High-Performance Transaction Processing in Disk-based Databases

Deukyeon Hwang

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington
2025

Reading Committee:
Simon Peter, Chair
Adriana Szekeres
Ratul Mahajan

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

High-Performance Transaction Processing in Disk-based Databases

Deukyeon Hwang

Chair of the Supervisory Committee:

Simon Peter
Computer Science and Engineering

Achieving high-performance transaction processing in disk-based databases has long required system designers to choose between lock-based concurrency control methods, which suffer from CPU overhead and reduced parallelism, and timestamp-based methods, which provide superior concurrency but incur prohibitive I/O overhead when timestamp metadata is stored on disk. Modern high-speed storage devices like NVMe SSDs exacerbate this trade-off, as the CPU becomes the bottleneck for lock-based methods while disk-based timestamp storage wastes the storage device's speed on frequent small metadata operations.

This dissertation introduces a novel approach that eliminates this fundamental trade-off through approximate timestamp storage and demonstrates that timestamp-based concurrency control protocols—specifically Strict Timestamp Ordering (STO), Multi-Version Timestamp Ordering (MVTO), and TicToc—can maintain correctness (serializability) even when timestamps are overapproximated for inactive keys, as long as active keys maintain exact timestamps throughout their transaction lifetime. This key insight enables designing FPSketch, a hybrid data structure combining a hash table for exact timestamps of active keys with a probabilistic sketch for approximate upper bounds of inactive keys.

The first contribution is the design, implementation, and evaluation of FPSketch integrated with STO, MVTO, and TicToc in the SplinterDB key-value store. FPSketch achieves nearly the idealized performance while requiring only minimal memory—as little as 32KiB for an 80GB database—by eliminating the need to access timestamp metadata from disk during normal operation. Experimental evaluation on modern NVMe

SSDs demonstrates that TicToc with FPSketch achieves up to 14× higher goodput than traditional two-phase locking, up to 5.9× higher goodput than disk-based timestamp storage.

The second contribution is a comprehensive analytical and experimental study evaluating FPSketch across the entire storage performance spectrum, from traditional hard disk drives with millisecond latencies to emerging CXL-based storage approaching DRAM-like speeds. The evaluation reveals that FPSketch's benefits scale with the fundamental gap between local memory and remote storage access, ensuring its continued relevance as storage technology evolves. On slow storage (HDDs and SATA SSDs), FPSketch enables timestamp-based protocols to outperform traditional concurrency control methods: on SATA SSD, TicToc-Focus-Sketch achieves up to 6.89× and 2.52× higher goodput than two-phase locking (2PL) and KR-OCC, respectively, while on HDD it reaches up to 1.8× the goodput of KR-OCC. FPSketch also eliminates the prohibitive overhead of timestamp disk accesses, achieving improvements of up to 569% over disk-based timestamp storage. On fast storage (simulated CXL-based SSDs), where systems transition from I/O-bound to CPU-bound, FPSketch continues to provide substantial benefits by keeping timestamp metadata in fast local memory, enabling timestamp-based protocols to significantly outperform traditional approaches.

Together, these contributions establish that approximate, in-memory metadata management enables high-performance transaction processing for disk-based databases. FPSketch demonstrates that approximate metadata management can unlock advanced concurrency control designs that would otherwise be impractical, providing a practical solution that enables efficient timestamp-based concurrency control across diverse storage technologies while requiring only minimal memory overhead.

Acknowledgements

I would like to express my sincere gratitude to all those who have supported and contributed to this work.

Special thanks to my advisor, Simon Peter, for his invaluable guidance, encouragement, and support throughout this research journey. His insights and mentorship have been instrumental in shaping this dissertation.

I am deeply grateful to my committee members—Adriana Szekeres, Ratul Mahajan, and Radha Poovendran—for their thoughtful feedback, constructive criticism, and dedication to helping me improve this work.

I would also like to acknowledge my collaborators—Alex Conway, Carlos Garcia-Alvarado, Jun Yuan, Naama Ben-David, and Rob Johnson—for their contributions and the fruitful discussions that have enriched this research.

I am particularly indebted to Jonggyu Park, who provided invaluable feedback on my dissertation and helped refine many aspects of this work.

DEDICATION

To My Wife, Gayoung Lee, My Beloved Family, and Friends

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Enduring Challenge of Concurrency Control in On-Disk Databases

Achieving high-performance transaction processing in disk-based databases is challenging because there is an inherent tension between ensuring data consistency and maximizing performance. To safely execute multiple transactions at once, databases rely on Concurrency Control (CC) techniques, which guarantee that the outcome matches some serial order of transactions [9, 14, 20, 21, 34, 35, 40]. As a result, a great deal of research has focused on creating effective CC mechanisms [8, 18, 25, 39, 50, 52, 54, 55]. For decades, selecting an appropriate CC method has required system designers to grapple with a difficult balance.

On one side, traditional lock-based methods like Two-Phase Locking (2PL) [19] are common. These methods do not need much memory because locks are only needed for the data items being used at any moment. However, lock-based methods can make the CPU do a lot of extra work, because transactions need to get and hold locks, sometimes for a long time. This holding of locks can force transactions to wait for each other, which reduces how many can happen at the same time and causes performance problems, especially when there are many transactions.

On the other side are timestamp-based methods, such as Strict Timestamp Ordering (STO) [4], Multi-Version Timestamp Ordering (MVTO) [41], and newer approaches such as TicToc [55]. These methods use timestamps to determine the serialization order of transactions, often achieving higher concurrency and fewer aborted transactions than lock-based approaches. In traditional timestamp ordering schemes like STO

and MVTO, each transaction receives a timestamp when it begins, which determines its position in the serialization order. When accessing a data item, these schemes compare the transaction's timestamp against the item's read and write timestamps: a transaction can read an item only if its timestamp is greater than the item's write timestamp, and can write an item only if its timestamp is greater than the item's read and write timestamps. If an access would violate these ordering rules, the transaction immediately aborts.

TicToc represents a more advanced approach: rather than assigning timestamps at transaction start, it employs a data-driven protocol that computes commit timestamps lazily during validation (the final check for conflicts before a transaction commits). By examining the actual timestamps of all accessed data items when a transaction is ready to commit, TicToc can dynamically find valid logical orderings that allow it to successfully commit transactions that traditional timestamp-based protocols would abort. This flexibility is key to avoiding aborts: for example, a transaction that commits physically after another transaction can receive a logical commit timestamp that places it before the other transaction in the serialization order, computed based on the timestamps of keys accessed by both transactions. In TicToc, a transaction's commit timestamp must be within the range of [write timestamp (wts), read timestamp (rts)] of the version that it read and must be greater than rts of the previous version it wrote. This ability to decouple physical commit order from logical timestamp order ultimately reduces conflicts and improves concurrency.

However, the timestamp-based methods face a significant challenge in disk-based databases: accessing timestamps on disk creates substantial I/O overhead. When these timestamps are stored on disk, even simple operations become expensive. For example, in a STO system [4], reading a data item requires writing its read timestamp back to disk, while updating a row requires first reading the item's existing timestamps before performing the update. This metadata overhead transforms what should be fast, simple operations into slower read-modify-write cycles, significantly reducing system performance.

One might think that modern, fast storage devices like NVMe solid-state drives (SSDs) [49] would solve this problem, but they do not. NVMe SSDs are much faster than older drives, but they actually make this trade-off more obvious. With such fast storage, the delays caused by lock-based protocols become more pronounced, since the system is often waiting for the CPU rather than the storage. At the same time, if timestamp metadata is kept on disk, the SSDs' high speed is wasted dealing with many small metadata I/O operations. As a result, no matter which method is chosen—CPU-heavy locks or I/O-heavy timestamps—the

system ends up limited by either CPU or storage. This makes it hard for designers to fully use the power of new hardware.

This fundamental trade-off raises a deeper question: why must timestamps persist on disk at all? The answer lies in the requirements of timestamp-based concurrency control protocols. When a new transaction begins and accesses a key, it must check that key's timestamp to determine whether the transaction can proceed or must abort to maintain serializability. Moreover, high-performance timestamp-based CC methods including TicToc [55] dynamically find a logical timestamp ordering during the validation phase based on the timestamps of the keys being accessed, resulting in higher concurrency by committing more transactions than traditional timestamp-based CC methods. In both cases, timestamps cannot be discarded simply because no transaction is currently accessing a key, as future transactions will need this information to make correct concurrency control decisions. However, maintaining exact timestamps for all keys in memory is prohibitively impractical for large databases, creating a significant storage space overhead.

