Focus! Fast On-disk Concurrency-control Using Sketches

DEUKYEON HWANG, University of Washington, USA
ALEX CONWAY, Cornell Tech, USA
CARLOS GARCIA-ALVARADO, CruxOCM, USA
JUN YUAN, Datadog, USA
NAAMA BEN-DAVID, Technion, Israel
ROB JOHNSON, VMware Research by Broadcom, USA
ADRIANA SZEKERES, Microsoft Research, USA

Concurrency-control (CC) mechanisms are essential for ensuring consistency in large-scale key-value stores, but traditional approaches face significant challenges. Mechanisms like 2PL and OCC incur high CPU overheads. Timestamp-based mechanisms are faster but require storing timestamps for every key, resulting in substantial space overhead and numerous I/O operations in disk-based systems. We address these challenges by decomposing timestamp-based CC schemes into two components: a timestamp storage system and a CC protocol. We then show that the timestamp storage system can approximate timestamps for keys not used by ongoing transactions, substantially reducing memory requirements and I/O while maintaining correctness for various protocols (STO, MVTO, and TicToc). We introduce FPSketch, an approximate timestamp storage system, and our evaluation with SplinterDB shows that FPSketch outperforms 2PL and OCC by up to 14× on some workloads and disk-based CC systems by up to 5.9×. Remarkably, FPSketch with just 32KiB of memory yields performance comparable to an idealized in-memory implementations in our evaluation. FPSketch makes timestamp-based concurrency control mechanisms practical for disk-based key-value stores.

CCS Concepts: • **Information systems** → **Data structures**; **Database transaction processing**.

Additional Key Words and Phrases: Timestamp-based Concurrency Control (CC), Approximate Timestamp Storage, Disk-based Key-Value Store

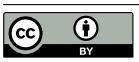
ACM Reference Format:
Deukyeon Hwang, Alex Conway, Carlos Garcia-Alvarado, Jun Yuan, Naama Ben-David, Rob Johnson, and Adriana Szekeres. 2025. Focus! Fast On-disk Concurrency-control Using Sketches. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 328 (December 2025), 27 pages. https://doi.org/10.1145/3769793

1 Introduction

Large key-value stores are the basis of many applications, such as databases and file systems. Many such applications must support not only single-key operations, but must also make sequences of operations atomic. As a result, many state-of-the-art key-value stores provide transactional semantics [9, 13, 17, 18, 29, 30, 36].

Concurrency control (CC) mechanisms govern how transactions interact with each other and ensure consistency is maintained in the kv-store. The choice of the CC mechanism can highly influence the performance of the overall system, as synchronization and memory overheads, as well

Authors' Contact Information: Deukyeon Hwang, deukyeon@cs.washington.edu, University of Washington, Seattle, WA, USA; Alex Conway, me@ajhconway.com, Cornell Tech, New York City, NY, USA; Carlos Garcia-Alvarado, carlos@cruxocm.com, CruxOCM, Santa Rosa, CA, USA; Jun Yuan, jun.yuan@datadoghq.com, Datadog, New York City, NY, USA; Naama Ben-David, bendavidn@technion.ac.il, Technion, Haifa, Israel; Rob Johnson, rob@robjohnson.io, VMware Research by Broadcom, Palo Alto, CA, USA; Adriana Szekeres, aszekeres@microsoft.com, Microsoft Research, Redmond, WA, USA.



ACM 2836-6573/2025/12-ART328
https://doi.org/10.1145/3769793

as the abort rate, all depend upon it. Therefore, significant research effort has gone into developing good concurrency-control mechanisms [8, 16, 22, 35, 45, 47, 50, 51].

An ideal CC mechanism would have low CPU overhead, low space overhead, and low abort rate. Achieving all three has been challenging. For example, two-phase locking (2PL) makes use of locks for items accessed by transactions. This can be memory-efficient; locks are needed only for keys accessed by ongoing transactions, and thus the memory footprint is proportional to the total size of the active transactions. However, holding all these locks for the entire duration of the transaction can limit throughput in high-concurrency scenarios.

In contrast, timestamp-based schemes, such as Strict Timestamp Ordering (STO) [4], Multi-Version Timestamp Ordering (MVTO) [37], and TicToc [51], can be extremely effective at avoiding CPU bottlenecks and reducing abort rates. Unfortunately, timestamp-based CC mechanisms typically have a larger memory footprint than, e.g. 2PL, because they maintain timestamps for each tuple in the database, whereas 2PL needs locks only for keys in active transactions. Moreover, timestamps are typically embedded within the database tuples, which can lead to extra I/O in disk-based databases. For example, in STO, every tuple access must read the old timestamps and write updated timestamps to the database. This means that reading a tuple also requires writing that tuple, and blindly overwriting a tuple requires reading the old value first. As a result, a CC mechanism that maintains per-tuple metadata can incur high I/O cost and substantially reduce throughput in a disk-based database. Consequently, timestamp-based CC mechanisms are rarely used in on-disk databases.

This leaves disk-based databases with a vexing trade-off. 2PL has small memory footprint but high CPU overhead, whereas timestamp based mechanisms have large storage and I/O overheads. Unfortunately, CPU overheads cannot be ignored in modern disk-based databases based on fast SSDs and modern data structures, such as LSM trees [31] and B^{ε}-trees [6].

Our contributions. We show how to overcome the memory footprint overhead of timestamp-based CC mechanisms, unlocking the ability to use them in on-disk databases. As a result, we obtain new CC mechanisms that offer up to 14× greater goodput than commonly used CCs, such as 2PL and OCC. Specifically, we:

- Show how to decompose timestamp-based CC mechanisms into two parts: a timestamp storage system and a CC protocol.
- Define ***approximate timestamp storage***, which identifies exactly when and how the storage system can approximate timestamps without affecting the correctness of the CC protocol.
- Implement a general approximate timestamp-storage data structure, which we call FPSketch.
- Combine FPSketch with three CC protocols: STO, MVTO, and TicToc.
- Evaluate the resulting CC mechanisms in SplinterDB, a state-of-the-art kv-store, and compare them against traditional CC mechanisms, such as 2PL and OCC.
- Show that FPSketch enables modern timestamp-based CC mechanisms, such as TicToc, to offer up to 14× greater goodput than 2PL and OCC in an on-disk database while maintaining a small memory footprint.

The key observation is that the timestamp storage system must provide precise timestamps only of keys in use by on-going transactions. For all other keys, it can provide merely upper bounds on their timestamps. Thus we can use a hashtable to store timestamps of in-use keys and a sketch data structure for all the other keys in the database, greatly reducing the memory required to store timestamps. The system can then use any eviction policy for moving inactive keys from the hashtable to the sketch. We call this hashtable-sketch combo an FPSketch. Interestingly, both components are required for correctness. See Section 4.

The approximations made by the sketch will sometimes induce aborts by the CC protocol. Thus the goal is to size the hashtable and the sketch to achieve both low memory usage and low abort rate. There are two ways to control this trade-off: the hashtable eviction policy and the size of the sketch. The eviction policy governs how precisely the storage system remembers timestamps of recently accessed keys. For example, an aggressive eviction policy could evict keys as soon as they become inactive, minimizing memory used by the hashtable but possibly increasing abort rates due to approximations in the sketch. A lazy policy, such as LRU or CLOCK, would allow the hashtable to store precise timestamps of recently active keys, increasing memory usage from the hashtable but potentially reducing the abort rate. Similarly, we can trade off memory and abort rate by adjusting the size of the sketch. A larger sketch will have better approximations and hence induce fewer aborts.

Implementation and evaluation. We implemented two FPSketch variants that represent external points in the design-space described above. In Focus-Counter, we use the smallest possible sketch—just a single counter that upper bounds the timestamps of all keys not in the hashtable—and evict keys from the hash table lazily using CLOCK. In Focus-Sketch, we use the smallest possible hashtable—we evict keys as soon as they become inactive—and use a variant of count-min sketch [12] for evicted keys. In our experiments, both schemes use the same total amount of memory.

We compose FPSketch with three timestamp-based mechanisms; the well-known Strict Timestamp Ordering (STO) [4] mechanism, Multi-Version Timestamp Ordering (MVTO) [37], a timestamp ordering multi-version concurrency control (MVCC) variant, and a newer optimistic CC mechanism called TicToc [51].

We also implemented disk-based and RAM-based exact timestamp storage systems as baselines. The disk-based timestamp storage scheme enables us to measure the I/O savings from approximation. The RAM-based scheme is not practical for large databases, but it enables us to measure the impact of approximation on abort rate. We benchmark the resulting CC mechanisms on SplinterDB [11], a highly concurrent write-optimized kv-store for modern storage.

Our main result is that approximate timestamp storage can deliver substantial performance gains by enabling us to use advanced concurrency-control mechanisms, such as TicToc and MVTO, in on-disk databases. For example, in our experiments, TicToc with approximate timestamp storage (both Focus-Counter and Focus-Sketch) outperformed 2PL and OCC by up to 14× on some workloads, and was never slower. In fact, TicToc with approximate timestamp storage was the fastest (or tied for fastest) of all the concurrency-control mechanisms in all but one of our benchmarks, where MVTO with approximate timestamp storage was the fastest.

Second, approximate timestamp storage is essential to achieving the performance of modern concurrency-control mechanisms in on-disk databases. For example, TicToc with approximate timestamp storage was up to 5.9× faster than TicToc with exact timestamps stored on disk. STO and MVTO showed 1.8× and over 160× between approximate and disk-based timestamp storage, respectively.

Furthermore, for STO and TicToc, using FPSketch of only 32KiB yields nearly the same performance as an idealized, *in-memory*, implementation that keeps, in RAM, timestamps for every record in the database (see Figure 1).

Finally, we find that, although both Focus-Counter and Focus-Sketch substantially outperform other CC mechanisms, Focus-Sketch offers more robust performance. In particular, our Focus-Sketch significantly outperforms Focus-Counter on several workloads and is never significantly slower.

2 Background

2.1 Timestamp-Based Concurrency Control

Timestamp-based CC mechanisms employ timestamps to order transactions. At a high level, each transaction is assigned a timestamp that indicates when it serializes in relation to all other transactions; a transaction T_1 with a timestamp $t_1 < t_2$ will generally be serialized before a transaction T_2 with timestamp t_2.

