oldest_oltp) to prune versions in the middle of a version chain using the vDriver Dead Zone [29] concept.

Cooperative GC using Worker LCB. We make the case for cooperative GC where each worker removes the garbage it has created. Using the watermarks, a worker w can remove all versions created by w transactions that started before LCB(w, oldest_tx or oldest_oltp). In the remaining sections, we discuss storage and GC techniques that build on this idea. Hence, we will use the shortcut *LCB(ts)* for LCB(w,ts) when w equals the same querying worker.

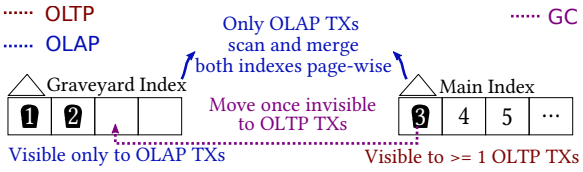


Figure 5: Moving deleted records only visible to long-running transactions to Graveyard Index

Watermark Maintenance. Our current implementation assumes that whether a transaction is OLTP or OLAP is determined by the upper levels of the systems (i.e., the query optimizer) – although it would also be possible to detect long-running transactions dynamically. The frequency of watermark updates varies from system to system. We opt for randomized and interleaved refreshing between transactions: after each transaction, we refresh with a probability of $1/\#W$. Each transaction publishes its TS_{start} in a shared global array and uses the most significant bit in the timestamp as a flag to distinguish OLTP from OLAP transactions. This way, a single iteration over the timestamps in the array is enough to calculate all the needed watermarks shown in Figure 4.

3.2 Graveyard Index for Robust OLTP

Deletion Creates Tombstones. In current MVCC systems, all keys that are visible to at least one transaction must remain in the index. When a transaction deletes a key, it marks it as logically removed but physically keeps the entry in the index for other transactions. We call such "dead" entries *tombstones*. Tombstones are opaque to the low-level index operations. Range lookups, for example, will return the first physical entry in the index whether it is a tombstone or a visible tuple. Only after the MVCC logic determines that the entry is not visible, can it ask the index to proceed to the next candidate entry. In the presence of long-running transactions, tombstones can accumulate on the hot path of operational transactions, which, as discussed in Section 2.2, can lead to performance degradation.

Observation About Tombstones. To solve the tombstone problem, we somehow have to bring the knowledge of tuple visibility to the indexes. To find the definitely-visible tuples in one logarithmic search without any manual skipping, one would have to index the begin and end timestamp beside the user-defined key using multi-dimensional indexes. However, such indexes have significant overhead and do not provide worst-case logarithmic bounds. Instead, we exploit the observation that tombstones are only visible to long-running (OLAP) transactions, which see an old snapshot. Short (OLTP) transactions, on the other hand, will refresh their snapshot frequently and therefore do not have to see most tombstones.

Graveyard Index. We propose an auxiliary data structure called *Graveyard Index*, which holds all tombstones that are not relevant to OLTP anymore. Every user-defined index has an associated Graveyard Index with the same key(s). As Figure 5 illustrates, only long-running OLAP transactions have to look into the Graveyard Index to retrieve old tuples that are deleted from the perspective of recent snapshots. Between transactions, worker threads move the tombstone from the main index to the Graveyard Index once the

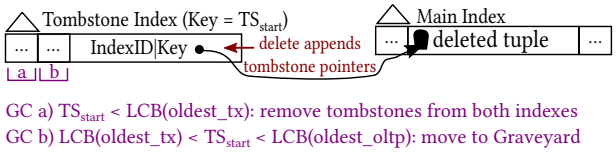


Figure 6: Tracking tombstones for garbage collection

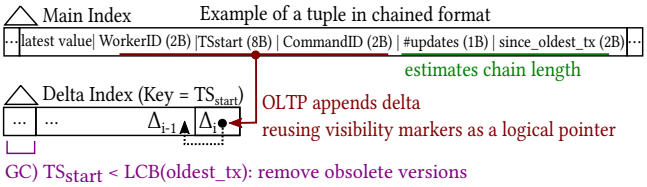


Figure 7: TX appends the before image to Delta Index

tombstone is not needed by any OLTP transaction. All known commit protocols allow deriving such kind of information – however, at different costs. With OSIC, a tombstone is moved whenever the delete commit time falls behind the *oldest_oltp* timestamp. OLTP transactions never query the Graveyard Index, which ensures that the complexity of index operations remains logarithmic in the number of *visible* tuples. This is a crucial feature of our design that makes OLTP performance robust and solves the *neworder problem*.

OLAP. OLAP queries must ensure that they are not missing any old tuple versions that have been moved to the Graveyard Index. With B$^+$-Tree indexes, this can be implemented as follows. Whenever an OLAP query enters a leaf page searching for a range of tuples, it intersects this range with the upper and lower bound of the leaf page it is scanning. The resulting range is then used to look for possibly visible tuples that are reachable through tombstones that were moved to the Graveyard Index. If relevant tombstones are found, then tuples from the main index leaf and the Graveyard Index must be merged and returned in the correct order. In practice, in large table scans only a small fraction of the tuples will have been deleted, and the Graveyard Index will usually return an empty result. Nevertheless, merging incurs some overhead for OLAP queries, but we argue that the robustness gained for operational transactions makes this worthwhile for systems that prioritize OLTP latency.