The key insight that resolves this dilemma is that, while exact timestamps are required for keys currently being accessed by active transactions, it is sufficient to maintain only an upper bound (a value that is guaranteed to be greater than or equal to the true timestamp) for inactive keys. Timestamp-based CC protocols compare transaction timestamps against key timestamps to make abort decisions. If inactive keys' timestamps were to be completely forgotten, the critical information needed to enforce serializability when new transactions access these keys would be lost. However, by maintaining an overapproximation (a value that may be larger than the true timestamp but never smaller) instead of exact values, correctness is preserved while dramatically reducing memory requirements. This overapproximation may cause harmless extra transaction aborts (when the system assumes a key's timestamp is larger than it actually is), but it will not violate database correctness.

Yet storing approximate timestamps creates an inherent compromise between memory footprint and accuracy. Allocating more space for timestamp storage keeps its values close to the precise timestamps, while shrinking the structure (e.g., by collapsing timestamps of inactive keys into a single shared counter such as the last commit timestamp) rapidly widens the gap between approximate and precise values. The CC protocol must then assume larger timestamps for inactive keys, triggering more conservative abort decisions and reducing the number of committed transactions.

This compromise particularly affects TicToc, whose advantage depends on having access to past timestamp information during validation. As noted earlier, TicToc requires this historical information to dynamically determine correct commit orderings. When keys become inactive and their timestamps are quickly overapproximated, TicToc loses access to the precise historical data it needs. Instead, it must base its validation decisions on overapproximated timestamps, which constrains its ability to find optimal commit orderings. For example, consider a transaction that reads a key whose true timestamps are $wts = 100$ and $rts = 160$, but has been overapproximated to $wts = 150$. TicToc must assume the key was last modified at time 150. If the transaction could have committed with any timestamp in the range $[100, 160]$ when using precise timestamps (since the commit timestamp must be within [wts, rts] of the version it read), the overapproximation reduces this viable range to $[150, 160]$, eliminating 50 possible commit timestamps.

This conservative behavior erodes TicToc's advantage: as overapproximation widens, TicToc's ability to leverage past timestamp information diminishes, making its dynamic timestamp selection less effective and degrading the number of committed transactions (shown in Figure 3.8). This fundamental tension between memory efficiency and timestamp precision motivates the need for a new approach that can maintain sufficiently accurate timestamps for timestamp-based CC protocols while avoiding the prohibitive memory overhead of storing exact timestamps for all keys.

1.2 FPSketch: A New Approach for Timestamp-Based CC on Disk

To address this challenge, the first main contribution presents a new approach for efficient timestamp-based concurrency control in disk-based databases. This approach is called FPSketch. The main idea behind FPSketch is to decompose traditional timestamp-based concurrency control mechanisms into two components: (1) the CC protocol itself (such as STO [4] or TicToc [55]), and (2) an approximate storage system for timestamps, which is called FPSketch.

A key observation enables this decomposition: in reality, only a small fraction of the database keys are being accessed by currently active transactions at any given time. While the system must maintain exact timestamps for active keys to ensure correctness, the vast majority of keys are inactive at any moment. Building on the insight that overapproximation preserves correctness, FPSketch can store approximate upper bounds for these inactive keys with minimal memory overhead, dramatically reducing the storage require-

ments compared to maintaining exact timestamps for all keys.

Based on these observations, FPSketch is designed as a memory-efficient and fast data structure, with two key components:

1. **A Hash Table (Foveated Region):** This part holds the exact timestamps for the keys that are currently active in transactions. A simple counting mechanism ensures that as long as a key is needed by a transaction, its timestamp remains exact.

2. **A Sketch (Peripheral Region):** This component, inspired by structures like the Count-Min Sketch [13], keeps approximate upper bounds for inactive keys. When a key becomes inactive (its reference count falls to zero), its last timestamp is moved from the hash table to this sketch.

Thanks to this hybrid structure, many slow and costly disk accesses for timestamps are replaced by very fast memory operations. In effect, FPSketch gives nearly the same performance as if every timestamp could be kept in fast memory, but it needs much less RAM (for example, only 32KiB for an 80GB database in experiments).

With the problem of metadata I/O removed, high-performance concurrency control protocols can be adapted for disk-based databases. Experiments with the SplinterDB key-value store [12] (detailed in Section 3.4) demonstrate that FPSketch has a large impact: using FPSketch, advanced protocols like TicToc achieve speeds up to 14× faster than traditional 2PL and KR-OCC [31], and up to 5.9× faster than standard disk-based timestamp methods on some workloads. These results demonstrate that FPSketch successfully resolves the tension between memory efficiency and timestamp precision, enabling high-performance timestamp-based concurrency control on disk-based databases.

1.3 Evaluating FPSketch Across the Storage Spectrum

While FPSketch works well on modern NVMe SSDs, these results raise a deeper research question: Is using approximate, in-memory metadata always useful, or is its benefit only clear for today's fast storage technologies like NVMe SSDs? Separating the concurrency control logic from storage performance seems powerful, but this requires evaluation across all types of storage, from traditional, slower devices to the most modern memory-like devices.

The second main contribution addresses this issue directly. It presents a thorough analytical study of the FPSketch approach across a variety of storage devices. The analysis explores how useful the approach remains when the performance gap between memory and disk access changes dramatically. The goal is to determine if the key idea behind FPSketch is broadly applicable and future-proof, or if it is mainly a solution for today's specific hardware.

1.3.1 A Taxonomy of Storage Performance

To make the analysis concrete, the following summarizes the important technical differences among the storage devices considered. These differences are not only in their performance measurements, but also in how each device's architecture interacts with the database and concurrency control systems.

- **Hard Disk Drives (HDDs):** HDDs are mechanical devices limited by moving parts, such as spinning platters and moving arms. As a result, their access times are dominated by seek and rotation delays, leading to average latencies in the 5-20 millisecond range. Their performance for random I/O is poor, typically capable of only 100-200 IOPS [43].

- **SATA SSDs:** These solid-state drives are much faster than HDDs since they have no moving parts. However, common SATA SSDs are limited by the older SATA interface and AHCI protocol, which were designed for spinning disks and do not take full advantage of flash memory's capability to handle many requests in parallel. This puts their latency in the 100-500 microsecond range, and throughput is about 600 MB/s [43].

- **NVMe SSDs:** The NVMe standard connects SSDs directly to the CPU using the PCIe bus and supports high degrees of parallelism (up to 64,000 command queues), leading to another big increase in speed. Newer NVMe SSDs can achieve latencies as low as 20-100 microseconds, random read IOPS above 1 million, and sequential throughput over 7,000 MB/s [49]. This was the platform where FPSketch was first tested.

- **CXL-based SSDs:** Compute Express Link (CXL)-based SSDs is the newest type of storage that tries to remove the distinction between memory and storage [53, 56]. These devices connect through PCIe but use the CXL.mem protocol, meaning the CPU can read their contents through direct memory

instructions, like main memory. This eliminates traditional storage software overhead and the device looks like a memory-only node. Access is not as fast as regular DRAM, but much faster than NVMe SSDs, with latency in the single-digit microsecond range.

These major differences are shown in Table 1.1, which provides the performance data used in the analyses that follow.

Table 1.1: Comparative Performance Characteristics of Storage Media [43, 49, 53]

	HDD	SATA SSD	NVMe SSD	CXL-based SSD
Latency	5-20 ms	100-500 µs	20-100 µs	1-10 µs
Sequential Throughput	150-250 MB/s	600 MB/s	7,000 MB/s	16 GB/s (PCIe 5.0 x4)
Interface/Protocol	SATA / ATA	SATA / AHCI	PCIe / NVMe	PCIe / CXL.mem
Primary Bottleneck	Mechanical (Seek/Rotation)	Interface/Protocol	CPU / Concurrency	Interconnect / Memory Controller

1.3.2 Slow Storage Environments

First, consider using FPSketch in HDDs and SATA SSDs. In these systems, each random I/O operation is extremely slow, measured in milliseconds for HDDs and hundreds of microseconds for SATA SSDs. Traditional approaches for disk-based databases, such as 2PL and OCC, have been the standard choice for these environments because they avoid the metadata I/O overhead. On slow storage devices, their CPU overheads are largely hidden by the dominant cost of disk I/O, making these methods acceptable despite their limitations. A simple approach that stores timestamps on disk would therefore be severely limited by slow disk I/O delays, making high-performance concurrency control protocols impractical. FPSketch's ability to keep timestamps in memory becomes even more critical and outperforms traditional approaches in these environments by eliminating the need for slow disk I/O operations.