A subclass of timestamp-based mechanisms, which we focus on in this paper, uses *per-version read and write timestamps* to detect conflicts. These timestamps correspond to the timestamp of the most recent transaction that read or wrote the key, respectively. Using both types of timestamps can offer higher concurrency than using a single write timestamp because it allows multiple transactions to read simultaneously. To maintain correctness, a transaction T must satisfy the following conditions in order to commit: (1) for every key k accessed by T, $k.write\_timestamp < T.timestamp$ and (2) for every key k in T's write set, $k.read\_timestamp < T.timestamp$. The process of checking whether these conditions are met is known as *validation.* A key feature of read/write timestamp-ordering mechanisms, which we leverage in this work, is that it is sufficient to maintain *upper bounds* on the read and write timestamps of keys that are not involved in any current transactions. An overinflated read or write timestamp may cause unnecessary, or *spurious* aborts, but will not jeopardize the consistency of the kv-store.

Timestamp-based CC schemes differ in when and how they assign timestamps to transactions, when the validation is done, and when updates are made to the keys that are being written, among other things. In this paper, we focus on three variants in particular, known as *Strict Timestamp Ordering (STO)* [4], Multi-Version Timestamp Ordering (MVTO) [37], and TicToc [51], as three examples of how approximate timestamp storage can be used. Below, we briefly outline how each of these variants works.

Strict Timestamp Ordering (STO).. In STO, each transaction is assigned a timestamp at the beginning of its execution by incrementing a global timestamp counter. Transactions are pessimistic; the first time a transaction T accesses a key k, T attempts to acquire k's lock, waiting on the lock if it's already taken (the locks are necessary for a correct multi-threaded implementation of STO). Once a lock is acquired for a key k, T checks whether its timestamp satisfies the conditions outlined above with respect to k's timestamps. If not, T releases all locks and aborts. Otherwise, it continues. Transactions do not release locks until they finish, although a transaction may need to check the timestamps of a key more than once if it accesses the same key via both a read and a write. To commit, T applies its changes to the data, updates the read timestamps of the keys in its read set and the write timestamps of the keys in its write set to its own timestamp, and releases all locks.

Multi-Version Timestamp Ordering (MVTO).. MVTO is considered to be the original MVCC protocol [49]. MVCC is useful for workloads that run long read-heavy transactions, since it allows readers to freely access old snapshots of the database. However, maintaining a history of updates to the records (versions), can be expensive in storage, memory, and computation for garbage collection and for finding the correct version for a transaction to access) [34].

Like STO, MVTO serializes transactions according to the timestamps assigned at the beginning of transactions, from a global timestamp counter and conflicts are handled in the same way.

TicToc. TicToc, in contrast, is an optimistic concurrency control (OCC) scheme, and dynamically chooses timestamps for its transactions during validation. A transaction T first reads all items in its read set without taking any locks. Then, during its validation phase, TicToc reads its read set, locks its write set, and assigns T a timestamp that is (1) larger than the read timestamp of each data item in T's write set and (2) larger or equal to the write timestamp of each item in T's read set. It then

verifies that no conflicting transactions have committed since it chose its timestamp. This approach minimizes the time between choosing a timestamp and actually committing, thereby minimizing aborts.

2.2 External-Memory Storage Systems

In this section, we explain hardware and software trends that have affected disk-based databases the last few decades. The high-level take-aways are (1) concurrency has become more important due to increasing core counts and I/O parallelism in solid-state storage devices, (2) transaction overheads can be more prominent as storage device speeds have increased more than CPU and RAM speeds, and (3) modern storage data structures have performance characteristics that must be considered when designing CC mechanisms.

Solid-state drives now support millions of I/Os per second and gigabytes/second of throughput, whereas spinning disks typically top out at around a thousand IOPS and a few hundred megabytes/second of bandwidth [44]. However, extracting the full performance of solid-state storage requires issuing multiple I/Os concurrently (i.e., maintaining a high "queue depth") [44]. If given only one I/O at a time, most drives support less than 100,000 IOPS, sometimes as few as 10,000 IOPS [44].

Over the last two decades, CPUs have gone from single cores to dozens, but the instructions executed per second on a single core has remained relatively flat [39].

Together, these two facts mean that supporting a high degree of concurrency has become critical for disk-based database systems. Furthermore, I/O speeds have gone up more than CPU speeds. For example, SSDs are about three to four orders of magnitude faster than spinning disks, in terms of IOPS, whereas today's multicore CPUs have increased total instruction throughput (across all cores) by only one to three orders of magnitude compared to single-core chips of the early 2000s [39]. This means that I/O cannot mask concurrency-control overheads to the same degree as 20 years ago.

Over the same time period, the data structures used for on-disk storage have also undergone a revolution due to the advent of ***write-optimized data structures***, such as log-structured merge trees (LSMs) and B$^\varepsilon$-trees. For example, mapped B$^\varepsilon$-trees, used in SplinterDB [11], can ingest data orders of magnitude faster than older indexing structures, such as B-trees [11], while supporting queries essentially as fast as in a B-tree. Consequently, modern storage systems can ingest new data much faster than they can query it. For example, on some benchmarks, SplinterDB could perform more than 2M uniformly random insertions per second, compared to only about half a million uniformly random queries per second. This is sometimes called the query/insert asymmetry [3], and is consistent with theoretical lower bounds on the I/O costs of inserts and queries [25], suggesting that this could be an enduring property of storage system performance.

The query/insert asymmetry is important for timestamp-based schemes because, if a timestamp-based mechanism stores timestamps in a write-optimized structure, then it may need to query for the timestamps of each key written by a transaction, inadvertently piggy-backing a (slow) query onto every (fast) write performed by the transaction, disproportionately impacting overall throughput.

Finally, we note that it is not always practical to store all timestamps in memory. For example, Facebook reports that many of its workloads have kv-pairs of less than 100 bytes in size [2, 7], so storing two 8-byte timestamps for each kv-pair, as required in STO and TicToc, would take about 16% as much space as the data itself. However, Facebook also reports that many of its RocksDB instances have a RAM-to-disk ratio of less than 5%, sometimes as little as 3% [7]. Even in other scenarios where kv-pairs are larger (reducing the relative overhead of timestamps) or RAM is more

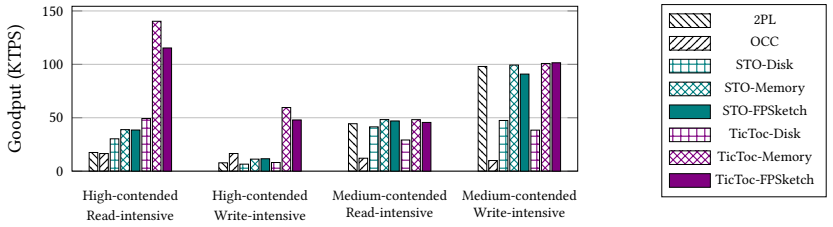


Fig. 1. Highlighting goodputs for various YCSB workloads with CC algorithms on top of SplinterDB.

plentiful, timestamps can still eat up a substantial portion of RAM that could otherwise be used for caching or other purposes.

2.3 Count-Min Sketch

The count-min sketch (CMS) [12] was originally proposed as a data structure for providing approximate counts of the occurrences of each item in a data stream. The CMS maintains a two-dimensional table A of height h and width w. Upon observing an item x in the stream, the CMS increments $A[i][h_i(x)]$ for each row i, where h_i are all hash functions. The CMS estimates the number of occurrences of an item x as $CMS[x] = min_i A[i][h_i(x)]$. The CMS obviously never underestimates the count of an item, and guarantees that

$$\Pr[CMS[x] \leq c_x + n/w] \geq 1 - e^{-d},$$

where c_x is the true count of x and n is the total number of observations added to the CMS.

3 Motivation

In this section, we present warm-up benchmarks to illustrate the promise and pitfalls of timestamp ordering for on-disk databases.

To demonstrate that timestamp ordering is a promising CC mechanism for on-disk databases, we implement STO-Memory and TicToc-Memory, versions of STO and TicToc that store all timestamps in RAM, and show that they often outperform 2PL and OCC, CC mechanisms commonly used for on-disk databases. Unfortunately, as described above, storing all timestamps in RAM is not practical. To demonstrate the potential pitfalls of using timestamp ordering with an external-memory database, we implement STO-Disk and TicToc-Disk, which store timestamps in SplinterDB, and show that the I/O of accessing timestamps can lead to poor performance. Finally, we preview results from STO-FPSketch and TicToc-FPSketch, which use our proposed approximate timestamping technique, and show that they have performance essentially as good as the in-memory versions while using much less memory. We describe FPSketch in Section 5.

All the CC mechanisms run on SplinterDB. We implement 2PL with the no-wait policy and Kung-Robinson OCC (OCC), commonly employed in disk-based databases. We use a transactional version of YCSB to generate transactions. Detailed information about the experimental setup is described in Section 7.1. In Figure 1, we highlight the goodput of committed transactions when utilizing 120 threads for various workloads.

A takeaway from this evaluation is that reading timestamps is a significant I/O overhead. Our disk-based timestamp-ordering mechanisms store timestamps for each record as part of the record. Thus, whenever a transaction reads a record, we get the timestamps "for free", i.e., without additional I/O. However, whenever a transaction writes a record, SplinterDB simply places that record in an in-memory structure called the memtable—it does not read the old version of that record. Thus, to obtain the timestamp of the old record, we need to perform additional I/O. As a result, storing

timestamps on disk can be particularly expensive for write-intensive workloads. This is apparent in Figure 1, where the gap between the Disk and Memory versions of STO and TicToc are larger for the write-intensive workloads than the read-intensive workloads.

The next major takeaway is that, when timestamps are stored in RAM, timestamp ordering schemes can match or substantially outperform 2PL and OCC. TicToc-Memory stands out in particular, outperforming every other mechanism in every workload we benchmarked. Unfortunately, storing timestamps in memory is not practical for large databases.

These results demonstrate the potential for a timestamp ordering mechanism that stores timestamps compactly in memory. Such a scheme should be able to get the performance benefits of timestamp ordering while avoiding the I/O costs of keeping timestamps on disk. Figure 1 shows that that is exactly what the schemes proposed in this paper, STO-FPSketch and TicToc-FPSketch, accomplish.