Tombstone Tracking. To move tombstones to the Graveyard Index and to remove them physically from all indexes when they are not needed any longer, we have to keep tight track of their locations. Therefore, in addition to the Graveyard Index, we need an additional data structure that acts as a "todo list" that points to the tombstones and bookmarks the TS_{start} of the transaction that created the tombstone, i.e., deleted the tuple. For this purpose, we use a per-worker append-optimized B$^+$-Tree that we call the *Tombstone Index* shown in Figure 6. To allow multiple tombstone pointers per transaction, we make keys unique by concatenating *CommandID* after TS_{start} in the key (not shown in the figure). Using the oldest_tx and oldest_oltp watermarks described in Section 3.1, we can find the tombstones that are not needed by any transaction and the ones only needed by the active OLAP ones. In the former case, we physically remove the tombstones from the main index and the corresponding pointers in the Tombstone Index. In the latter case, we move the tombstone from the main index to the Graveyard Index. However, we keep the tombstone pointer until the deleted version is not needed any more as in the former case.

3.3 Adaptive Version Storage and GC

Overview. We combine both off-row (Delta Index) and in-row (FatTuple) version storage. The default is off-row storage where old versions are stored in partitioned delta indexes using transaction TS_{start} as a key. This effectively makes the Delta Index (and also the Tombstone Index) a "per-thread GC todo list" and enables very efficient delta appends and range garbage collection. When a tuple is updated frequently, we automatically convert it to the in-row FatTuples format that stores all tuple versions in the main index. Inlining enables precise GC without risking random I/O to storage. To achieve robustness and efficiency for complex workloads we combine off-row with in-row storage and several kinds of GC.

Delta Index: Off-Row Versioning and Todo List. By default, the main index only stores the most recent tuple version. Older versions are stored in a per-worker-thread *Delta Index*, which serves as an off-row version store as well as a todo list for garbage collection. As Figure 7 illustrates, tuples in the default chaining format store some metadata besides the latest committed value: WorkerID, TS_{start} determines the visibility according to OSIC protocol rules shown in Listing 1. Together with the CommandID, they form the Delta Index key and act as a logical pointer to the delta entry for the previous version. This makes all delta inserts append operations that can be heavily optimized using pointers to the rightmost leaf. Delta Index GC translates to an efficient key range delete. Each delta entry also contains the same triple to point to its previous version (i.e., deltas are chained). CommandID distinguishes between several operations of a transaction and can be omitted in the Delta Index by using WorkerID|TS_{start}|IndexID|Key as the key. The #updates and since_oldest_tx fields are used to estimate the chain length as we will show later in this section.

Detecting Long Version Chains. Our second storage format, FatTuple, is only useful for tuples with long chains that cannot be purged using the classical watermark method because of an active long-running transaction. Hence, the chain length plays a significant role in this design. Maintaining the exact chain length counter for each tuple is not a viable option because it costs an additional index traversal to update the counter for every delta we remove from the Delta Index. A randomized approach can lead to many false positives that we only recognize after having traversed the whole chain and possibly hit evicted pages to find out the true length. We propose the following heuristic to estimate the chain length at the cost of 3 bytes per tuple. As shown in Figure 7, we store two additional fields in each tuple to estimate the chain length. The first field, #updates tracks the number of times we update the tuple while the oldest transaction in the system has not changed. The second field, since_oldest_tx stores the least significant two bytes of the current oldest transaction TS_{start}. Whenever a transaction updates a normal (chained) tuple, it checks whether the current oldest transaction TS_{start} is still equal to since_oldest_tx. If it is the case, then it increments the update counter. Otherwise, it resets it to zero. When the counter exceeds the number of workers, then we switch to FatTuple.

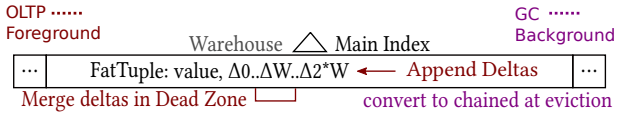


Figure 8: FatTuple format for frequently-updated tuples

FatTuple: Optimized Tuple Format for Hot Tuples. In the conversion process, we follow the chain in the Delta Index and append the required deltas to our inline format shown in Figure 8. In FatTuple, all deltas are stored next to each other in a slotted page similar format and directly in the main index at the same key but with large enough payload size. This gives us fast random access to deltas within the main index, which we need for efficient precise garbage collection, especially when we have to merge deltas that cover different attributes. Note that we only use the FatTuple format for very hot tuples. This avoids index bloat, as (by definition) only a small portion of the database can be hot. We implement on-demand precise garbage collection (OPGC) that first prunes tuples in the Dead Zones [29] maintained by the OSIC watermarks and only use all possible Dead Zones, by fetching all concurrent start timestamps, before having to enlarge the FatTuple. As we only write the TS_{start} in the tuple, we take $TS_{start} - 1$ of the next version as the TS_{commit} of the version before. The resulting reduced accuracy is negligible in practice for frequently-updated tuples.

Delta vs. Full Copy. So far, we have discussed deltas that only work for updates that do not change the tuple size and touch only few attributes. The full copy approach must also be implemented for updates that actually replace the tuple and change its length. To support such different-length updates in our design, we store full copies in the Delta Index using the same mechanism. However, in FatTuple, instead of storing the delta, we store a logical pointer (which also includes the visibility information) to the full copy in the Delta Index. Using only the version visibility information stored inline, a precise GC algorithm can decide which versions to remove without having to look into the version itself – as it would do to merge the attributes in the delta case.