1.3.3 Fast Storage Environments

Second, consider emerging CXL-based SSDs, which represent a fundamental shift in storage architecture. CXL (Compute Express Link) enables byte-addressable storage that appears to the CPU as a remote NUMA node, allowing direct memory access without the traditional I/O stack. CXL-based SSDs offer latencies in the low microsecond range—much faster than block devices, yet still incurring remote memory commu-

nication overhead. Since CXL-based SSDs are still emerging technologies, the evaluation simulates their performance rather than using physical hardware.

In this fast storage environment, FPSketch's purpose shifts from avoiding slow disk I/O to maintaining critical metadata in fast local memory. If timestamps were stored directly on the CXL-based SSD, each access would incur several microseconds of remote memory communication latency. Even simple protocols like TicToc require many such accesses per transaction, meaning these small delays accumulate and significantly impact performance.

FPSketch solves this problem by acting as an application-specific cache for concurrency metadata, with its hash table storing the exact timestamps for active keys close to the CPU in local DRAM. This ensures frequently used metadata is always available with the lowest possible delay. While FPSketch avoids the performance bottlenecks that arise from remote memory access, it introduces its own overhead from hash table operations and memory allocation. On fast storage, these overheads become visible relative to storage access costs, yet FPSketch still provides substantial benefits by keeping metadata in fast local memory and enabling timestamp-based protocols to significantly outperform traditional approaches.

This analysis shows that FPSketch is more than just a new data structure; it is, in fact, a flexible method for managing metadata in a tiered memory/storage world. Its value becomes even greater as the distinction between memory and storage fades.

1.4 Thesis Statement

The main thesis is: ***Approximate, in-memory metadata management enables high-performance transaction processing for disk-based databases.*** Its advantages do not depend on any single storage technology; instead, they scale with the basic, long-lasting gap between local computation and remote storage access, allowing advanced concurrency control protocols to reach nearly optimal performance across all storage types. First, the design, implementation, and evaluation of FPSketch using modern NVMe SSDs demonstrates this. Then, an analytical study of storage devices from slow, mechanical drives to emerging high-speed, memory-like storage shows its wider value and ongoing relevance.

1.5 Dissertation Outline

This dissertation builds on the following publication:

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`

The dissertation is organized as follows:

Chapter 2: Background and Motivation

This chapter provides a full background on timestamp-based concurrency control protocols, modern storage technologies, and the performance features that motivate the research.

Chapter 3: FPSketch

This chapter presents the design and implementation of FPSketch, which is the first main contribution. It describes the method for approximate timestamp storage, the hybrid hash table and sketch data structure, and how it integrates with STO, MVTO, and TicToc protocols. It also provides a thorough experimental evaluation of FPSketch on a state-of-the-art NVMe SSD, comparing its performance to traditional concurrency control methods and ideal baselines across various workloads.

Chapter 4: Evaluating FPSketch Across the Storage Spectrum

This chapter presents the second primary contribution: a detailed analysis of the FPSketch paradigm across legacy (HDD, SATA SSD) and new (CXL-SSD) storage technologies, examining its impact and evolving role.

Chapter 5: Related Work

This chapter reviews related research on disk-based concurrency control, approximate data structures in databases and other areas.

Chapter 6: Conclusion and Future Work

This chapter summarizes the dissertation's contributions, restates the main thesis, and outlines possible directions for future study.

Chapter 2

Background and Motivation

This chapter provides the theoretical foundation and practical motivation for approximate timestamping in on-disk database systems. It examines the fundamental principles of concurrency control in transactional systems, then traces the evolution of storage technology that has reshaped performance bottlenecks, introduces a representative approximate data structure, and finally presents the compelling case for why approximate timestamping represents a necessary innovation.

2.1 Concurrency Control in Transactional Systems

Concurrency control is the set of mechanisms that ensure multiple transactions can execute simultaneously while maintaining data consistency. To understand why this is crucial, this section first examines the fundamental properties that all transactional systems must guarantee.

2.1.1 The ACID Properties and the Need for Concurrency Control

Transactional database systems are built upon four fundamental guarantees, collectively known as the ACID properties [26]: Atomicity, Consistency, Isolation, and Durability. These properties ensure that database operations behave reliably even when multiple users access the system simultaneously.

Atomicity ensures that each transaction is treated as a single, indivisible unit. Either all operations within a transaction complete successfully, or none of them do. If any part of a transaction fails, the entire transaction is rolled back, leaving the database in its original state.

Consistency guarantees that a transaction brings the database from one valid state to another. All data integrity rules and constraints are maintained, ensuring that the database remains in a correct state after each transaction.

Durability ensures that once a transaction commits, its effects are permanent and will survive system failures. Committed data is stored persistently and can be recovered even after crashes or power outages.

Isolation is perhaps the most complex property and the primary focus of concurrency control. It ensures that concurrent transactions do not interfere with each other, making it appear as if transactions execute one at a time, even when they actually run simultaneously. This property is crucial because without proper isolation, concurrent transactions could see partial results from each other, leading to inconsistent data states.

Concurrency control mechanisms are specifically designed to enforce the isolation property. They ensure that concurrent transactions result in a system state that is equivalent to some serial execution of those transactions—a property called *serializability*. This means that even though transactions may run at the same time, the final result is the same as if they had executed one after another in some order.

2.1.2 A Taxonomy of Concurrency Control Protocols

Concurrency control protocols can be broadly categorized into three main families, each with distinct approaches to managing concurrent access to data. Understanding these approaches is essential for appreciating the trade-offs involved in different concurrency control strategies.

Lock-Based Concurrency Control

Lock-based concurrency control operates on the principle of preventing conflicts before they occur. These protocols assume that conflicts are likely and take proactive measures to avoid them by controlling access to data items.

Two-Phase Locking (2PL) [19] is the most widely used lock-based protocol. It operates in two distinct phases. During the *Growing Phase*, a transaction can acquire locks on data items but cannot release any locks. This phase continues as the transaction reads and writes data. Once the transaction enters the *Shrinking Phase*, it can only release locks and cannot acquire new ones. This phase continues until the transaction completes.

This two-phase rule ensures that no other transaction can access a data item between the time when a transaction reads it and when it writes to it, preventing certain types of inconsistencies. This property, combined with the requirement that all locks are held until commit, enables 2PL to prevent cascading aborts: once a transaction commits, no other transaction that read its data needs to be aborted. However, these benefits come at a cost: managing locks requires significant computational resources, including maintaining lock tables and handling lock requests, resulting in high CPU overhead. When transactions wait for each other's locks, deadlocks can occur, requiring detection and resolution mechanisms. These deadlock risks are particularly problematic in high-contention scenarios, where many transactions may be blocked waiting for locks, reducing overall system throughput and limiting parallelism.

To address these deadlock issues, practical implementations often use policies like *no-wait* [42, 47], where transactions immediately abort if they cannot acquire a needed lock, rather than waiting and potentially creating deadlocks.

Optimistic Concurrency Control

Optimistic concurrency control (OCC) [31] takes a fundamentally different approach to handling concurrent transactions. Rather than using locks to prevent conflicts from occurring, OCC assumes that conflicts are rare and permits transactions to proceed in parallel without synchronization during most of their execution.

A typical OCC protocol consists of three phases. During the *Read Phase*, a transaction reads data and performs computations, keeping track of the data items it accesses but making changes only to private, local copies. When the transaction is ready to commit, the *Validation Phase* begins, where the system checks whether the data the transaction read has been modified by other transactions since it started. If so, the transaction must abort and is typically retried. If validation succeeds, the *Write Phase* applies the transaction's changes to the database atomically.

There is no blocking on locks, so deadlocks cannot occur. Transactions operate freely during the read phase, maximizing parallelism and providing high concurrency. When conflicts are rare, aborts are infrequent and performance is high, making OCC efficient for low-contention workloads. However, in workloads with frequent conflicting accesses, many transactions may be aborted and retried, reducing throughput due to high abort rates under contention. The process of checking for conflicts at commit time incurs compu-

tational overhead, especially as the number of transactions scales. Careful protocol design is required to prevent subtle anomalies like write skew in some OCC implementations.

Overall, OCC is well-suited to environments where transactions are short and conflicts are rare, such as many in-memory and read-mostly workloads.