4 Approximate Timestamp Storage

While timestamp-based CC mechanisms assume the guarantees of a *map* for storing their read and write timestamps, where a *get* returns the latest *put*, here we show that for STO, MVTO, and TicToc such strong guarantees are not necessary to ensure serializability. Specifically, we show that an *approximate timestamp storage* system with the following two properties allows all three CC mechanisms to function correctly without any algorithmic changes.

Property 1. *The approximate timestamp storage system can return an* overapproximated *value for the timestamps it stores, only if there is no on-going transaction that has previously accessed the respective timestamps.*

Property 2. *When storing pairs of timestamps (read and write timestamps), if the CC mechanism stores only pairs for which their write timestamp is not larger than the read timestamp, then the approximate timestamp storage ensures that, for every pair of (overapproximated) timestamps it returns, the write timestamp is not larger than the read timestamp.*

STO assigns a serialization time τ to each transaction when the transaction begins (see Algorithm 4 for reference). STO aborts a transaction if $\tau < t_k$, where t_k is the timestamp of a key k that is being accessed. Intuitively, since the comparison is always checking that τ is *smaller* than one of the key's timestamps, overapproximating key timestamps can only cause transactions to abort. We make this more formal with a simulation argument. Suppose STO with approximate timestamp storage processes transactions $T_1, \ldots, T_n$. Consider STO with exact timestamp storage processing the same set of transactions with the same ordering of their operations. Suppose that STO with exact timestamps has the additional property that it aborts any transaction that STO with approximate timestamp storage aborts. This version of STO still guarantees serializability. So, if we can show that it would commit all the transactions that approximate STO would commit, then this would mean that approximate STO is also serializable. STO only ever does *put*(k, τ), i.e. the only timestamps that it stores in the timestamp storage engine are the timestamps assigned to transactions. Thus the sequence of *put* operations performed by approximate and exact STO will be identical. Hence every *get* performed in approximate STO will return a value at least as large as the one returned in exact STO (Property 1). This implies that every timestamp check that causes an abort in exact STO will also cause an abort in approximate STO. Hence approximate STO is serializable.

For MVTO's correctness, as long as timestamp overapproximation does not change the original timestamp ordering of the versions for any record, the same argumentation as for STO applies.

For TicToc, a simulation argument does not appear to be feasible, but the original TicToc proof [51] continues to work, even for approximate timestamp storage. This is because the proof reasons almost

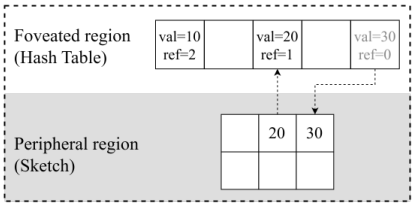


Fig. 2. FPSketch consists of the foveated region (hash table) storing accurate timestamps for active keys, peripheral region (sketch) for approximated timestamps, and the eviction mechanism transferring data between them.

entirely about timestamp inequalities across physical time, and approximate timestamp storage will preserve those inequalities. We only need the following lemma for approximate timestamp storage in order to make the original TicToc proof go through:

Lemma 3. *Each timestamp increases monotonically in physical time in TicToc with approximate timestamp storage. Furthermore, every write to a key k causes k's write timestamp to increase.*

Proof. We establish the second fact first. Consider a committing transaction that writes to k. TicToc guarantees that, when the transaction performs $put(k.wts, \tau)$ during its commit phase, τ is larger than the read timestamp returned by a previous $get(k.rts)$ performed by the same transaction. By Property 2, $k.wts \leq k.rts$ at the time of the *get* and, since the transaction references k for the entire duration between the *get* and *put*, the approximate timestamp storage cannot spontaneously increase $k.wts$ during that interval. Furthermore, the transaction holds a lock on k the entire time between the *get* and the *put*, so no other transaction could have performed a *put* on $k.wts$. Hence $k.wts$ cannot change between the transaction's *get* and its *put*, ensuring that the *put* increases $k.wts$.

The above argument also shows that, when a transaction performs $put(k.rts, \tau)$ in its commit phase, it does not decrease $k.rts$.

A transaction that reads k may also update $k.rts$ during its validation phase, but this trivially does not decrease $k.rts$ because the transaction does $put(k.rts, \max(\tau, get(k.rts)))$ in an atomic section.

Thus *put* operations can never decrease a key's timestamps. The only other way a timestamp can change is through approximation, which is guaranteed not to decrease a timestamp (Property 1). Thus timestamps increase monotonically over physical time. □

The two facts established in the above lemma are sufficient to enable TicToc's original proof of serializability to go through.

In the next section, we design an approximate timestamp storage data structure that meets the above requirements.

5 FPSketch

In this section we describe FPSketch, an approximate timestamp storage structure that meets the requirements identified in Section 4.

As shown in Figure 2, FPSketch consists of two regions: a "foveated" region providing accurate timestamps and a lossy "peripheral" region that maintains overapproximated timestamps. FPSketch is a hybrid data structure comprising both a hash table for the foveated region and a sketch for the peripheral region. Additionally, the FPSketch uses an *eviction mechanism* to move timestamps from the foveated region to the peripheral one.

Algorithm 1 Pseudocode of Sketch's PUT and GET functions using atomic operations.

```
struct SKETCH
    int rows, cols
    value table[rows][cols]                          ▷ value must support atomic ops.
    uint64 seeds[rows]
end struct
 1: procedure TIMESTAMPMAX(TIMESTAMPS t1, TIMESTAMPS t2)
 2:     (v.wts, v.rts) ← (MAX(t1.wts, t2.wts), MAX(t1.rts, t2.rts))
 3:     return v
 4: procedure TIMESTAMPMIN(TIMESTAMPS t1, TIMESTAMPS t2)
 5:     (v.wts, v.rts) ← (MIN(t1.wts, t2.wts), MIN(t1.rts, t2.rts))
 6:     return v
 7: procedure PUT(key k, value v)
 8:     for row = 0 to rows − 1 do
 9:         col ← HASH(k, seeds[row]) mod cols
10:         current_value ← ATOMICLOAD(&table[row][col])
11:         is_success ← false
12:         while !is_success do
13:             new_value ← TIMESTAMPMAX (current_value, v)
14:             is_success ← ATOMICCOMPAREEXCHANGE(&table[row][col],
                                &current_value, new_value)
15: procedure GET(key k)
16:     col ← HASH(k, seeds[0]) mod cols
17:     value ← ATOMICLOAD(&table[0][col])
18:     for row = 1 to rows − 1 do
19:         col ← HASH(k, seeds[row]) mod cols
20:         current_value ← ATOMICLOAD(&table[row][col])
21:         value ← TIMESTAMPMIN (current_value, value)
22:     return value
```

Hash table. The hash table stores timestamps of keys that are currently being accessed, ensuring that FPSketch does not spontaneously increase the timestamps of these keys. Each record in the hash table has a reference count, and each transaction increments the reference count the first time it accesses a record and decrements it upon commit/abort. Only keys for which the reference count reaches zero may be evicted from the hash table to the sketch. Depending on the eviction mechanism chosen, these keys may either be evicted immediately or kept in the hash table until evicted due to space pressure using, e.g. LRU.

Sketch. Timestamps evicted from the hash table are not discarded; instead, they are stored in the sketch. Algorithm 1 provides pseudocode for the PUT and GET functions of the sketch, utilizing atomic operations. The core logic is analogous to the count-min sketch, except it takes max instead of incrementing during PUT operations, which guarantees Property 1. For STO, MVTO, and TicToc, timestamps are ordered pairs (wts, rts) of write and read timestamps, respectively. The sketch takes field-wise max and min, as shown in Algorithm 1. Note that field-wise max and min preserve that $wts \leq rts$, so our sketch preserves Property 2.

When considering the memory limitations for both the hash table and the sketch within the FPSketch framework, it's important to note that the hash table's memory usage is directly tied to the quantity of keys currently accessed in parallel. Conversely, the size of the sketch is predetermined and remains constant based on the configured number of rows and columns, with the number of rows aligning with the number of hash functions employed.

Algorithm 2 Pseudocode of FPSketch Operations. The pseudocode assumes linearizability using locks.

```
struct FPSketch
    HashTable hashtable
    Sketch sketch
end struct
1:  procedure IncRef(key k)
2:      e ← hashtable.GetItemPtr(k)
3:      if e = NULL then
4:          ts ← sketch.Get(k)
5:          hashtable.Put(k, ts)                          ▷ Initialize refcount = 1
6:          Evict()
7:      else
8:          e.refcount++
9:  procedure DecRef(key k)
10:     e ← hashtable.GetItemPtr(k)                       ▷ e cannot be NULL
11:     e.refcount−−
12:     if e.refcount = 0 then
13:         Evict()
14: procedure Get(key k)
15:     return hashtable.GetItemPtr(k).ts
16: procedure Put(key k, timestamp ts)
17:     hashtable.GetItemPtr(k) .ts ← ts
18: procedure Evict()
19:     E ← {item in hashtable | item.val.refcount = 0}   ▷ Only choose to evict from the set of keys that are
    not in-use
20:     for all item in toEvictPolicy(E) do
21:         sketch.Put(item.k, item.val.ts)
22:         hashtable.Remove(item.k)
```

5.1 FPSketch Operations

FPSketch provides four operations, as shown in Algorithm 2: Get, Put, IncRef, and DecRef. IncRef and DecRef increment and decrement the refcounts on keys in the FPSketch, and may need to transfer timestamps between the hashtable and the sketch. Transactions are required to call IncRef(k) before their first access to a key k, through a Get or Put operation, and should not access the key after calling DecRef(k). Although we omit locking from the pseudocode, all operations ensure linearizability for each item in the hash table through the use of per-bucket locks.

IncRef. allocates a hashtable slot for k if one does not already exist, initializes its timestamp from the sketch, and sets its refcount to 1. Otherwise it just increments the refcount of the existing entry. IncRef executes atomically via a lock on the hashtable bucket.

DecRef. looks up the key in the hashtable, which is guaranteed to exist due to the prior IncRef, and it decrements the entry's refcount. Again, this sequence of operations is executed atomically by using per-bucket locks on the hashtable. If the refcount reached 0, it calls the eviction method that implements the chosen eviction mechanism. The eviction mechanism may only evict keys for which refcount is 0, and it implements an eviction policy, toEvictPolicy, which selects which of these keys to evict.