Decompose FatTuple at Page Eviction. With our default chained tuple format, we tightly keep track of versions in Delta Index. The system can always find them using index range scans and hence they can be safely evicted to storage. With FatTuple, a version only exists in-lined in the tuple. There is no external logical pointer to it and the system does not bookmark FatTuples. This is a deliberate design decision because FatTuples versions are usually short-lived. However, workloads change over time and frequently-updated FatTuples may become cold. Blindly evicting FatTuples to storage can lead to a garbage leak if the evicted FatTuple is never accessed again. Thus, before we evict any page, we check whether it contains FatTuples and convert them to the chained format. This transforms versions to Delta Index entries that will eventually be GCed.

3.4 Durability and Recovery

Logging. We separate recovery from versioning and use WAL as the only source of truth after a crash. Although we store the previous version in the Delta Index, we keep the before image in the WAL entries. We also write the start and commit timestamp to the WAL. This eliminates the durability requirement for Commit Log, Graveyard Index, Tombstone Index, and Delta Index. Only the page identifiers that the buffer manager allocated for the auxiliary indexes are durable in order to truncate these indexes after a crash. FatTuple changes must be logged and recovered just like any other operation in the main indexes.

Recovery. After we determine the winning transactions in the first recovery phase (analysis), we insert the commit timestamps for the last committed transaction from every worker in its own Commit Log. While applying the redo log, delete operations for winning transactions remove the tombstones from the main index.

Transaction Abort. In our design, the Commit Log holds entries only for successfully committed transactions. Otherwise, we would not able to represent the set of committed transactions by a worker using a single timestamp anymore because of the gaps caused by aborts. Consequently, on abort, we revert all transaction changes and recycle the same start timestamp for the next transaction.

Early Lock Release. In traditional systems, concurrency control locks are only released after all WAL entries, including the commit, are written and flushed to durable storage (hardened). Only then the transaction is *signaled* as committed to the user. In skewed workloads, this may lead to lock contention that limits scalability. To avoid such log-induced lock contention, we implement the Early Lock Release (ELR) [15, 28] technique. ELR releases the write lock after inserting the commit timestamp in the Commit Log without waiting until the logs are hardened. This way, we remove the log flush latency from the critical path and reduce contention in skewed workloads. To still preserve durability and recoverability, we have to make sure that all transactions we read from are durable when we signal the commit. This corresponds to having all WAL logs of transactions with commit timestamp less than the current transactions start timestamp hardened on storage. This is implemented by introducing an additional atomic counter per worker thread ("hardened_commit_ts") that tracks the commit timestamp of the latest transaction that got its WAL hardened. The minimum from each worker thread is then tracked by a global watermark that is used to check whether it is safe to signal the transaction after ELR.

3.5 Putting Everything Together

After describing the techniques individually, let us now go back to Figure 2 and bring all pieces together.

Transaction Begin. A transaction (TX) starts by drawing a TS_{start} from the Global Logical Clock. This start timestamp fixes the snapshot, i.e., the set of visible transactions. The TX publishes the timestamp and its type (OLAP or OLTP) in a global array, so it can be used later to refresh watermarks. This finalizes the initialization phase and now TX can start processing user commands.

Update Operation. If the updated key is found in the main index then the TX has to check whether the current version v is visible. If it is not visible, TX has to abort to prevent dirty writes [12]. Using the commit log of the worker that wrote the version, we query $q = LCB(v.WorkerID, TX.TS_{start})$ and cache it to find the latest visible transaction to us from that worker thread. The current version is then visible if $q > v.TS_{start}$. If the tuple is in the default chained format, we then update the value and the chain-length estimation counters accordingly. If the length exceeds a pre-defined

threshold, e.g., number of workers, then we convert it to FatTuple. Otherwise, TX proceeds normally and inserts the delta in its own Delta Index. For FatTuple, TX appends the delta in the same index entry and triggers precise garbage collection if the FatTuple is full before enlarging it.

Transaction Commit. We first exclusively lock the worker thread commit log to force other threads to wait before calling LCB on our commit log until we finish committing. Then, we draw the commit timestamp from the Global Logical Clock and insert it in the commit log before we unlock it.

Garbage Collection. In our adaptive storage scheme, the GC techniques adapt to the user workload at the tuple level. The methods we use for deltas and tombstones of normal (chained) tuples differ from the FatTuple ones. In Section 3.3, we have discussed GC in the context of each storage format. Here, we list all the GC methods we use according to the place where we execute them:

(1) GC Between Transactions: Using watermarks and commit logs, we compute for every worker the set of transactions that are visible to all active transactions and the ones visible only to the OLTP transactions. This corresponds to $LCB(oldest\_tx)$ and $LCB(oldest\_oltp)$, as shown in Figure 2. Then, we purge all update deltas and tombstone pointers in Delta and Graveyard Indexes created by globally-visible transactions. We also remove all tombstones that the Graveyard Index entries pointed to from their main index. The remaining tombstones in the range $TS_{start} < LCB(oldest\_oltp)$ are only needed by OLAP transactions. Our Graveyard Index technique moves the latter tombstones from their relation's main index to their graveyard.