Timestamp-Based Concurrency Control

Timestamp-based protocols use temporal ordering to determine the serialization order of transactions, assigning each transaction a unique timestamp that determines when it should appear to execute relative to other transactions.

Strict Timestamp Ordering (STO) [4] is a traditional good timestamp-based protocol. Each transaction receives a timestamp when it starts, and this timestamp determines its position in the serialization order. When accessing a data item, the transaction compares its timestamp against the item's read and write timestamps. If the access would violate the timestamp ordering rules, the transaction immediately aborts.

STO uses both read and write timestamps for each data item, allowing higher concurrency than single-timestamp approaches. A transaction can read an item if its timestamp is greater than the item's write timestamp, and can write an item if its timestamp is greater than the item's read timestamp.

Multi-Version Timestamp Ordering (MVTO) [41] extends timestamp ordering by maintaining multiple versions of each data item. This allows read-only transactions to access older, consistent snapshots without conflicting with concurrent write transactions. MVTO is particularly effective for read-heavy workloads where many transactions need to read data while fewer transactions perform updates.

However, MVTO introduces significant overhead in terms of storage space (for maintaining multiple versions) and garbage collection (for removing old versions that are no longer needed).

TicToc (Optimistic Timestamp Ordering) [55] represents an advanced approach that combines the benefits of timestamp ordering with optimistic execution. Instead of assigning timestamps at transaction start, TicToc dynamically chooses a commit timestamp during the validation phase. By selecting a timestamp that is guaranteed to be valid with respect to all accessed data, TicToc minimizes the time window for potential conflicts, significantly reducing abort rates compared to traditional optimistic protocols.

2.2 The Evolution of On-Disk Storage Systems

This section examines how advances in storage technology have fundamentally transformed the performance characteristics of database systems. The section begins by examining how the performance bottleneck has shifted from I/O latency to CPU overhead, then traces the evolution of storage media from hard disk drives to modern NVMe SSDs, and finally introduces CXL as an emerging technology that bridges storage and memory semantics.

2.2.1 The Shifting Performance Landscape

For decades, mechanical storage (HDDs) imposed millisecond-scale latency that dominated end-to-end performance, masking other sources of overhead such as the CPU cost of concurrency control. With the advent of solid-state storage—and especially NVMe SSDs—access latencies dropped to tens of microseconds, exposing non-I/O bottlenecks, most notably CPU overhead from concurrency control protocols.

Hard Disk Drives (HDDs) represent the traditional storage technology that dominated database systems for decades. Their performance is fundamentally limited by mechanical factors: the seek time required to move the read/write head to the correct track, and the rotational latency needed for the desired data to rotate under the head. These mechanical limits yield very low random I/O performance, typically only 100–200 input/output operations per second (IOPS) with access times of several milliseconds [43]. HDDs remain in use due to low cost and high capacity.

SATA Solid-State Drives (SSDs) eliminated mechanical delays but were constrained by the legacy SATA III interface, which was designed for single-queue spinning disks. The SATA interface caps bandwidth at approximately 600 MB/s, and the AHCI protocol limits parallelism. Despite these constraints, SATA SSDs achieve latencies in the hundreds of microseconds and can support up to approximately 100,000 IOPS—a significant improvement over HDDs [43].

NVMe Solid-State Drives (SSDs) represent the current state-of-the-art, designed from the ground up for flash memory. NVMe utilizes the high-bandwidth, low-latency PCIe bus and supports massive parallelism with up to 64,000 command queues. Modern Gen4 NVMe drives can achieve [49] throughput exceeding 7 GB/s, over 1 million IOPS, and latencies in the tens of microseconds.

2.2.2 Compute Express Link (CXL): Accelerating SSDs with Memory-Semantic IO

Compute Express Link (CXL) [10, 16] is a cache-coherent protocol atop PCIe that unifies CPUs, accelerators, and memory devices into a single memory-centric domain. It enables shared, coherent load/store access between host and devices, reducing synchronization and data-movement overheads, and it scales via CXL switches to attach additional devices without redesign. In particular, Type-3 (memory-expansion) devices expose host-managed device memory (HDM) that the CPU accesses directly with ordinary loads and stores [16].

In the context of Type-3 memory expansion, CXL can accelerate SSDs by bringing them into the cache-coherent domain, overcoming fundamental limitations of traditional PCIe storage [29, 53]. Historically, while PCIe devices could map internal memory through Base Address Registers (BARs), the host had to treat such accesses as non-cacheable, preventing use of CPU caches and degrading performance. With CXL, load/store requests to a CXL-attached SSD's address space—mapped into the host's cacheable system memory—become cacheable. Incorporating PCIe storage into the coherent hierarchy bridges block semantics to byte-addressable, memory-compatible semantics and substantially lowers latency: projections report up to 129.5$\times$ lower latency than PCIe-based memory expanders due to host cache usage [29], and CXL-enabled SSDs can serve 68–91% of memory requests in under a microsecond [53].

2.2.3 Modern Write-Optimized Data Structures

The evolution of storage technology has driven the development of new data structures designed to exploit the characteristics of solid-state storage.

Log-Structured Merge (LSM) trees [36] and similar write-optimized structures have become increasingly popular for on-disk storage systems. These structures are designed to maximize write performance by batching and organizing writes efficiently.

A key property of these structures is the Query/Insert Asymmetry [3]: they can ingest new data (writes) orders of magnitude faster than they can service random queries (reads). For example, SplinterDB [12], which uses a write-optimized structure called a B^{ε}-trees [6], can perform over 2 million random insertions per second but only about 500,000 random queries per second.

This asymmetry is crucial for understanding the performance implications of different concurrency con-

trol approaches. When timestamp-based protocols store timestamps in write-optimized structures, they may need to perform a slow query operation to fetch existing timestamps before each fast write operation, effectively negating the performance benefits of the write-optimized structure.

2.3 Approximate Data Structures

Approximate data structures are a class of data structures that provide an approximation of the data stored in the data structure. This approximation is used to reduce the memory requirements of the data structure.

2.3.1 Count-Min Sketch

The count-min sketch (CMS) [13] 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.

2.4 Motivation: The Case for Approximate Timestamping

The preceding discussions on concurrency control and storage systems converge to create a compelling argument for why approximate timestamping represents a necessary innovation for modern database systems.

2.4.1 The Concurrency Control Bottleneck Revisited

With the dramatic reduction in I/O latency provided by NVMe SSDs, the CPU overhead inherent in lock-based protocols like 2PL is no longer masked by storage delays. This overhead now stands out as a primary factor limiting system throughput, making computationally efficient protocols like timestamp ordering theoretically more appealing.

The computational efficiency of timestamp-based protocols comes from their ability to make concurrency control decisions based on simple timestamp comparisons rather than complex lock management. However, applying these protocols directly to on-disk systems introduces new challenges.

2.4.2 The Prohibitive Cost of Naive On-Disk Timestamps

The critical flaw in applying timestamp ordering directly to on-disk systems lies in the storage requirements for timestamps. Storing per-key timestamps on disk creates a new, I/O bottleneck that can negate the performance benefits of timestamp-based protocols.

This problem is particularly acute when combined with the Query/Insert Asymmetry of write-optimized data structures. A fast write operation in an LSM-tree becomes a slow read-modify-write cycle when timestamps are stored on disk, as the system must first perform a slow query to fetch the key's existing timestamps before the fast write can proceed. This disproportionately degrades write performance and eliminates the primary advantage of write-optimized structures.

2.4.3 The Impracticality of a Fully In-Memory Solution

The obvious alternative of storing all timestamps in RAM is not viable for large-scale database systems. Real-world deployments, such as Facebook's RocksDB [20] instances, demonstrate that the memory required for timestamps alone could consume a substantial portion of available RAM.

For example, with key-value pairs averaging less than 100 bytes [2, 7], storing two 8-byte timestamps per pair would require approximately 16% of the data size in additional memory. However, many production systems operate with RAM-to-disk ratios of less than 5%, sometimes as little as 3% [7]. Even in scenarios with larger key-value pairs or more abundant RAM, timestamps would still consume a significant portion of memory that could otherwise be used for caching or other purposes.