GET and PUT. just return or update the timestamps in the hashtable entry, which is guaranteed to exist due to the prior call to INCREF.

5.2 FPSketch Variants

FPSketch allows customizable sizes for both its regions; this customization exposes the trade-off between timestamp accuracy and space. Increasing the size of either of these regions also increases the overall accuracy of the timestamps, but the space consumption for the accuracy boost varies across the two regions. In this paper we explore two FPSketch variants: Focus-Counter, which favors a large foveated region, and Focus-Sketch which, in contrast, favors a large peripheral region. More precisely, given a fixed size space, S, that can be used to store the timestamps of the keys that are not in-use, Focus-Counter allocates most of S for the foveated region, while Focus-Sketch allocates most of S for the peripheral region, as we detail further. For Focus-Sketch, we provide intuition on how to size the sketch (in terms of the number of rows and columns) in the next section.

5.2.1 Focus-Counter. is a variant of FPSketch that uses the smallest possible sketch, a sketch of size one, and allocates all the remaining available space to the hashtable. This variant uses the CLOCK eviction mechanism to evict timestamps from the hashtable when it runs out of space to store timestamps of keys that are not in-use.

5.2.2 Focus-Sketch. is a variant of FPSketch that uses the smallest possible hashtable by implementing an aggressive eviction policy that evicts a key from the hashtable as soon as its refcount reaches 0 (i.e., is no longer in-use by any on-going transaction). All the available space is allocated to the sketch.

5.3 Sizing FPSketch

In this section, we give a simple heuristic argument that the size of the sketch should be proportional to $\Theta(CK)$ columns and $\Theta(\log CK)$ rows, where C is the average number of concurrent transactions and K is the average number of keys accessed by each transaction. Thus the total size of the sketch need be at most $O(CK \log CK)$.

We argue that, if the sketch is large enough, then, during the entirety of some transaction T's execution, the sketch will not spontaneously increase the timestamps of any of the keys accessed by T. Let A_T be the set of keys accessed by *any* transaction during T's execution. Note that the average size of A_T is around CK. If the sketch has CK columns then, within each row, a constant fraction of the keys will not collide with any other key in A_T. Since there are $\Theta(\log CK)$ rows, with high probability every key is collision-free in at least one row. Thus the sketch will not make any new approximation errors on any of the keys in A_T during T's execution.

So, most of the time, the system will behave as if it is using exact timestamp storage during a transaction T's execution. We just have to argue that approximation errors made before T begins won't affect whether T aborts. This is straightforward from examining each CC mechanism. For example, STO compares timestamps of keys only with T's timestamp, τ. All keys have timestamps less than τ before T begins, so the only way T can abort is if some other transaction updates a timestamp of a key accessed by T. But then T would have aborted in a system with exact timestamps, as well. Similar arguments can be made for TicToc and MVTO.

Note this analysis is merely a heuristic rule of thumb. It assumes that the transactions do not try to cause aborts. A malicious actor could attempt to discover colliding keys (by issuing transactions on different keys and seeing which ones abort) and then issue a sequence of transactions that it knows will all abort due to hash collisions in the sketch. Second, it ignores the fact that, if a transaction aborts due to a timestamp approximation, that may allow another transaction to

commit, which in turn may cause another transaction to abort, and so on. So it is not strictly true that timestamp approximations cannot affect future transactions. However, this effect is not likely to be significant. Finally, it ignores variance in the sizes and concurrency of transactions, which may necessitate larger sketches, and it ignores skew in key-access patterns, which may allow much smaller sketches.

As for the hashtable, recall that the hashtable needs to hold only active keys, and so it should be sized to hold roughly $\Theta(CK)$ keys, as well. In practice, we would advise using a resizable hashtable and simply letting it grow as needed. Several state-of-the-art hashtables support concurrent operations while resizing [26, 27, 32], so this need not cause any hiccups in throughput.

5.4 FPSketch Implementation

We implement the hash table of FPSketch on top of IcebergHT [32] and the sketch from scratch. We modified IcebergHT to perform reference-count management and to support holding bucket locks across multiple operations. For the Focus-Counter variant that uses a sketch of size one, we use an optimized version of Algorithm 1 that no longer needs to compute hashes.

6 Integration with the Existing CC Algorithms

In this section, we describe how to integrate FPSketch into STO, MVTO, and TicToc. STO and MVTO differ substantially from TicToc in that TicToc verifies transactions post-execution in an optimistic manner, whereas STO and MVTO can abort transactions mid-execution. All three maintain read and write timestamps per key (per version in MVTO's case).

6.1 TicToc

Algorithm 3 shows the pseudocode for TicToc with FPSketch. There are only two changes to the algorithm. First, timestamps are stored in the FPSketch instead of inline with the tuples, as in the original TicToc paper, so the syntax for accessing timestamps is different. Second, we need to call IncRef and DecRef when a transaction first accesses a key and when it finishes. Note that all the logic for dealing with timestamp approximation is encapsulated within the FPSketch code—TicToc is not aware of the approximations.

Transactions increment a key's refcount the first time they access it, and decrement it only during commit (or as part of the abort procedure, which is not shown). This ensures that the key's timestamps will not change spontaneously due to approximations in the timestamp storage structure during the transaction's execution. It then records the timestamps in the transaction's read set, which is structured as a map from keys to timestamps.

The write set is a map from keys to data items. When a transaction writes to a key, we just record the specified key-value pair in the transaction's write set. Note that a transaction doesn't need to check the timestamps of a key in its write set until validation time, as in the original TicToc. Although not shown explicitly, the full implementation checks the transaction's write set during reads so that a transaction sees its own writes.

TicToc requires a mechanism to lock keys during transaction validation. Although this is abstracted in our pseudocode, in our implementation this is accomplished by having a lock bit in a key's entry in the timestamp hash table.

When a transaction either commits or aborts, it is essential to unlock the write set and decrease the reference counts of the accessed keys. It's worth noting that we omit the code for finalizing transactions upon abort due to space constraints. However, the steps involved in unlocking and removing entries for transaction aborts are the same as those outlined in Line 34 and Line 33.

Algorithm 3 TicToc integrated with FPSketch. Each transaction T maintains a read set R and a write set W. TicToc buffers write values in the write set W until commit time.

```
1:  procedure READ(database D, FPSketch S, transaction T, key k)
2:      begin atomic section
3:          if k ∉ T.R then
4:              S.INCREF(k)
5:          T.R[k] ← S.GET(k)
6:          T.R[k].data ← D.READ(k)                    ▷ Read k's value from disk
7:      end atomic section
8:      return T.R[k].data
9:  procedure VALIDATE(transaction T)
10:     for k ∈ T.W in sorted order do
11:         LOCK k
12:         S.INCREF(k)
13:     T.τ ← 0                                        ▷ Compute our commit timestamp T.τ
14:     for k ∈ T.W ∪ T.R do
15:         if k ∈ T.W then
16:             T.τ ← max(T.τ, 1 + S.GET(k).rts)
17:         else
18:             T.τ ← max(T.τ, T.R[k].wts)
19:     for k ∈ T.R do
20:         begin atomic section
21:             if T.R[k].rts < T.τ then
22:                 if T.R[k].wts ≠ S.GET(k).wts ∨ (S.GET(k).rts ≤ T.τ ∧ LOCKED(k) ∧ k ∉ T.W) then
23:                     ABORT
24:                 else
25:                     curr ← S.GET(k)
26:                     curr.rts ← max(T.τ, curr.rts)
27:                     S.PUT(k, curr)
28:         end atomic section
29: procedure COMMIT(database D, FPSketch S, transaction T)
30:     for k ∈ T.W do
31:         D.WRITE(k, T.W[k].data)                    ▷ Write k's value to disk
32:         S.PUT(k, (T.τ, T.τ))
33:         S.DECREF(k)
34:         UNLOCK k
35:     for k ∈ T.R do
36:         S.DECREF(k)
```

6.2 STO

Algorithm 4 shows the pseudocode of STO with FPSketch. The changes are similar to those in TicToc.

STO assigns each transaction a commit timestamp when it begins. Whenever a transaction reads or writes an entry in the database, STO locks that key, checks that the key's timestamps are compatible with the commit timestamp of the transaction, and then updates the key's timestamps. STO does not allow transactions to access uncommitted data. This can be achieved with read-write locks, as illustrated in Algorithm 4. We implemented these locks with "dirty bits"[48]. With dirty bits, readers do not block writers, enabling more concurrency than in 2PL. Dirty bits work as follows. When a writer updates a tuple, it sets the tuple's dirty bit to 1. This blocks subsequent readers and writers until the writing transaction commits or aborts. Readers, however, use the

Algorithm 4 STO integrated with FPSketch. Each transaction T maintains a read set R and a write set W.

```
1:  procedure BEGIN(global timestamp *c, transaction T)
2:      T.τ ← ATOMICFETCHADD(c, 1)
3:  procedure READ(database D, FPSketch S, transaction T, key k)
4:      if k ∉ T.R then
5:          READLOCK k
6:          T.R ← T.R ∪ {k}
7:          S.INCREF(k)
8:          if T.τ < S.GET(k).wts then
9:              ABORT
10:         begin atomic section
11:             curr ← S.GET(k)
12:             curr.rts ← max(T.τ, curr.rts)
13:             S.PUT(k, curr)
14:         end atomic section
15:         T.R[k].data ← D.READ(k)                    ▷ Read k's value from disk
16:         UNLOCK k
17:     return T.R[k].data
18: procedure WRITE(database D, FPSketch S, transaction T, key k, val v)
19:     if k ∉ T.W then
20:         WRITELOCK k
21:         S.INCREF(k)
22:         if (T.τ < S.GET(k).wts) ∨ (T.τ < S.GET(k).rts) then
23:             ABORT
24:         curr ← S.GET(k)
25:         curr.wts ← max(T.τ, curr.wts)
26:         S.PUT(k, curr)
27:     T.W[k] ← v
28: procedure COMMIT(database D, FPSketch S, transaction T)
29:     for k ∈ T.R do
30:         S.DECREF(k)
31:     for k ∈ T.W do
32:         D.WRITE(k, T.W[k])                         ▷ Write k's value to disk
33:         S.DECREF(k)
34:         UNLOCK k
```

dirty bit only as a latch to atomically read the tuple into a private buffer, and hence do not block subsequent writers (or, obviously, subsequent readers).