(2) GC Within Transactions: When we update a tuple, we estimate its current chain length. If it exceeds a pre-defined threshold, we shorten the chain length using precise garbage collection as described by vDriver [29] and then convert it to FatTuple. Subsequent updates on the FatTuple follow an on-demand precise garbage collection to keep the chain length short for robust OLAP scans.

(3) GC At Page Eviction: Versions in FatTuples are not tracked by the Delta Index. Thus, we avoid evicting pages with FatTuples because the workload may never touch the evicted FatTuples again, and we would end up with garbage leak on storage. On eviction, we convert FatTuples back to the default format and create Delta Index entries for each version, so they eventually get garbage collected.

4 EVALUATION

Implementation. We took the open source implementation of LeanStore [2], which had no transactions support, as a basis and implemented three variants on top of it: *LeanStore (Basic)* uses the same snapshotting and committing protocol used in well-known systems like WiredTiger [3], PostgreSQL [1] and, MySQL InnoDB [7]. For version storage, it uses our Delta Index to store versions off-row and to maintain pointers to tombstones. However, it has no Graveyard, FatTuple, or precise garbage collection capabilities. *LeanStore (NoSteal)* is almost the same as Basic but uses a no steal commit protocol that imitates main-memory protocols by writing the commit timestamp into the write set after committing. *LeanStore (Full)* combines all the techniques proposed in this paper (the OSIC protocol, Delta Index version storage, the FatTuple format, and the Graveyard Index). Both variants interleave Garbage Collection between

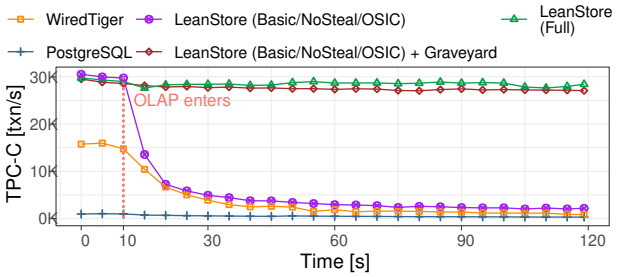


Figure 9: Graveyard technique stabilizes TPC-C with a long-running transaction. (1 TPC-C thread, 1 scan thread)

transactions and do not use any background GC threads. We also implemented per-thread write-ahead logging [27], group commit, and Early Lock Release with dependency tracking [5]. The source code is available: https://github.com/leanstore/leanstore/tree/mvcc.

Competitors. We compare against *WiredTiger* in version 10.0.0 [3] and *PostgreSQL* in version 12.9 [1]. For WiredTiger and LeanStore, we use a C++ client and the same benchmark implementation. For PostgreSQL, we use the JDBC-based client BenchBase [17, 40] and its benchmark suite. Both competing systems take snapshots by building a vector of all concurrent in-progress transaction IDs and both rely only on high watermark garbage collection. We configured all systems to run in snapshot isolation mode.

Hardware. We ran all experiments on a single-socket server with an AMD EPYC 7713 64-Core CPU (128 hardware threads) with 512 GB of DRAM. For storage, we use XFS file system on top of a RAID-0 of ten 3.8 TB Samsung PM1733 SSDs.

TPC-C + Scan: OLTP Performance. We begin by evaluating the robustness of operational transactions in mixed workloads. In Figure 1, we show that TPC-C performance collapses when a long-running transaction enters the system. In Figure 9, we run the same experiment (1 TPC-C + 1 long-running OLAP thread), but include the proposed techniques and show absolute numbers. Because only two threads are involved in this experiment, varying the commit protocol has little impact on performance – which we later evaluate separately. We therefore merge the plot lines for (Basic/NoSteal/OSIC). Only by using the Graveyard Index, we manage to retain a stable OLTP throughput. The small performance loss after OLAP enters is due to the additional work that the OLTP thread has to do to move tombstones from the main *neworder* index to its Graveyard Index. FatTuple limits the version chain length and thereby stabilizes OLAP performance but not OLTP. In the Full variant, FatTuple improves the in-memory performance slightly because it saves some Delta Index traversals for frequently-updated tuples. However, as we will see in the out-of-memory TPC-C experiment, the larger FatTuple payload can lead to higher page miss rate and outweigh the benefit of less index traversals. Note that LeanStore is not just much more robust than the competitors, even at the start of the experiment its throughput is 2×/30× faster than WiredTiger/PostgreSQL. We therefore omit the other systems from the following experiments.

TPC-C + Scan: OLAP Robustness. We demonstrate the effectiveness of the FatTuple optimization in retaining OLAP scan throughput over frequently-updated tuples using a mixed workload of 10

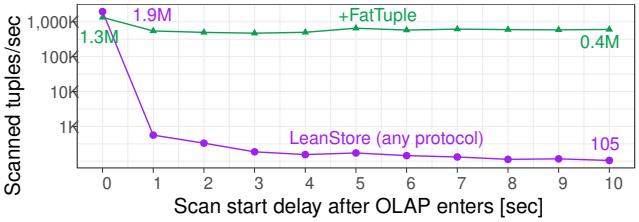


Figure 10: Scan performance with frequently-updated tuples (10 TPC-C threads, 1 scan thread)