2.4.4 Case Study: The Promise and the Pitfall

Figure 2.1 evaluates YCSB workloads, modified to execute transactions in parallel, spanning contention (high, medium) and access patterns (read- vs. write-intensive). It reports goodput, the throughput of successfully committed transactions, to discount wasted work from aborts. The evaluation compares 2PL, OCC,

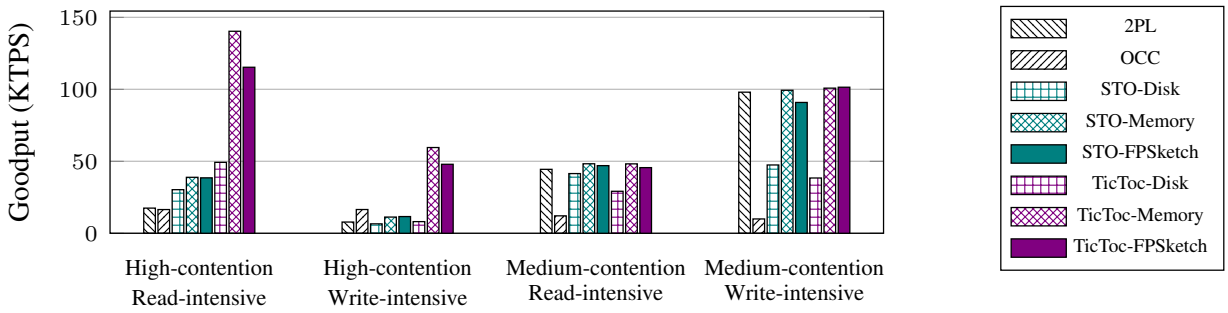


Figure 2.1: Performance comparison of various concurrency control mechanisms across different YCSB workloads, demonstrating the promise of in-memory timestamp ordering and the pitfalls of disk-based approaches. (Running 120 threads on NVMe SSD)

and two timestamp-based methods (STO and TicToc), each under three timestamp placement strategies: stored on disk, in an idealized in-memory exact timestamp store, and the proposed approximate timestamp store (FPSketch with 32KiB). This bracketing (Disk vs. Memory) with FPSketch in between isolates the core tradeoff—CPU efficiency of timestamp-based methods versus the I/O and lookup costs of materializing per-key timestamps on disk—while holding hardware constant. The observed goodput gaps motivate approximate timestamping to achieve near-in-memory performance with a minimal memory footprint.

The Promise (TicToc-Memory): When timestamps are stored in an idealized in-memory hash table, advanced protocols like TicToc dramatically outperform traditional 2PL and OCC. TicToc-Memory achieves the highest throughput across all workloads, with particularly impressive performance in high-contention scenarios, demonstrating the immense potential of timestamp-based schemes.

The Pitfall (TicToc-Disk): When these same timestamps are stored on disk, the I/O overhead becomes so crippling that performance collapses, often falling far below that of 2PL and OCC, especially in write-intensive workloads. The performance gap between Memory and Disk variants is particularly pronounced for write-intensive workloads, where the Query/Insert Asymmetry has the greatest impact.

The Opportunity: This stark contrast between the in-memory and on-disk variants perfectly frames the research problem. It establishes a clear need for a solution that can deliver the high performance of the in-memory approach without its prohibitive memory cost. The results from STO-FPSketch and TicToc-FPSketch, which use the proposed approximate timestamping technique, demonstrate that such a solution is

achievable, setting the stage for the detailed presentation of this approach in the following chapters.

Chapter 3

FPSketch: Approximate Timestamp Storage for Timestamp-Based Concurrency Control

This chapter introduces the concept of *approximate timestamp storage* for enabling efficient timestamp-based concurrency control (CC) mechanisms in modern on-disk key-value stores. Exact timestamp storage systems are costly to maintain at large scale, especially when data does not fit entirely in main memory. This chapter details the key properties that an approximate timestamp storage system must possess to preserve correctness—specifically serializability—for three common CC protocols: STO [4], MVTO [41], and Tic-Toc [55]. It further describes the design and implementation of FPSketch, a novel approximate timestamp storage approach. Through careful analysis, it demonstrates that the overapproximation guarantees provided by FPSketch are sufficient for correctness, while significantly improving efficiency and scalability. This approach unlocks new possibilities for optimizing disk-based storage systems without sacrificing correctness guarantees required by transactional workloads.

3.1 Approximate Timestamp Storage

Timestamp-based CC mechanisms typically assume a *map* abstraction for storing read and write timestamps, where each *get* returns the value from the most recent *put*. For STO, MVTO, and TicToc, such strong consistency guarantees are not required to ensure serializability. Instead, an *approximate timestamp storage* system satisfying the following two properties is sufficient to enable all three CC mechanisms to function correctly without any algorithmic modifications.

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. A simulation argument fomalizes this intuition. 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. If it can be shown 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 [55] continues to work, even for approximate timestamp storage. This is because the proof reasons almost entirely about timestamp inequalities across physical time, and approximate timestamp storage will preserve those inequalities. Only the following lemma is needed 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. This proof establishes 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.

The next section presents the design of an approximate timestamp storage data structure that meets the above requirements.

3.2 FPSketch

This section describes FPSketch, an approximate timestamp storage structure that meets the requirements identified in Section 3.1.

As shown in Figure 3.1, 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.

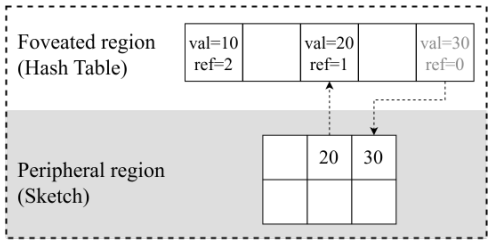


Figure 3.1: FPSketch consists of the foveated region (hash table) storing accurate timestamps for active keys, peripheral region (sketch) for approximated timestamps, and the eviction mechanism transferring data between them.

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: The system does not discard timestamps evicted from the hash table; instead, it stores them 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

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 t_1, TIMESTAMPS t_2)
2:      (v.wts, v.rts) ← (MAX(t_1.wts, t_2.wts), MAX(t_1.rts, t_2.rts))
3:      return v
4:  end procedure
5:  procedure TIMESTAMPMIN(TIMESTAMPS t_1, TIMESTAMPS t_2)
6:      (v.wts, v.rts) ← (MIN(t_1.wts, t_2.wts), MIN(t_1.rts, t_2.rts))
7:      return v
8:  end procedure
9:  procedure PUT(key k, value v)
10:     for row = 0 to rows − 1 do
11:         col ← HASH(k, seeds[row]) mod cols
12:         current_value ← ATOMICLOAD(&table[row][col])
13:         is_success ← false
14:         while !is_success do
15:             new_value ← TIMESTAMPMAX (current_value, v)
16:             is_success ← ATOMICCOMPAREEXCHANGE(&table[row][col], &current_value, new_value)
17:         end while
18:     end for
19: end procedure
20: procedure GET(key k)
21:     col ← HASH(k, seeds[0]) mod cols
22:     value ← ATOMICLOAD(&table[0][col])
23:     for row = 1 to rows − 1 do
24:         col ← HASH(k, seeds[row]) mod cols
25:         current_value ← ATOMICLOAD(&table[row][col])
26:         value ← TIMESTAMPMIN (current_value, value)
27:     end for
28:     return value
29: end procedure
```

shown in Algorithm 1. Note that field-wise $\max$ and $\min$ preserve that $wts \leq rts$, so the 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.

3.2.1 FPSketch Operations

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:     end if
10: end procedure
11: procedure DECREF(key k)
12:     e ← hashtable.GETITEMPTR(k)                       ▷ e cannot be NULL
13:     e.refcount−−
14:     if e.refcount = 0 then
15:         EVICT()
16:     end if
17: end procedure
18: procedure GET(key k)
19:     return hashtable.GETITEMPTR(k).ts
20: end procedure
21: procedure PUT(key k, timestamp ts)
22:     hashtable.GETITEMPTR(k) .ts ← ts
23: end procedure
24: procedure EVICT()
25:     E ← {item in hashtable | item.val.refcount = 0}   ▷ Only choose to evict from the set of keys that are not
    in-use
26:     for all item in TOEVICTPOLICY(E) do
27:         sketch.PUT(item.k, item.val.ts)
28:         hashtable.REMOVE(item.k)
29:     end for
30: end procedure
```

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 must 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 the pseudocode omits locking, all operations ensure linearizability for each item in the hash table through 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 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 return or update the timestamps in the hashtable entry, which is guaranteed to exist due to the prior call to INCREF.

3.2.2 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. FPSketch has two 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 detailed further below. For Focus-Sketch, intuition is provided on how to size the sketch (in terms of the number of rows and columns) in the next section.