As in our TicToc implementation, we store the reader-writer locks in the entries in the FPSketch. Although not shown explicitly, if a transaction writes a key after reading it, it needs to be able to atomically upgrade its reader lock to a writer lock. If upgrading is not possible, the transaction aborts.

Note that STO needs to update the timestamps in the sketch only the first time a transaction accesses a key, since the timestamp updates performed by a transaction using STO are idempotent. During a read, we need to read and write the timestamps in the sketch atomically, since other readers may also be updating the key's read timestamp. This isn't necessary for writes, since the transaction already holds an exclusive lock on the key.

To commit, a transaction needs only to write its data back to the database, release all references to entries in the timestamp storage structure, and unlock all the keys. Aborting, not shown, is similar.

6.3 MVTO

In a standard MVTO system, the database may maintain multiple versions of each key-value pair—a new version is created whenever a transaction commits a write to the respective key. Each version has associated timestamps and, when a transaction first reads a key, it looks through the versions to find one whose timestamps make it visible to that transaction and uses that version. If the transaction cannot find a compatible version (e.g. if the old versions have already been garbage collected) then it must abort.

As we noted in Section 4, MVTO requires that the approximate timestamp storage system must maintain the relative timestamp ordering of the versions of a key-value pair. We satisfy this requirement by ensuring that our system approximates the timestamps of only the most recent version of any key-value pair.

Specifically, our MVTO implementation works as follows. When a key is inactive, we store the (approximate) timestamps of the most recent version of that key in FPSketch. When the key becomes active, we copy the timestamps of the most recent version from FPSketch into the hashtable. Whenever a new version of the key is created, we store the timestamps of the new version in the hashtable, as well. Once the key becomes inactive, we move the timestamps of the most recent version back to FPSketch and discard the timestamps of all the older versions.

Whenever a new version is created, it is assigned a monotonically increasing version number and stored in SplinterDB indexed as (key, version-number) $\rightarrow$ value. Each entry in the hashtable also contains the version number so, once a transaction finds an entry with compatible timestamps in the hashtable, it learns the version number that it needs in order to look up that version of the key-value pair in SplinterDB using a point lookup.

When a transaction accesses an inactive key, however, the sketch can tell the transaction only the timestamps of the most recent version, but not its version number. Thus the transaction does not know what version number to use to look up the key in SplinterDB. We considered two solutions to this problem. The first was to have the transaction perform a predecessor query in SplinterDB, i.e. to query for the predecessor of (key, ∞). However, predecessor queries in SplinterDB are slower than point queries because predecessor queries cannot use SplinterDB's maplets [10].

To avoid predecessor queries, we implemented the following optimization. Whenever a new version of a key-value pair is inserted into SplinterDB, we also insert the same key-value pair into SplinterDB with the special version number 0, i.e. we maintain the invariant that the most recent version of a key is always stored as (key, 0). This incurs two insertions into SplinterDB for each version, but enables queries of inactive keys to perform point queries instead of range queries. Overall, we found this approach offered higher performance, so this is the scheme we use in all our benchmarks.

Persistence and crash safety. It is possible to handle crashes using standard techniques like write-ahead logging and checkpoints. Note that we do not need to persist the timestamps across crashes or shutdowns since there can be no serializability violations between transactions that execute before and after a crash or shutdown.

7 Evaluation

We evaluate the effectiveness of FPSketch in scaling timestamp-based CC mechanisms to on-disk databases. There are five high-level take-aways from our experiments:

- FPSketch-based versions of STO, MVTO, and TicToc outperform versions that keep timestamps on disk in all our experiments, often by dramatic margins. For example, FPSketch improves TicToc goodput by up to 5.9x, STO by up to 1.8x, and MVTO by over 160x.
- Similarly, Focus-Sketch-based protocols can offer up to 2x greater goodput than Focus-Counter-based ones. Furthermore, TicToc and STO with Focus-Sketch are never substantially slower than the Focus-Counter variants.
- The goodputs of TicToc and STO with FPSketch are close to an idealized algorithm, which assumes we can fit all timestamp metadata in RAM, across diverse workloads.
- TicToc-Focus-Sketch in particular outperforms 2PL and OCC across the board.
- FPSketch requires a tiny amount of memory – 32KiB in our experiments with an 80GB database.

In summary, FPSketch offers essentially strictly improved performance with little to no downside for STO, MVTO, and TicToc across a wide range of benchmarks.

7.1 Experimental Setup

We run our experiments on a CloudLab [15] Clemson r6525 machine with two 32-core AMD 7543 at 2.8GHz, 256GB memory, and one 1.6TB PCIe v4.0 NVMe SSD.

We use SplinterDB, a fast, scalable write-optimized key-value store, as the underlying storage system for the concurrency control mechanisms we consider in this evaluation. We configure SplinterDB to use the raw device I/O (with the DIRECT_IO flag) to avoid the filesystem overhead. We size SplinterDB's internal cache to a percentage of each workload's database size, as we detail below.

In all our experiments, we utilize 120 threads, which is the maximum supported by the version of SplinterDB used in this work. Each thread executes transactions in a closed loop. When a transaction aborts, it can be retried up to 5 times. The retry mechanism uses exponential backoff. We tuned the initial backoff interval for each scheme by initially setting it to be roughly twice the average uncontended transaction completion time, and then tuned it until the abort rate roughly matched that of the Memory variant.

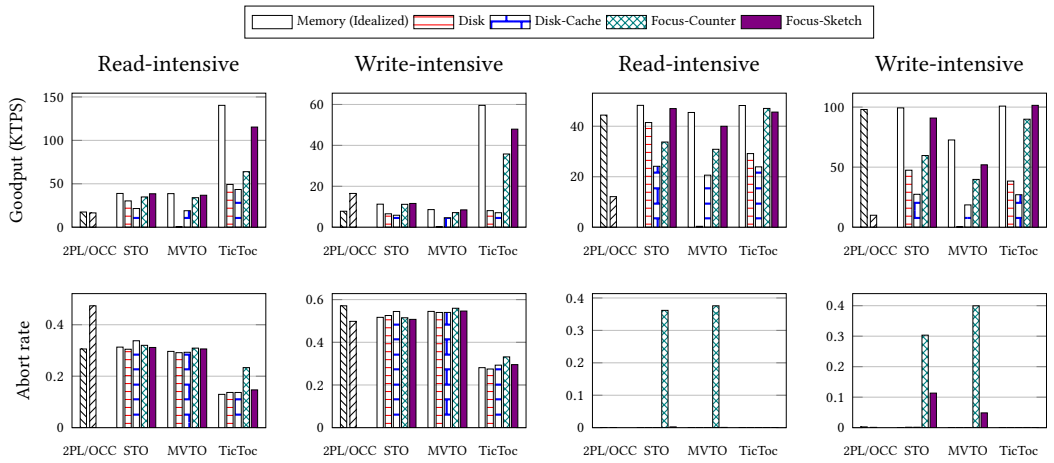
We measure Goodput in Kilo Transactions Per Second (KTPS) and also evaluate the abort rate. Goodput represents the number of successfully committed transactions per second. The abort rate is calculated as the ratio of the total number of aborts, which can be up to 6 per transaction (including the initial attempt and 5 retries), to the total number of attempted transactions, which is the sum of all aborts and commits.

Concurrency control mechanisms. We implement and evaluate several CC mechanisms, including all three timestamp-based CCs described in Section 2.1 (STO, MVTO, and TicToc), and two classic CCs which are widely used in disk-based databases: two-phase locking (2PL) and Kung-Robinson Optimistic Concurrency Control (OCC). Unlike timestamp-based CCs, these two classic mechanisms do not require maintaining per-record metadata, thus avoiding disk I/O overhead for metadata access. For 2PL, we employ a no-wait policy after evaluating multiple deadlock prevention mechanisms (including wound-wait and wound-die). We selected no-wait due to its generally good performance in our experiments, which aligns with findings from previous works [38, 43].

We implement five variants of STO, MVTO, and TicToc. These variants are distinguished by their methods for storing and accessing per-record timestamps, which range from relying entirely on disk to using fully in-memory solutions.

First, we establish the performance boundaries with two baseline variants:

- *Disk*: This is the simplest approach. Timestamps are stored on disk as part of each record, requiring an I/O operation for every timestamp access.



(a) High-contended YCSB (zipf=0.99) (b) Medium-contended YCSB (zipf=0.6)

Fig. 3. YCSB small-transaction workloads results with 120 processing threads. In write-intensive workloads, each transaction performs 8 reads and 8 writes. In read-intensive workloads, each transaction performs 15 reads and 1 write. Note that, in the medium-contended workload, several schemes have abort rates very close to 0.

- *Memory*: This represents a theoretical upper bound on performance. It stores timestamps for all keys in a single, large in-memory hash table. While fast, this approach is generally impractical due to its prohibitive memory requirements.

The remaining three variants maintain an in-memory hash table to store the keys of currently active transactions. They differ primarily in how they manage timestamps for the much larger set of inactive keys:

- *Disk-Cache*: This variant enhances the Disk approach by adding a cache for inactive keys, inspired by PostgreSQL's SLRU cache [33], buffering on-disk metadata. It uses the same active-key hash table as the sketch variants. For inactive keys, it employs a 32 KiB in-memory cache with a CLOCK eviction policy to buffer timestamps read from disk. This cache operates alongside SplinterDB's standard page caching. We also explore the impact of larger caches in Section 7.2.
- *Focus-Counter*: As one variant of FPSketch, described in Section 5.2, it supplements the active-key hash table with an additional 32 KiB of memory dedicated to storing timestamps for inactive keys. Also, a CLOCK eviction policy is employed.
- *Focus-Sketch*: The other variant of FPSketch also uses the active-key hash table but employs a 32 KiB probabilistic sketch for managing inactive key timestamps. The sketch is configured with 2 rows to balance space efficiency and error rates [40], with the number of columns set to fit the memory budget.

Workloads. We use two benchmarks to evaluate the performance of the CCs we implement: (1) YCSB [20], which models large-scale online services, and (2) TPC-C [14], the industry standard for evaluating OLTP systems.