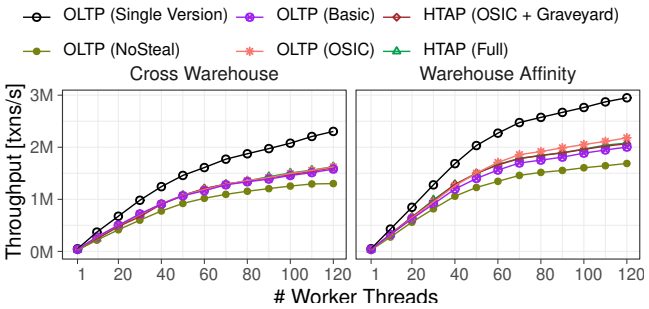


Figure 11: TPC-C in-memory (OLTP only vs. HTAP)

TPC-C threads and 1 OLAP thread. The analytical query scans other tables (or just sleeps) for a varying number of seconds then scans all tuples in the *warehouse* and *district* relations. This is an extremely challenging workload because these two tables are updated frequently by OLTP threads, and a longer delay means that many versions accumulate. The scan throughput depending on the starting time is shown in Figure 10. Scanning the two hot relations while the snapshot is fresh (t=0) is very fast. Over time, the skewed updates of TPC-C lead to longer chains that the OLAP transaction has to traverse to reach the version it got assigned when it acquired its snapshot. FatTuple mostly overcomes this issue by pruning unnecessary versions in-between. There is still a small performance gap between the ideal state (t=0) and what FatTuple delivers (t>0). Because a FatTuple is larger than normal tuples, we can fit fewer tuples on a single leaf and the OLAP scan has to traverse more leaves. However, the alternative off-row versioning would require traversing other trees and skipping invisible in-between versions to retrieve older versions, causing very low OLAP performance.

TPC-C + Scan: Scalability. We evaluate the overhead and scalability of our techniques by comparing the performance of TPC-C alone (OLTP) vs. in parallel to a long-running OLAP query (HTAP). We also compare against the Single Version implementation of LeanStore and the *NoSteal* variant that installs the commit timestamp in the write set tuples as main-memory protocols do. We omit HTAP configurations without the Graveyard Index because the TPC-C performance would drop quickly as tombstones accumulate in the neworder table. The results in Figure 11 show that the full variant (Graveyard and FatTuple) stabilizes performance while achieving almost the same throughput as in the pure *OLTP (OSIC)* case. With warehouse affinity (i.e., under very low contention), our OSIC protocol does not have to look into the commit log of other threads and hence performs better than the basic variant. With cross warehouse transactions (i.e., under moderate contention), the benefit of OSIC over the basic variant diminishes because OSIC will have to look into the commit log of other threads anyway. The variant with the NoSteal commit protocol is the slowest because it requires 1.6× more index traversals than the OSIC or Basic protocol to install the commit timestamp in the write set.

Out-of-Memory Breakdown. In the next experiment, we evaluate the impact of the proposed MVCC techniques on robustness and performance using 1 TPC-C thread and 1 long-running OLAP thread. We use a scale factor of 10 warehouses (no affinity) and a small buffer pool that only fits half of the working set needed by a single warehouse. Because the TPC-C performance deteriorates in all variants without Graveyard, we show the average measurements results of the first five seconds. The following table shows the TPC-C transaction rate, number of index traversals (op) per transaction, I/O page reads per transaction, and the resulting robustness:

	TX /s	Index Ops	Page Read	robust OLTP	robust OLAP
OSIC	1,517↓	24↑	8.9↑	N	N
OSIC+PGC [13]	1,253↓	34↑	10.9↑	N	Y
OSIC+FatTuple	1,322↓	23↑	10.3↑	N	Y
OSIC+Graveyard	1,531	26	8.9	Y	N
Full	1,330	25	10.2	Y	Y

(↓) Decrease over time (↑) Increase over time

The *OSIC+PGC* variant imitates the in-memory Steam [13] precise garbage collection technique in our buffer-managed system LeanStore. PGC achieves OLAP robustness but at a high cost because of the IO and index traversal cost of jumping back and forth between the versions in the main index and Delta Entry. Our FatTuple limits the length of version chains and stabilizes OLAP performance at a lower cost. Most of the performance loss with FatTuple is due to the larger values in the main index that lead to additional page misses in the current LeanStore implementation. A better replacement strategy could lead to better results. Enabling the Graveyard technique does indeed cost us additional index traversals to move the tombstones but not any additional page faults because the move happens shortly after its creation. As a result, it stabilizes OLTP performance and quickly outperforms the basis variant (OSIC) as it saves us the ever-increasing number of tombstones that the delivery transaction would have to skip. We believe that robustness and predictability is worth the performance cost of the full variant. Note that in a pure OLTP workload, the full variant would deliver the same performance as OSIC because Graveyard and FatTuple are triggered only in the presence of a long-running transaction.