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.

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.

3.2.3 Sizing FPSketch

This section presents 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 $O(CK \log CK)$.

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. It remains 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, a resizable hashtable is recommended, allowing it to grow as needed. Several state-of-the-art hashtables support concurrent operations while resizing [30, 32, 37], so this need not cause any hiccups in throughput.

3.2.4 FPSketch Implementation

The implementation builds the hash table of FPSketch on top of IcebergHT [37] and constructs the sketch from scratch. The implementation modifies 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, the implementation uses an optimized version of Algorithm 1 that no longer needs to compute hashes.

3.3 Integration with the Existing CC Algorithms

This section describes 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).

3.3.1 TicToc

Algorithm 3 shows the pseudocode for TicToc with FPSketch. There are only two changes to the algorithm. First, the system stores timestamps in the FPSketch instead of inline with the tuples, as in the original TicToc paper, so the syntax for accessing timestamps differs. Second, INCREF and DECREF must be called when a transaction first accesses a key and when it finishes. Note that the FPSketch code encapsulates all the logic for dealing with timestamp approximation—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

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.

procedure READ(database D, FPSketch S, transaction T, key k)
  begin atomic section
    if $k \notin T.R$ **then**
      S.INCREF(k)
    end if
    $T.R[k] \leftarrow S$.GET(k)
    $T.R[k].data \leftarrow D$.READ(k) ▷ Read k's value from disk
  end atomic section
  return $T.R[k].data$
end procedure
procedure VALIDATE(transaction T)
  for $k \in T.W$ in sorted order **do**
    LOCK k
    S.INCREF(k)
  end for
  $T.\tau \leftarrow 0$ ▷ Compute the commit timestamp $T.\tau$
  for $k \in T.W \cup T.R$ **do**
    if $k \in T.W$ **then**
      $T.\tau \leftarrow \max(T.\tau, 1 + S.\text{GET}(k).rts)$
    else
      $T.\tau \leftarrow \max(T.\tau, T.R[k].wts)$
    end if
  end for
  for $k \in T.R$ **do**
    begin atomic section
      if $T.R[k].rts < T.\tau$ **then**
        if $T.R[k].wts \neq S.\text{GET}(k).wts \vee (S.\text{GET}(k).rts \leq T.\tau \wedge \text{LOCKED}(k) \wedge k \notin T.W)$ **then**
          ABORT
        else
          $curr \leftarrow S.\text{GET}(k)$
          $curr.rts \leftarrow \max(T.\tau, curr.rts)$
          $S.\text{PUT}(k, curr)$
        end if
      end if
    end atomic section
  end for
end procedure
procedure COMMIT(database D, FPSketch S, transaction T)
  for $k \in T.W$ **do**
    $D.\text{WRITE}(k, T.W[k].data)$ ▷ Write k's value to disk
    $S.\text{PUT}(k, (T.\tau, T.\tau))$
    S.DECREF(k)
    UNLOCK k
  end for
  for $k \in T.R$ **do**
    S.DECREF(k)
  end for
end procedure

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, the specified key-value pair is recorded 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 the pseudocode, in the 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 the code for finalizing transactions upon abort is omitted due to space constraints. However, the steps involved in unlocking and removing entries for transaction aborts are the same as those outlined in Line 43 and Line 42.

3.3.2 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. Read-write locks achieve this, as illustrated in Algorithm 4. The implementation uses "dirty bits"[52] for these locks. 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 dirty bit only as a latch to atomically read the tuple into a private buffer, and hence do not block subsequent writers (or subsequent readers).

As in the TicToc implementation, the entries in the FPSketch store the reader-writer locks. 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.

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

procedure BEGIN(global timestamp $*c$, transaction T)
    $T.\tau \leftarrow$ ATOMICFETCHADD(c, 1)
end procedure
procedure READ(database D, FPSketch S, transaction T, key k)
    if $k \notin T.R$ **then**
        READLOCK k
        $T.R \leftarrow T.R \cup \{k\}$
        S.INCREF(k)
        if $T.\tau < S$.GET(k).wts **then**
            ABORT
        end if
        begin atomic section
            $curr \leftarrow S$.GET(k)
            $curr.rts \leftarrow \max(T.\tau, curr.rts)$
            S.PUT($k, curr$)
        end atomic section
        $T.R[k].data \leftarrow D$.READ(k) ▷ Read k's value from disk
        UNLOCK k
    end if
    return $T.R[k].data$
end procedure
procedure WRITE(database D, FPSketch S, transaction T, key k, val v)
    if $k \notin T.W$ **then**
        WRITELOCK k
        S.INCREF(k)
        if $(T.\tau < S$.GET(k).$wts) \vee (T.\tau < S$.GET(k).$rts)$ **then**
            ABORT
        end if
        $curr \leftarrow S$.GET(k)
        $curr.wts \leftarrow \max(T.\tau, curr.wts)$
        S.PUT($k, curr$)
    end if
    $T.W[k] \leftarrow v$
end procedure
procedure COMMIT(database D, FPSketch S, transaction T)
    for $k \in T.R$ **do**
        S.DECREF(k)
    end for
    for $k \in T.W$ **do**
        D.WRITE(k, $T.W[k]$) ▷ Write k's value to disk
        S.DECREF(k)
        UNLOCK k
    end for
end procedure

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, the timestamps in the sketch must be read and written 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.

3.3.3 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 noted in Section 3.1, MVTO requires that the approximate timestamp storage system must maintain the relative timestamp ordering of the versions of a key-value pair. This requirement is satisfied by ensuring that the system approximates the timestamps of only the most recent version of any key-value pair.

Specifically, the MVTO implementation works as follows. When a key is inactive, the system stores the (approximate) timestamps of the most recent version of that key in FPSketch. When the key becomes active, the system copies the timestamps of the most recent version from FPSketch into the hashtable. Whenever a new version of the key is created, the system stores the timestamps of the new version in the hashtable, as well. Once the key becomes inactive, the system moves the timestamps of the most recent version back to FPSketch and discards the timestamps of all the older versions.

Whenever a new version is created, the system assigns it a monotonically increasing version number and stores it 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. Two solutions to this problem can be considered. The first is 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 [11].

To avoid predecessor queries, the following optimization is viable. Whenever a new version of a key-value pair is inserted into SplinterDB, the same key-value pair is also inserted into SplinterDB with the special version number 0, i.e. 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, this approach offers higher performance, so this is the scheme used in all the benchmarks.

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

3.4 Evaluation

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

- FPSketch-based versions of STO, MVTO, and TicToc outperform versions that keep timestamps on disk in all the experiments, often by dramatic margins. For example, FPSketch improves TicToc goodput by up to 5.9×, STO by up to 1.8×, and MVTO by over 160×.
- Similarly, Focus-Sketch-based protocols can offer up to 2× 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

all timestamp metadata can fit 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 the 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.

3.4.1 Experimental Setup

The experiments run on a CloudLab [17] Clemson r6525 machine with two 32-core AMD 7543 at 2.8GHz, 256GB memory, and one 1.6TB PCIe v4.0 NVMe SSD.

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

All the experiments run with 120 threads, which is the maximum supported by the version of SplinterDB used. 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. The initial backoff interval for each scheme is tuned by initially setting it to be roughly twice the average uncontended transaction completion time, and then tuning it until the abort rate roughly matches that of the Memory variant.

The evaluation measures goodput in Kilo Transactions Per Second (KTPS) and also evaluates the abort rate. Goodput represents the number of successfully committed transactions per second. The evaluation calculates the abort rate 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

The evaluation implements and tests several CC mechanisms, including all three timestamp-based CCs described in Section 2.1.2 (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. After evaluating multiple deadlock prevention mechanisms (including wound-wait and wound-die), 2PL employs a no-wait policy because of its generally good performance, which aligns with findings from previous works [42, 47].