In the YCSB workloads, we first load a dataset of 673 million key-value pairs, with 24-byte keys and 100-byte values. This loading phase is performed before every run, followed by the execution phase. To evaluate FPSketch's across diverse scenarios, we execute seven YCSB workloads: four "small" transaction workloads, two "mixed" transaction workloads, and one "large" transaction

workload. Each of the small workloads is either *read-intensive* or *write-intensive*. In write-intensive workloads, transactions perform 8 reads and 8 writes. In read-intensive workloads, transactions perform 15 reads and 1 write. The keys used in these transactions are generated based on a Zipfian distribution. Each of the workloads is either *high-contention* or *medium-contention*. For the high-contention workloads the Zipfian distribution uses *theta*=0.99 (10% of the pairs are accessed by 80% of the operations). For the medium-contention workloads, *theta*=0.6 (10% of the pairs are accessed by 40% of all the operations). The "mixed" transaction workloads consist of a mix of small transactions (reading and writing 2 keys each) and medium-sized transactions, which read 28 keys each. For these workloads, the Zipfian *theta* values of 0.9 and 0.6 are used for high- and medium-contention, respectively. Finally, we run a high-contention (Zipfian 0.9) "large" transaction workload, where 5% of the transactions read 1000 keys, and the remaining transactions each read and write 8 keys. SplinterDB's internal cache is 6GiB, which is less than 10% of the database size. We run YCSB experiments over a duration of 240 seconds. We repeat each run three times and report average numbers.

As in many previous works, we evaluated TPC-C workloads that comprise only two of the five transaction types in TPC-C, namely `Payment` and `NewOrder`, with each type making up 50% of the workload. These two transaction types constitute 88% of the default TPC-C mix and are the most interesting for our evaluation. We ran the TPC-C workloads for various number of warehouses, 4, 8, 16, and 32, which dictate the initial database size and the contention level. SplinterDB's internal cache was configured to 256MB. The TPC-C experiments are run for 120 seconds and repeated three times to minimize noise caused by random variation.

7.2 YCSB Results

Small-transaction workloads. Figure 3 shows the goodput and abort rates for all our timestamp-based CC mechanisms on our four small-transaction YCSB workloads.

One important result from Figure 3 is that in all scenarios, integrating FPSketch with the timestamp-based CCs enables them to substantially outperform their "Disk" and "Disk-Cache" counterparts. Except with MVTO, which we discuss below, Disk-Cache is slightly slower than Disk due to the overhead of managing the extra cache. We provide a deeper dive on how cache size affects Disk-Cache's performance in another experiment described below. The STO and MVTO variants match their "Memory" counterparts. TicToc experiences performance penalties from aborts because validation occurs after optimistically executing all operations, requiring them to be re-executed. This effect is particularly pronounced for read operations. Nevertheless, TicToc with FPSketch still achieve the best performance across our workloads. The Focus-Sketch-based protocols also match or outperform their Focus-Counter-based variants on these workloads.

We can draw several other conclusions from these results.

- TicToc variants generally outperform not only 2PL/OCC but also their STO and MVTO analogues, due to TicToc's advanced timestamp-based scheme.
- Disk-based variants are substantially slower, due to I/O to fetch timestamps. I/O overhead is particularly pronounced in write-intensive workloads because, as explained in Section 3, reads bring in the timestamps "for free" but writes to SplinterDB do not cause the old value (and hence the old timestamps) to be read into cache. Consequently, Disk-based variants incur reads for every write, incurring high overhead in write-intensive workloads.
- The gap between Disk-based and other variants is also larger in the medium-contended workloads because these workloads have less locality, increasing cache miss rates.
- MVTO-Disk is also particularly slow because it stores old versions on disk and uses a range query to find the most recent version whenever it needs to bring a key into cache. Range queries are slower than point queries because range queries do not use filters to avoid I/O

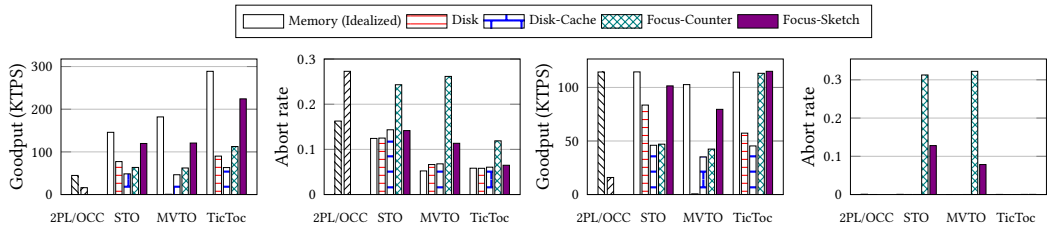


Fig. 4. YCSB mixed-transaction workloads results with 120 processing threads. 80% of transactions perform 2 reads and 2 writes, and 20% of transactions perform 28 reads. Note that, in the medium-contended workload, several schemes have abort rates very close to 0.

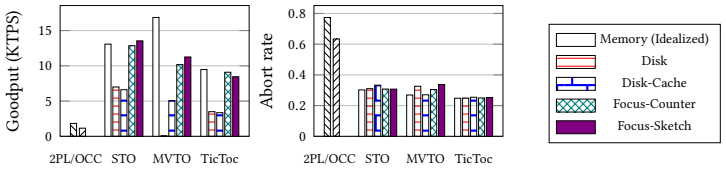


Fig. 5. YCSB long-transaction workloads results with 120 processing threads. 5% of transactions perform 1000 reads and the rest of transactions perform 8 writes and 8 reads. The distribution is Zipfian with 0.9.

in SplinterDB. In contrast, MVTO-Disk-Cache is able to utilize the same *V0* technique as our FPSketch-based MVTO variants, which enables it to use point queries instead of range queries to bring keys into cache. As a result, MVTO-Disk-Cache substantially outperforms MVTO-Disk.

Mixed-transaction workloads. Figure 4 shows the goodput and abort rates for our timestamp-based CC mechanisms on our mixed-transaction workloads. Several of the same observations apply here, as well: TicToc variants generally perform better than all other variants, 2PL, and OCC; Disk variants are substantially slower; and MVTO-Disk suffers in particular due to its use of range queries when bringing a key into cache. In high-contention workloads, STO and MVTO implementations with Focus-Sketch demonstrate higher goodput compared to 2PL and OCC. However, in medium-contention scenarios, they experience performance degradation due to unnecessary aborts caused by timestamp approximation. Additionally, MVTO-Focus-Sketch incurs extra overhead from maintaining a special V0 version. These workloads also show that the Focus-Sketch-based variants consistently substantially outperform their Focus-Counter-based variants. TicToc-Focus-Sketch, STO-Focus-Sketch, and MVTO-Focus-Sketch have about 50% greater goodput than their Focus-Counter variants, respectively. This advantage stems from Focus-Sketch's ability to maintain more accurate timestamp approximations compared to Focus-Counter, resulting in fewer unnecessary aborts.

Long-transaction workload. Figure 5 shows our final YCSB workload, which includes some long transactions that read 1000 keys. Most of the trends here are similar to the other YCSB workloads.

The main take-away is that STO and TicToc with approximate timestamp storage work just as well as their Memory variants, demonstrating that approximate timestamps can handle large transactions, not just small ones.

Approximate timestamp storage versus caching. Figure 6 shows how varying the SplinterDB cache size affects goodput of TicToc-Disk and TicToc-Focus-Sketch. The purpose of this experiment is

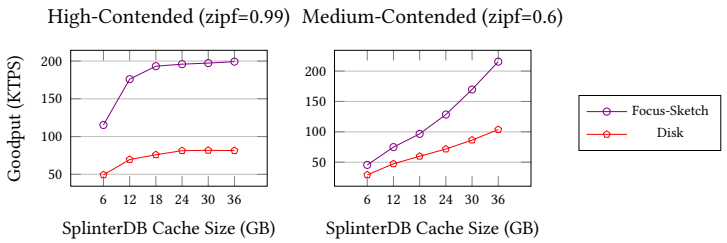


Fig. 6. YCSB read-intensive small-transaction workloads for TicToc Disk and Focus-Sketch with varying SplinterDB Cache Size.

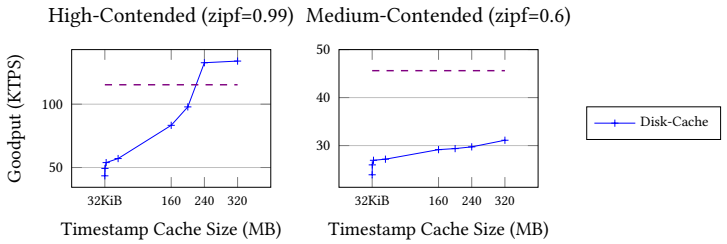


Fig. 7. YCSB read-intensive small-transaction workloads for TicToc Disk-Cache compared to Focus-Sketch with 32KiB of sketch size (Dashed line).

to answer the question: can we fix the performance problems in Disk-based schemes by simply increasing cache sizes?

Figure 6 suggests that the answer is "No." In the high-contended workload, TicToc-Disk with a 37 GiB cache is still slower than TicToc-Focus-Sketch with a mere 6GiB cache. And in the medium-contended workload, TicToc-Disk needs roughly twice as much cache to match the goodput of TicToc-Focus-Sketch, which needs only 32KiB to store timestamps.

There are likely several reasons that database cache is not as effective as in-memory approximate timestamp storage. First, caches store disk blocks, which may store numerous key-value-timestamp records. So, in order to store a single timestamp for a single key, the block cache will need to hold an entire disk block, whereas approximate timestamp storage will need only a few bytes. Thus approximate timestamp storage just uses memory more efficiently. Second, timestamp storage is optimized for quick, in-memory access, whereas database caches can have high overheads due to locking, latching, and data structure traversals, among other things.

Metadata Memory Usage. We now compare the memory used to store metadata in each scheme. The Memory variant stored in RAM 40 bytes for each KV-pair: a 24-byte key and a 16-byte structure containing a read timestamp, write timestamp, and a latch. There were 673M keys in the database, so the Memory variant stored 27GiB of metadata. The Disk variant also stored 16 bytes of metadata with each 124-byte record in the database. All metadata was stored in SplinterDB's 6GiB cache, so the total memory used for metadata was approximately $16 / (124 + 16) \times 6$ GiB ≈ 685 MiB. The Disk-Cache just adds a small 32KiB cache to the Disk scheme and hence has similar memory usage. The FPSketch variants use a 32KiB cache plus a hashtable of 40-byte entries for each active key, for a total average memory usage of 160KiB.