Bulk Loading. A common workload pattern is bulk loading: a transaction inserts or updates large amount of data then commits. In such a workload, stamping the commit timestamp in the write set is way more expensive than in short transactions because the write set does not fit in the cache anymore. As the left-hand side of Figure 12 shows, when the buffer pool has just enough space to fit the entire data set, NoSteal (main-memory) protocols causes 1.5× overhead. When the data size exceeds the buffer pool, then the overhead of in-memory protocols quickly exceeds 2× for the following reason. Initially, the transaction can create new pages by consuming free memory from the buffer pool and let the background threads handle

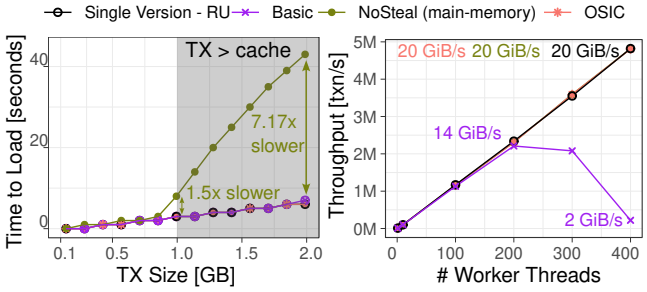
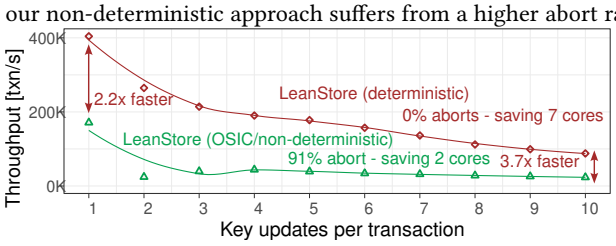


Figure 12: Left) Bulk loading Right) Out-of-memory rnd. read

eviction. However, the created pages are now evicted to storage and the transaction now has to synchronously read the page first from SSD before writing the commit timestamp.

Out-of-Memory Key/Value. Modern NVMe SSDs support millions of IOPS, but their read latency is still on the order of 100 microseconds. This means that to get good throughput one needs to perform many concurrent transactions. We evaluate out-of-memory scalability using a key/value random point lookup workload and a large number concurrent short transactions. The results on the right side of Figure 12 show that OSIC scales linearly and very close to the single-version read uncommitted variant. With 400 threads, it is able to maintain 20 GiB/s read bandwidth from storage. The classical commit protocol, on the other hand, struggles with the high degree of concurrency. Its performance stops scaling and starts to deteriorate at about 300 threads because of the increasing number of L1-cache misses it has to pay to build its snapshot.

Deterministic Execution Under Contention. In our last experiment, we compare non-deterministic with deterministic execution under high contention. We use 10 threads and a highly skewed workload with multiple key updates per transaction. We implement a deterministic execution layer on top of LeanStore where the set of keys that a transaction updates is known a priori at the start of each transaction. The deterministic layer latches the B-Tree leaves that contain the keys in ascending order before starting the transaction. This makes the competing transactions gracefully wait on the low-level synchronization mutexes, which effectively serializes the transactions. As the following figure shows, under high contention our non-deterministic approach suffers from a higher abort rate:



The resulting throughput is 2.2-3.7× slower than what the deterministic model is able to achieve. In the non-deterministic model, the more keys a transaction has to update, the more likely it is to encounter a yet invisible latest version that forces it to abort. Moreover, the deterministic approach occupies only 3 cores out of 10 instead of 8 cores because threads are put to sleep in turn at the mutexes instead of retrying aggressively. Replacing OSIC with Basic or NoSteal protocol yields the same results.

5 RELATED WORK

MVCC is a well-studied topic [8, 16, 18, 20, 22, 31, 32, 32, 45, 48, 50–52], with much of the work focusing on *peak* performance.

5.1 Snapshot Isolation Commit Protocols

The *PostgreSQL* [1, 46] community acknowledged the limitations of their current snapshotting protocol and proposed an alternative based on Commit Sequence Number (CSN) [4]. CSN orders snapshots and commits to avoid constructing an array of in-progress transaction IDs (XID). This is done using a central spooled ring buffer that maps XID to CSN. However, this never made it into production and remains an interesting proof of concept that reinforces the need for better snapshotting protocols in disk-based systems.

To realize atomic commits, in-memory systems implement an indirection that maintains the state of the transaction until it aborts or installs the commit timestamp in the write set. *SAP HANA* [33] uses a pointer from the written tuples to the in-memory transaction state object. *Hekaton* [16, 32] uses a central hash table to store the state (active, preparing to commit, committed, or aborted) and possibly the commit timestamp for all in-flight transactions. Not only can such a centralized data structure become a contention hot spot, it cannot fully substitute touching the write set. Its size would grow indefinitely as the system processes more transactions. OSIC, in contrast, keeps the memory footprint bounded by exploiting worker identifiers and the Transitive Commit Invariant.

Umbra [22] avoids re-timestamping for large write transactions by extending Hyper [39] with an additional single-writer mode that uses a central atomic counter to mark the bulk transaction as committed for future transactions. Our design solves the problem without restricting the concurrency of write transactions.

5.2 Garbage Tracking and Tombstones

NoisePage schedules garbage collection tasks to maintain its physical data structures using an in-memory Deferred Action Framework (DAF) [53]. DAF tags tasks such as tombstone removal and obsolete version cleaning by a timestamp and executes them once the smallest start timestamp of active transactions is larger than the tagged timestamp. Our Tombstone and Delta Index are based on similar ideas. The difference is that we track the smallest start timestamp of short-running OLTP and long-running OLAP separately and trigger different actions depending on who still need the tombstone. Moreover, we use buffer-managed B^+-Trees that support range queries to find ready-to-execute actions.