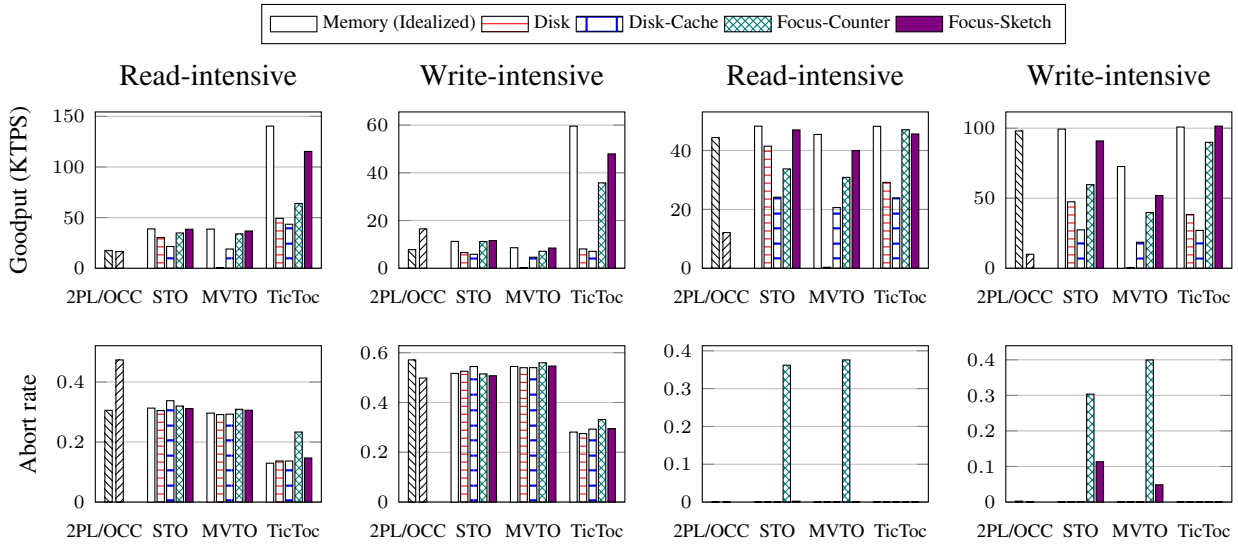
The evaluation implements five variants of STO, MVTO, and TicToc. These variants are distinguished by their methods for storing and accessing per-record timestamps, which range from relying entirely on disk to using fully in-memory solutions.

First, two baseline variants establish the performance boundaries:

- *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.
- *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 [38], 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. The impact of larger caches is also explored in Section 3.4.2.
- *Focus-Counter*: As one variant of FPSketch, described in Section 3.2.2, 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 [44], with the number of columns set to fit the memory budget.



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

Figure 3.2: 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-contention workload, several schemes have abort rates very close to 0.

Workloads

The evaluation uses two benchmarks to test the performance of the implemented CCs: (1) YCSB [23], which models large-scale online services, and (2) TPC-C [15], the industry standard for evaluating OLTP systems.

In the YCSB workloads, the evaluation first loads a dataset of 673 million key-value pairs, with 24-byte keys and 100-byte values. It performs this loading phase before every run, followed by the execution phase. To evaluate FPSketch across diverse scenarios, the evaluation executes 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, a high-contention (Zipfian 0.9) "large" transaction workload is run, 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. YCSB experiments run for 240 seconds. Each run is repeated three times and average numbers are reported.

As in many previous works, the evaluation considers TPC-C workloads comprising 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 this evaluation. The TPC-C workloads run for various numbers of warehouses, 4, 8, 16, and 32, which dictate the initial database size and the contention level. SplinterDB's internal cache is configured to 256MB. The TPC-C experiments run for 120 seconds and repeated three times to minimize noise caused by random variation.

3.4.2 YCSB Results

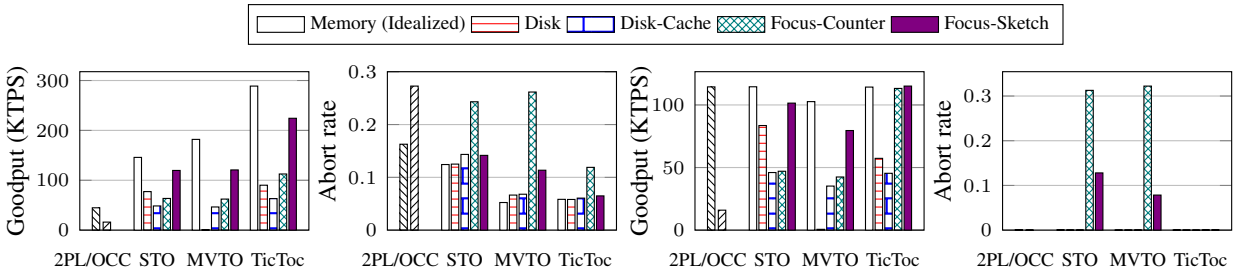
Small-Transaction Workloads

Figure 3.2 presents the goodput and abort rates for all timestamp-based CC mechanisms on four small-transaction YCSB workloads.

One important result from Figure 3.2 is that integrating FPSketch with the timestamp-based CCs enables them to substantially outperform their "Disk" and "Disk-Cache" counterparts. Except with MVTO, which is discussed below, Disk-Cache is slightly slower than Disk due to the overhead of managing the extra cache. A deeper dive on how cache size affects Disk-Cache's performance is provided 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 the workloads. The Focus-Sketch-based protocols also match or outperform their Focus-Counter-based variants on these workloads.

Several other conclusions can be drawn from these results:

- TicToc variants generally outperform not only 2PL/OCC but also their STO and MVTO analogues,



(a) High-contention (zipf=0.99) (b) Medium-contention (zipf=0.6)

Figure 3.3: 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-contention workload, several schemes have abort rates very close to 0.

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 2.4, 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-contention 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 in SplinterDB. In contrast, MVTO-Disk-Cache is able to utilize the same *V0* technique as the 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 3.3 presents the goodput and abort rates for timestamp-based CC mechanisms on 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

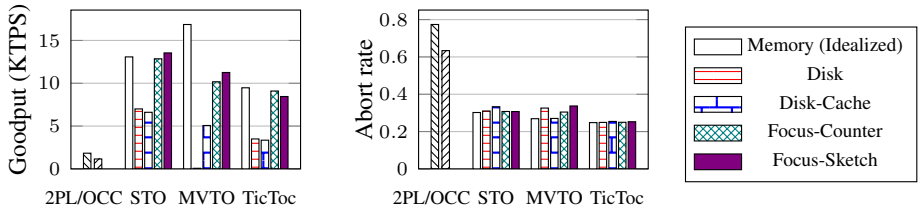


Figure 3.4: YCSB long-transaction workloads results with 120 processing threads. 5% of transactions perform 1000 reads and the rest of transactions perform 8 writes and 8 reads. The distribution is Zipfian with 0.9.

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 3.4 presents the 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 as well as their Memory variants, demonstrating that approximate timestamps can handle large transactions, not only small ones.

Approximate Timestamp Storage Versus Caching

Figure 3.5 shows how varying the SplinterDB cache size affects goodput of TicToc-Disk and TicToc-Focus-Sketch. The purpose of this experiment is to answer the question: can increasing cache sizes alone fix the performance problems in Disk-based schemes?

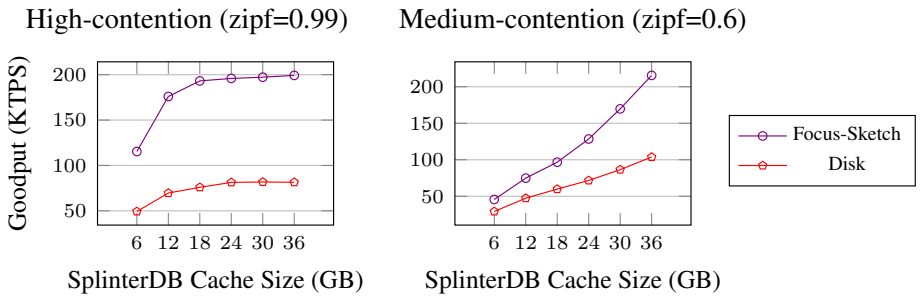


Figure 3.5: YCSB read-intensive small-transaction workloads for TicToc Disk and Focus-Sketch with varying SplinterDB Cache Size.