Disk-Cache versus Focus-Sketch. Figure 7 compares the performance of Disk-Cache with Focus-Sketch. We used the same configuration for Focus-Sketch–32KiB cache. The dashed line is the

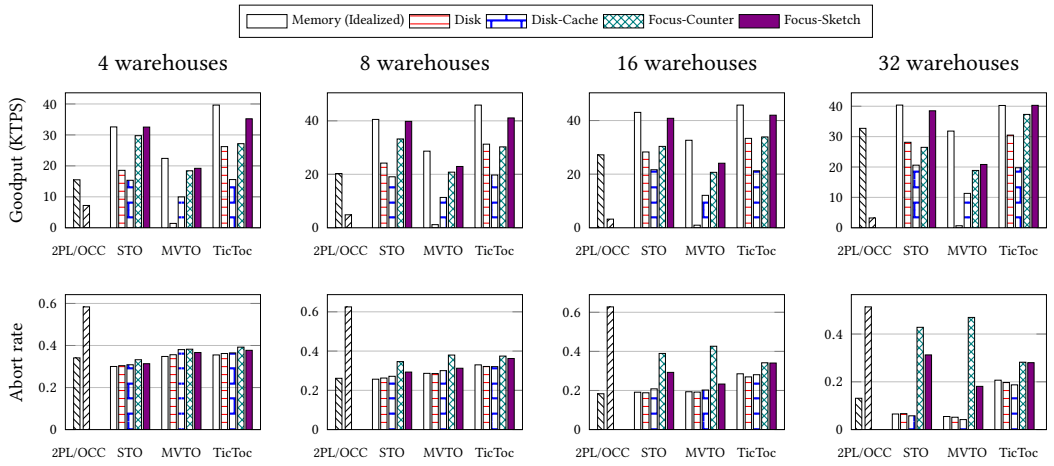


Fig. 8. TPC-C results with 120 processing threads (More warehouses means less contention).

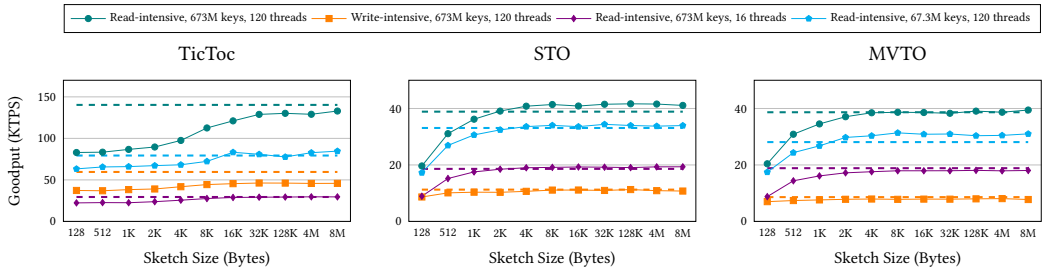


Fig. 9. The goodput of TicToc, STO, and MVTO for high-contended YCSB workloads with varying sketch sizes. We configure workloads by operations proportion (Read-intensive vs. Write-intensive), the database size (containing 673M vs. 67.3M keys), and the concurrency level (120 vs 16 threads). The dashed lines represent the goodputs of the Memory variant for each configuration.

goodput of Focus-Sketch. In high-contended workloads, Disk-Cache requires about 220MB of cache to achieve the same goodput as Focus-Sketch, which uses only 32KiB of memory. Disk-Cache never achieves the same goodput as Focus-Sketch in our medium-contended workloads because the Disk-Cache variant still requires I/O to access timestamps, while Focus-Sketch does not.

7.3 TPC-C Results

Figure 8 shows the goodput and abort rates for TPC-C workloads for various numbers of warehouses. The primary take-away from these experiments is that the trends seen in the YCSB experiments generalize to other workloads.

The main observation is that, as in the YCSB experiments, STO-Focus-Sketch and TicToc-Focus-Sketch are able to nearly match the performance of their Memory variants.

Moreover, TicToc variants generally outperform 2PL, OCC, and the other variants; Disk-based variants are substantially slower than Memory, Focus-Counter, and Focus-Sketch; MVTO-Disk is particularly slow due to its use of range queries; the gap between STO and TicToc variants tends to close as contention decreases (i.e. as the warehouses increase); and the Focus-Sketch variants never perform substantially worse than the Focus-Counter variants.

7.4 Sketch Size

We conduct a factor-analysis of the sketch size of FPSketch for TicToc, STO, and MVTO. Figure 9 presents the goodput for the high-contended YCSB workloads as the sketch size increases. We vary workload type (read-intensive vs. write-intensive), the number of worker threads varying the amount of data updating the sketch, and the database size determined by the number of keys in the database, representing the key range of workloads.

The main takeaway from these experiments is that regardless of the setting, the sketch can be at most a few KiB, which is less than 1% of the database size, to avoid being a limiting factor since the goodput is similar to that of their memory counterparts.

8 Related Work

Forgetful STO. Bernstein, et al. [4] attempted to reduce the memory footprint of STO by proposing a timestamp purge mechanism. The approach assumes that timestamps are derived from a reasonably precise real-time clock and that transactions are short-lived. Their approach involves selecting a threshold timestamp $(\tau - \delta)$ at τ, purging keys with timestamps below this threshold from memory, tagging these keys with the threshold timestamp, and aborting any transactions whose timestamp is below the threshold. Effectively, this overapproximates the purged keys' timestamps to the threshold timestamp. While Bernstein, et al. argued that this was safe to do, they did not specify any policy for how to select the threshold timestamp that determines which keys get evicted. Such a policy is crucial to balancing memory usage and abort rate. They did not implement or evaluate their scheme.

Bernstein's approach does not meet the requirements of an approximate timestamp storage system defined in Section 4; it sometimes approximates timestamps of keys in use by on-going transactions, aborting those transactions. This works with STO and MVTO but, surprisingly, aborting the transactions of the affected keys is not sufficient to guarantee serializability in TicToc!

Below are two examples that illustrate this issue based on TicToc's validation algorithm [51].

Example 1. Consider these transactions:

Step	TID	Operation	Note
1	T_1	read(k_1)	$k_1.rts = 10$, $k_1.wts = 10$
2	T_2	write(k_1), lock(k_1)	$commit\_ts = 11$
3	T_1	write(k_2), lock(k_2)	$commit\_ts = 12$
4	T_1	abort	$k_1.tuple.rts \leq commit\_ts \wedge$ isLocked(k_1) $\wedge$ k_1 not in W
5	T_2	commit	

Initially, k_1 has $k_1.tuple.rts = 10$ and $k_1.tuple.wts = 10$ in the database. Here, $k_1.tuple.*$ refers to database values, while $k_1.*$ refers to transaction-local values. Let's examine each step in detail:

(1) T_1 reads k_1, recording $k_1.rts = 10$, $k_1.wts = 10$.
(2) T_2 writes to k_1, locks it, and sets $commit\_ts = 11$ during validation.
(3) T_1 writes to k_2, locks it, and sets $commit\_ts = 12$ during validation.
(4) T_1 aborts because another transaction (T_2) is validating k_1. The abort condition is met when:
   - $k_1.tuple.rts = 10 \leq commit\_ts = 12$ (true)
   - k_1 is locked (true)
   - k_1 is not in T_1's write set (true)
(5) T_2 commits with $commit\_ts = 11$.

TicToc maintains serializability by aborting T_1, which would otherwise read outdated data.

However, if timestamp purging occurs during validation, serializability can be violated. Suppose keys with timestamps less than 20 are purged before Step 4, setting $k_1.tuple.rts$ to 20. (This can happen because TicToc processes transactions based on keys accessed by them in a lazy and

distributed manner while the purge system would keep track of the largest timestamp of all keys and pick 20 as a safe threshold timestamp.) Since T_1 already set $commit\_ts = 12$, the validation check will now fail ($k_1.tuple.rts = 20 > commit\_ts = 12$), allowing T_1 to commit with $commit\_ts = 12$. Then T_2 commits with $commit\_ts = 11$. This violates serializability because T_1 reads an old version but is serialized after T_2.

Example 2. Consider these transactions:

Step	TID	Operation	Note
1	T_1	write(k_1), lock(k_1)	$k_1.tuple.rts = 10$
2	T_2	read(k_1)	$k_1.rts = 10$, $k_1.wts = 10$
3	T_2	write(k_1)	
4	T_1	commit	$commit\_ts = 11$, $k_1.tuple.wts = 11$
5	T_2	abort	$k_1.wts \neq k_1.tuple.wts$

Initially, k_1 has $k_1.tuple.rts = 10$ and $k_1.tuple.wts = 10$. Here is an explanation of each step:
(1) T_1 writes to k_1.
(2) T_2 reads k_1, recording $k_1.rts = 10$, $k_1.wts = 10$.
(3) T_2 writes to k_1. ($commit\_ts$ will be 11.)
(4) T_1 commits with $commit\_ts = 11$ and updates $k_1.tuple.wts = 11$.
(5) T_2 aborts because it detects a version mismatch:
- $k_1.tuple.wts = 11 \neq k_1.wts = 10$ (true)

T_2 correctly aborts after detecting a new version through the updated write timestamp. However, if timestamp purging happens after Step 1, purging keys with timestamps below 11 and updating both $k_1.tuple.rts$ and $k_1.tuple.wts$ to 11, serializability is broken. In Step 2, T_2 would read $k_1.rts = 11$, $k_1.wts = 11$. When T_1 commits with $commit\_ts = 11$, T_2 proceeds to commit with $commit\_ts = 12$ (calculated as $k_1.tuple.rts + 1$). Since $k_1.tuple.wts = 11$ matches $k_1.wts = 11$, T_2 cannot detect the version change and incorrectly commits with $commit\_ts = 12$. This breaks serializability because T_2 reads a stale version yet is serialized after T_1.

Therefore, Bernstein's scheme itself without tracking in-use keys is incompatible with TicToc because accurate timestamp data is critical during validation, and TicToc relies on decentralized, fine-grained validation.