MySQL/InnoDB [7] stores versions in undo log records in buffer-managed slotted pages. Tuples in the main index contain 7-bytes logical pointers to previous versions. Once a transaction commits, InnoDB appends pointers to all generated undo records to a History List, which is used a todo list by background purge threads. Our Delta Index is more efficient as its key is part of the visibility information (start timestamp and worker id) we store in the tuple. Using B^+-Tree range scan over Delta Index, we can find the obsolete versions without having to maintain an additional todo list.

Umbra [22] uses per-page hash tables to link tuples with their version chains. These tables are allocated outside the buffer pool to avoid persisting temporary versions. This design has low in-memory performance overhead but assumes that concurrent OLTP

transactions fit entirely in memory – requiring additional budgeting for the hash table memory consumption. For larger-than-memory transactions, an additional optimization is needed to avoid physical memory allocation. Our design handles larger-than-memory workloads transparently, relying solely on buffer-managed data structures. We append versions in per-worker Delta Indexes built on an optimized B$^+$-Tree [5]. Large bulk loading transactions are transparently handled by the replacement strategy. For short transactions, older versions in indexes will get pruned before the buffer manager evicts them and recent versions are accessed in near in-memory performance.

5.3 Precise Garbage Collection

vDriver [29] is a version management architecture that brings precise garbage collection to disk-based systems. To identify obsolete versions, vDriver uses the *Dead Zone* concept, which we also use in FatTuple. Dead Zones are time ranges between the start of two consecutive transactions. A version is considered *dead* and can be reclaimed when its visibility starts and ends in one of the Dead Zones. In Section 3.1, we propose maintaining a single global Dead Zone between the newest OLAP and oldest OLTP which we use in our FatTuple OPGC heuristic.

vDriver and its successor Diva [30] require substantial changes in the storage scheme. Their Single In-row Remaining Off-row (SIRO) versioning places the most recent and first oldest versions in-row close to each other to accelerate recovery while pushing the remaining versions to an off-row version store. The off-row storage uses mixture of in-memory and buffer-managed data structures for metadata maintenance and version clustering. Our FatTuple data structure is a simpler and can be integrated into any system.

The main-memory systems *HANA* [33] and Hyper with Steam [13, 39] use a precise (interval-based) garbage collection technique for purging all tuple versions that are not required by any transaction. However, both rely on fast in-memory version chain traversal.

5.4 Alternative HTAP Techniques

In LeanStore, we use a unified row-store storage engine for HTAP workloads. There is another body of work that examines column-stores for HTAP [36, 44] or a combination of both [9, 24].

L-Store [44] proposes a lineage-based columnar storage format for real-time analytics in an unified storage engine. OLTP transactions append changes in write-optimized pages that L-Store consolidates into read-optimized pages for fast analytics in a contention-free and transactional manner. While the L-Store format delivers superior analytical performance, in the presence of a long-running transaction it is still vulnerable to the performance anomalies that this paper addresses: entries to dead RIDs will remain in the indexes, slowing down OLTP; and without precise garbage collection version chains can grow beyond need, slowing OLAP down. We plan to investigate adopting the L-Store columnar format together with our robustness techniques for better OLAP performance.

There are also systems with separate types of storage engines. *BatchDB* [38] uses primary-secondary replica to isolate OLTP from OLAP. It processes analytical queries in a batch on the same snapshot before applying recent changes from the OLTP primary to reach a new snapshot. Long-running queries do not run on the OLTP engines and tombstones will not accumulate on the hot path eliminating the need for a Graveyard index. However, if more than one snapshot is allowed to exist in the OLAP engine, then our FatTuple approach becomes applicable to detect unnecessary long version chains. The isolation achieved in this model comes at the cost of keeping the analytics replicas in sync with the primary. Our design keeps OLTP safe from degradation while maintaining all data in a single storage engine.

Wildfire [10, 11] extends Spark with stateful engines that handle both transactional and analytical queries. At a fixed interval, Wildfire stages data from the local SSDs of the transactional engines to a shared file system, which is then accessed by dedicated stateless analytical Spark engines. Analytical queries with strict freshness requirements must run on the same stateful engines, jeopardizing OLTP performance as shown in Section 2.2.

Deterministic Systems. LeanStore uses a cursor-oriented transaction model with no assumptions on the user workload. Deterministic engines, on the other hand, exploit a priori knowledge of read/write-sets to unlock more performance potential. BOHM [20] determines the serialization order and creates the necessary versions for transactions prior to the execution phase. QueCC [42] improves multi-core scalability by parallelizing the planning and execution phases using thread-to-queue assignment and priority-based planning. Unlike LeanStore, deterministic systems can make their writes visible before committing [19]. What deterministic approaches have in common is that they work on batches of transactions. For short-running OLTP transactions, this constraint is not a problem. OLAP queries, on the other hand, can take a long time, in particular in out-of-memory systems such as LeanStore. Thus, a single long-running OLAP query may effectively halt OLTP processing for a long time. Multi-versioning in the mentioned deterministic schemes maintains the previous version only for transactions within the same batch. This therefore does not solve the performance problems of long-running, across-batch queries.

Serializability. In this paper, we only consider snapshot isolation. While this is a relatively high isolation level – the default for most production-grade systems is only read committed [14] – serializability would clearly be even better. Modular techniques such as Serializable Snapshot Isolation [21, 41], Serial Safety Net [49], Precision Locking [39], Timestamp Range Conflict Detection [37], and graph-based certifiers [26, 43] can be used on top of our design.

6 SUMMARY

We have shown that porting existing disk-based MVCC designs to high-performance storage engines leads to limited scalability and severe performance cliffs in mixed workloads. Designing a robust, scalable, and efficient MVCC system for out-of-memory engines requires rethinking the SI commit protocol, storage layer, and garbage collection. The OSIC commit protocol provides cheap commits, scalable transaction processing, and enables precise garbage collection. Together with an adaptive storage layer, our design leads to robust performance on complex workloads.

ACKNOWLEDGMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project 447457559.

REFERENCES

[1] 2021. PostgreSQL. https://github.com/postgres/postgres/releases/tag/REL_12_9.

[2] 2022. LeanStore - A High-Performance Storage Engine for Modern Hardware. https://leanstore.io/.

[3] 2022. WiredTiger Storage Engine. https://source.wiredtiger.com/10.0.0/index.html.

[4] Ants Aasma. 2021. Proposal for CSN based snapshots. https://www.postgresql.org/message-id/CA%2BCSw_tEpJ%3Dmd1zgxPkjH6CWDnTDft4gBi%3D%2BP9SnoC%2BWy3pKdA%40mail.gmail.com.

[5] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW*.

[6] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*.

[7] Oracle and/or its affiliates. 2021. MySQL 8.0 InnoDB. https://github.com/mysql/mysql-server/blob/8.0/storage/innobase/trx/trx0purge.cc

[8] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. In *CIDR*.

[9] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Janus: A Hybrid Scalable Multi-Representation Cloud Datastore. *IEEE Trans. Knowl. Data Eng.* (2018).

[10] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, René Müller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam J. Storm, Yuanyuan Tian, Pinar Tözün, Daniel C. Zilio, Matt Huras, Guy M. Lohman, Chandrasekaran Mohan, Fatma Özcan, and Hamid Pirahesh. 2017. Evolving Databases for New-Gen Big Data Applications. In *CIDR*.

[11] Ronald Barber, Vijayshankar Raman, Richard Sidle, Yuanyuan Tian, and Pinar Tözün. 2019. Wildfire: HTAP for Big Data. In *Encyclopedia of Big Data Technologies*.

[12] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*.

[13] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *PVLDB* 13, 2 (2019), 128–141.

[14] Chaoyi Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D. Bond, and Yang Wang. 2023. Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases. In *CIDR*.

[15] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*.

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*.

[17] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.

[18] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Found. Trends Databases* 8, 1-2 (2017), 1–130.

[19] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *PVLDB* 10, 5 (2017), 613–624.

[20] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *PVLDB* 8, 11 (2015), 1190–1201.

[21] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* (2005).

[22] Michael J. Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems. *PVLDB* 15, 11 (2022), 2797–2810.

[23] Andres Freund. [n.d.]. Improving Postgres Connection Scalability: Snapshots. https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/improving-postgres-connection-scalability-snapshots/ba-p/1806462

[24] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. 2010. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB* 4, 2 (2010), 105–116.

[25] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[26] Thanasis Hadzilacos. 1988. Serialization Graph Algorithms for Multiversion Concurrency Control. In *PODS*.

[27] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*.

[28] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *PVLDB* 3, 1 (2010), 681–692.

[29] Jong-Bin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived Transactions Made Less Harmful. In *SIGMOD*.

[30] Jong-Bin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *SIGMOD*.

[31] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*.

[32] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.

[33] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *SIGMOD*.

[34] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *ICDE*.

[35] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* (2019).

[36] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *PVLDB* 14, 4 (2020), 534–546.

[37] David B. Lomet, Alan D. Fekete, Rui Wang, and Peter Ward. 2012. Multi-version Concurrency via Timestamp Range Conflict Management. In *ICDE*.

[38] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD*.

[39] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*.

[40] Andrew Pavlo. 2023. BenchBase (formerly OLTPBench) is a Multi-DBMS SQL Benchmarking Framework. https://github.com/cmu-db/benchbase

[41] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *PVLDB* 5, 12 (2012), 1850–1861.

[42] Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *Middleware*.

[43] Stephen Revilak, Patrick E. O'Neil, and Elizabeth J. O'Neil. 2011. Precisely Serializable Snapshot Isolation (PSSI). In *ICDE*.

[44] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-time OLTP and OLAP System. In *EDBT*.

[45] Mohammad Sadoghi and Spyros Blanas. 2019. *Transaction Processing on Modern Hardware*. Morgan & Claypool Publishers.

[46] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*.

[47] Hironobu SUZUKI. 2023. The Internals of PostgreSQL. https://www.interdb.jp/pg/pgsql05.html

[48] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*.

[49] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.* 26, 4 (2017), 537–562.

[50] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792.

[51] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220.

[52] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD*.

[53] Ling Zhang, Matthew Butrovich, Tianyu Li, Andrew Pavlo, Yash Nannapaneni, John Rollinson, Huanchen Zhang, Ambarish Balakumar, Daniel Biales, Ziqi Dong, Emmanuel J. Eppinger, Jordi E. Gonzalez, Wan Shen Lim, Jianqiao Liu, Lin Ma, Prashanth Menon, Soumil Mukherjee, Tanuj Nayak, Amadou Ngom, Dong Niu, Deepayan Patra, Poojita Raj, Stephanie Wang, Wuwen Wang, Yao Yu, and William Zhang. 2021. Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems. In *CIDR*.