On-disk concurrency control. Conventional concurrency control mechanisms, such as two-phase locking (2PL) and optimistic concurrency control (OCC), are widely employed by both academic and commercial disk-based databases [1, 13, 17, 21, 28, 30, 41, 42]. For example, RocksDB supports both pessimistic and optimistic concurrency controls through 2PL and OCC, while Google F1 (Spanner) uses timestamp ordering with locks and OCC. In contrast, there has been significant research on concurrency control mechanisms tailored for in-memory databases [51]. FPSketch brings modern concurrency control algorithms to disk-based transactional databases, including those originally designed for in-memory contexts, instead of relying solely on classical methods.

Acceleration of timestamp accesses. Many disk-based RDBMSs (e.g., Postgres and MySQL) and write-optimized systems (e.g., RocksDB) embed timestamps in their tuple structures, thereby avoiding extra I/O once the relevant pages are in memory. Recent studies on Bf-Tree [23] and Umbra [19] highlight the critical role of efficient page cache management in boosting performance, as on-disk timestamps can be fetched quickly once loaded. By contrast, our approach decouples timestamps from on-disk tuple structures and stores them in FPSketch. This design not only accelerates timestamp operations of CCs but also offers greater opportunities for high cache utilization of records on disk. We believe that transactional systems leveraging FPSketch can attain even stronger performance gains when paired with robust page cache management.

Approximation and summarization in databases. Numerous databases approximate or summarize transaction metadata in order to optimize I/O or reduce memory usage. For example, PostgreSQL maintains hintbits [24] in each tuple, indicating whether the transction that added that tuple has committed, enabling it to avoid a query to its commit log. AWS Aurora maintains Min Read Point LSNs [46], which conservatively indicate the oldest LSN that might still be read, enabling garbage collection of pages with older LSNs. And many MVCC schemes use high watermarks [5] to determine which old versions can be garbage collected.

We believe FPSketch is fundamentally different from prior work because, in all these prior schemes, approximations and summations are used as fast paths (e.g. hintbits) or to drive garbage collection. However, they are not used as part of the CC mechanism itself.

Approximation with sketches. In networking, FlexSwitch [40] uses an approximation technique called min-timestamp to manage packet routing through network switches. However, min-timestamp operates by overwriting conflicting timestamps without guaranteeing a lower bound for these values, allowing overwrites to occur at any time by any entity. In contrast, FPSketch ensures monotonically increasing timestamps. Additionally, it provides high-resolution timestamps that can be shared among active transactions. This not only maintains correctness in timestamp-based concurrency control mechanisms but also optimizes space utilization.

9 Conclusion

In this paper we introduced *approximate timestamp storage*, a novel timestamp storage system that enables timestamp-based CC mechanisms to operate efficiently in modern on-disk key-value stores. Approximate timestmap storage provides guarantees we identified to be suffient for preserving the correctness of the timestmap-based CC mechanisms we studied, while being extremely memory efficient. We presented the design of an approximate timestamp storage system, FPSketch, and implemented two of its variants, Focus-Counter and Focus-Sketch. We integrated FPSketch with three timestamp-based CC protocols–STO, MVTO, and TicToc. Our evaluation shows superior performance across various workloads using our FPSketch approach. Notably, TicToc with Focus-Sketch achieves up to 11× better performance than traditional 2PL and OCC approaches, and up to 27× improvement over disk-based timestamp storage. FPSketch unlocks the ability to adapt existing CC techniques and paves the way for innovative CC mechanisms in future disk-based key-value stores.

We believe FPSketch is broadly applicable across a variety of domains, although the performance benefits may vary based on other overheads in the system. For example, our approach could benefit distributed systems by improving local node performance. In a shared-nothing architecture, a per-partition FPSketch instance could manage timestamps for its corresponding keys, providing the same local memory efficiency and reduced disk I/O demonstrated in our work. In distributed systems or in systems based on older data structures, such as B-trees, other performance bottlenecks will obviously reduce the impact of an optimized concurrency-control mechanism. However, extensive research has focused on optimizing these aspects through techniques like reducing network round-trips and improving global timestamp synchronization. As those orthogonal optimizations are applied, the local performance improvements provided by FPSketch would become even more promising for overall system throughput.

Acknowledgments

This work was supported in part by NSF grant CSR-2212580 and CSR-2504470.

References

[1] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. AsterixDB: A scalable, open source BDMS. *arXiv preprint arXiv:1407.0454* (2014).

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*.

[3] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Betrees and Write-Optimization. *USENIX ;login:* (2015).

[4] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-Wesley Reading.

[5] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable garbage collection for in-memory MVCC systems. *Proc. VLDB Endow.* 13, 2 (Oct. 2019), 128–141. doi:10.14778/3364324.3364328

[6] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Maryland) *(SODA '03)*. Society for Industrial and Applied Mathematics, USA, 546–554.

[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[8] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 19–33. doi:10.1145/3514221.3517879

[9] Howard Chu. 2015. LMDB: The Lightning Memory-Mapped Database Manager. http://www.lmdb.tech/doc/

[10] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1, Article 46 (May 2023), 27 pages. doi:10.1145/3588726

[11] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 49–63.

[12] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55, 1 (April 2005), 58–75. doi:10.1016/j.jalgor.2003.12.001

[13] Couchbase, Inc. 2010. Couchbase. https://www.couchbase.com.

[14] Transaction Processing Performance Council. 2010. *TPC Benchmark C Standard Specification*. Technical Report TPC-C Rev. 5.11.0. Transaction Processing Performance Council. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[16] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 343–360. https://www.usenix.org/conference/osdi23/presentation/eldeeb

[17] Facebook, Inc. 2013. RocksDB. https://github.com/facebook/rocksdb.

[18] Apache Software Foundation. 2019. Apache Cassandra. http://cassandra.apache.org.

[19] Michael Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-optimized multi-version concurrency control for disk-based database systems. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2797–2810.

[20] Steffen Friedrich and Norbert Ritter. 2019. YCSB. In *Encyclopedia of Big Data Technologies*. Springer.

[21] Google, Inc. 2019. LevelDB. https://github.com/google/leveldb.

[22] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data*. 658–670.

[23] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17, 11 (2024), 3442–3455. doi:10.14778/3681954.3682012

[24] Cary Huang. 2024. A Deeper Look Inside PostgreSQL Visibility Check Mechanism. https://www.highgo.ca/2024/04/19/a-deeper-look-inside-postgresql-visibility-check-mechanism/

[25] John Iacono and Mihai Patrascu. 2012. Using hashing to solve the dictionary problem. In *ACM-SIAM Symposium on Discrete Algorithms*. https://api.semanticscholar.org/CorpusID:7469433

[26] Antonios Katsarakis, Vasilis Gavrielatos, and Nikos Ntarmos. 2024. DLHT: A Non-blocking Resizable Hashtable with Fast Deletes and Memory-awareness. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) *(HPDC '24)*. Association for Computing Machinery, New York, NY, USA, 186–199. doi:10.1145/3625549.3658682

[27] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. doi:10.14778/3389133.3389134

[28] Pranab Mazumdar, Sourabh Agarwal, Amit Banerjee, Pranab Mazumdar, Sourabh Agarwal, and Amit Banerjee. 2016. Azure SQL Database. *Pro SQL Server on Microsoft Azure* (2016), 129–156.

[29] MongoDB. 2020. The database for modern applications. https://www.mongodb.com/.

[30] Michael A Olson, Keith Bostic, and Margo I Seltzer. 1999. Berkeley DB.. In *USENIX Annual Technical Conference, FREENIX Track*. 183–191.

[31] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[32] Prashant Pandey, Michael Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance PMEM Hash Tables Through Stability and Low Associativity. In *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data*.

[33] PostgreSQL. 2025. PostgreSQL Source Code: src/backend/access/transam/slru.c. Source Code Repository. https://github.com/postgres/postgres/blob/master/src/backend/access/transam/slru.c Accessed on July 15, 2025.

[34] Shujian Qian and Ashvin Goel. 2024. Massively Parallel Multi-Versioned Transaction Processing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 765–781. https://www.usenix.org/conference/osdi24/presentation/qian

[35] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2023. 2PLSF: Two-Phase Locking with Starvation-Freedom. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 39–51. doi:10.1145/3572848.3577433

[36] Redis. 2022. Redis. https://redis.io/

[37] David P. Reed. 1983. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.* 1, 1 (feb 1983), 3–23. doi:10.1145/357353.357355

[38] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1583–1598. doi:10.1145/2882903.2882958

[39] Karl Rupp. 2018. 42 years of microrocessor trend data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[40] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 67–82. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma

[41] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. In *VLDB*.

[42] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2018. *C-Store: A Column-Oriented DBMS*. Association for Computing Machinery and Morgan & Claypool, 491–518. https://doi.org/10.1145/3226595.3226638

[43] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 809–822. https://www.usenix.org/conference/atc18/presentation/tang

[44] Les Tokar. 2023. Crucial T700 PCIe 5 SSD Review – 12.4GB/s Throughput with over 1.6 Million IOPS. https://www.thessdreview.com/our-reviews/nvme/crucial-t700-pcie-5-ssd-review/3/.

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.

[46] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations

for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. doi:10.1145/3035918.3056101

[47] Stephan Wolf, Henrik Mühe, Alfons Kemper, and Thomas Neumann. 2015. An evaluation of strict timestamp ordering concurrency control for main-memory database systems. In *In Memory Data Management and Analysis: First and Second International Workshops, IMDM 2013, Riva del Garda, Italy, August 26, 2013, IMDM 2014, Hongzhou, China, September 1, 2014, Revised Selected Papers 1*. Springer, 82–93.

[48] Stephan Wolf, Henrik Mühe, Alfons Kemper, and Thomas Neumann. 2015. An evaluation of strict timestamp ordering concurrency control for main-memory database systems. In *In Memory Data Management and Analysis: First and Second International Workshops, IMDM 2013, Riva del Garda, Italy, August 26, 2013, IMDM 2014, Hongzhou, China, September 1, 2014, Revised Selected Papers 1*. Springer, 82–93.

[49] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.* 10, 7 (mar 2017), 781–792. doi:10.14778/3067421.3067427

[50] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proc. ACM Manag. Data* 1, 1, Article 44 (may 2023), 24 pages. doi:10.1145/3588724

[51] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*. 1629–1642.

Received April 2025; revised July 2025; accepted August 2025