Figure 3.5 suggests that the answer is "No." In the high-contention workload, TicToc-Disk with a 37 GiB cache is still slower than TicToc-Focus-Sketch with a mere 6GiB cache. And in the medium-contention 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 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

This section compares 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 are 673M keys in the database, so the Memory variant stored 27GiB of metadata. The Disk variant also stores 16 bytes of metadata with each 124-byte record in the database. All metadata is stored in SplinterDB's 6GiB cache, so the total memory used for metadata is approximately 16 / (124 + 16) $\times$ 6 GiB $\approx$ 685 MiB. The Disk-Cache adds a small 32KiB cache to the Disk scheme and has similar memory usage. The FPSketch variants use a 32KiB cache plus a hashtable of 40-byte entries for

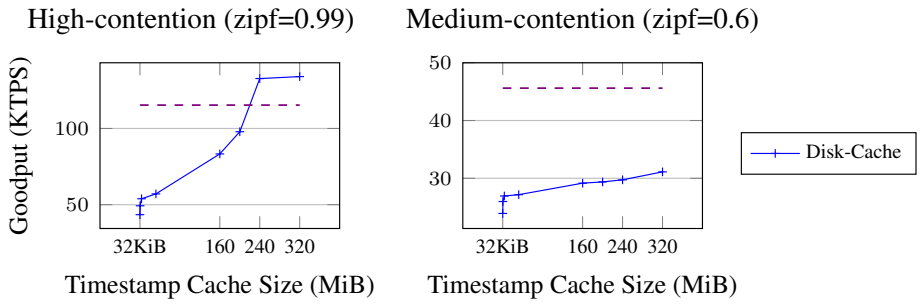


Figure 3.6: YCSB read-intensive small-transaction workloads for TicToc Disk-Cache compared to Focus-Sketch with 32KiB of sketch size (Dashed line).

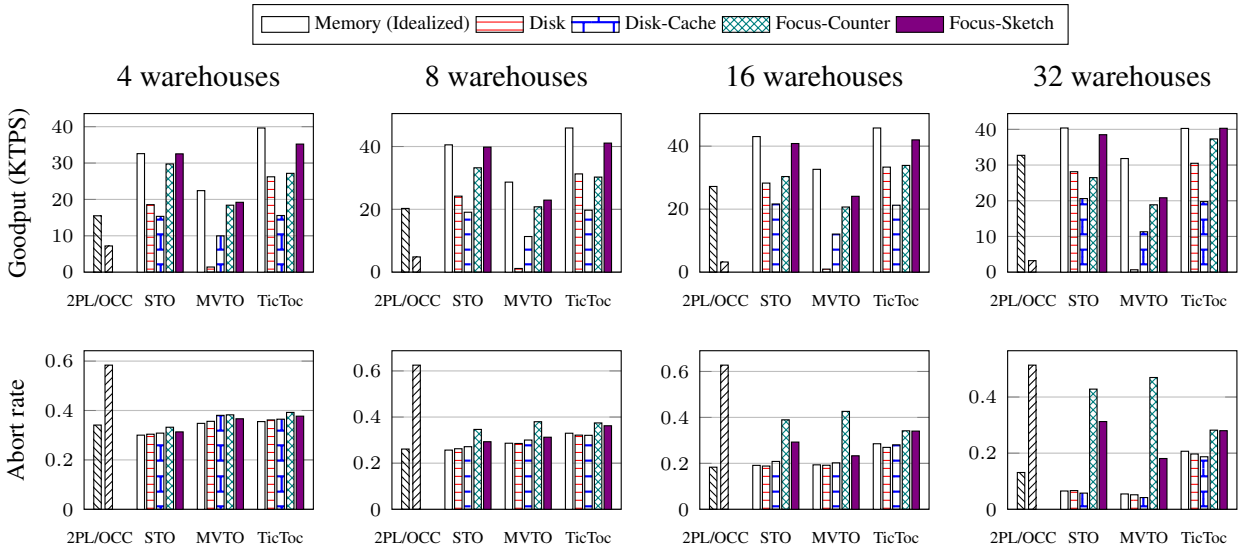


Figure 3.7: TPC-C results with 120 processing threads (More warehouses means less contention).

each active key, for a total average memory usage of 160KiB.

Disk-Cache Versus Focus-Sketch

Figure 3.6 compares the performance of Disk-Cache with Focus-Sketch. Focus-Sketch–32KiB cache uses the same configuration. The dashed line is the goodput of Focus-Sketch. In high-contention workloads, Disk-Cache requires about 220MiB 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 medium-contention workloads because the Disk-Cache variant still requires I/O to access timestamps, while Focus-Sketch does not.

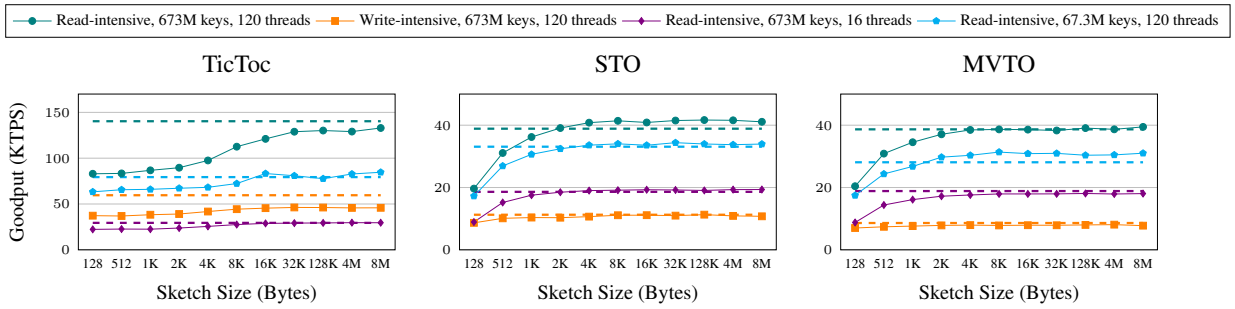


Figure 3.8: The goodput of TicToc, STO, and MVTO for high-contention YCSB workloads with varying sketch sizes. Workloads are configured by operations proportion (Read-intensive vs. Write-intensive), the database size (containing 673M vs. 67.3M keys), and the concurrency level (120 vs 16 threads). The dashed lines represent the goodputs of the Memory variant for each configuration.

3.4.3 TPC-C Results

Figure 3.7 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.

3.4.4 Sketch Size

A factor-analysis of the sketch size of FPSketch for TicToc, STO, and MVTO is presented. Figure 3.8 presents the goodput for the high-contention YCSB workloads as the sketch size increases. Workload type is varied (read-intensive vs. write-intensive), along with 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.

3.5 Summary

This chapter introduces *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 timestamp storage provides guarantees that are sufficient for preserving the correctness of the timestamp-based CC mechanisms studied here, while being extremely memory efficient. This chapter presents the design of an approximate timestamp storage system, FPSketch, and implements two of its variants, Focus-Counter and Focus-Sketch. The implementation integrates FPSketch with three timestamp-based CC protocols–STO, MVTO, and TicToc. Through this integration, FPSketch unlocks the ability to adapt existing CC techniques and paves the way for innovative CC mechanisms in future disk-based key-value stores.

The experimental evaluation on modern NVMe SSDs demonstrates that FPSketch enables timestamp-based CC mechanisms to achieve performance comparable to idealized in-memory configurations while requiring only minimal memory overhead. Across diverse workloads, Focus-Sketch variants consistently outperform Focus-Counter variants, and both significantly exceed disk-based timestamp storage approaches. Notably, TicToc with Focus-Sketch achieves up to 14× better performance than traditional 2PL and OCC approaches, and up to 5.9× improvement over disk-based timestamp storage. These findings establish FPSketch as a practical and effective solution for enabling modern timestamp-based concurrency control in disk-based key-value stores. While this evaluation focuses on NVMe SSDs, production environments span a wide spectrum of storage technologies. The next chapter extends this analysis by evaluating FPSketch across diverse storage media, from slow storage (SATA SSD and HDD) to fast storage (simulated CXL-based SSDs), to understand how storage characteristics fundamentally shape FPSketch's performance benefits and trade-offs.

Chapter 4

Evaluating FPSketch Across the Storage Spectrum

Storage technologies in production environments span several orders of magnitude in performance—from traditional hard disk drives (HDDs) with the order of millisecond latencies [43] to emerging byte-addressable storage like CXL-based SSDs approaching DRAM-like speeds [53]. Since FPSketch enables modern timestamp-based concurrency control for disk-based key-value stores, evaluating its effectiveness across this performance spectrum is critical for understanding both current deployment opportunities and future applicability as storage technologies continue to evolve.

This chapter presents a comprehensive evaluation across slow storage (SATA SSD and HDD) in Section 4.1 and fast storage (simulated CXL-based SSDs) in Section 4.2 to reveal how FPSketch's benefits and trade-offs evolve with storage characteristics. The analytical evaluation demonstrates that FPSketch remains effective across this wide range, though the nature of its advantages changes fundamentally as storage performance improves.

4.1 Slow Storage

Slow storage represents traditional disk-based systems where storage latency significantly impacts transaction processing performance. On slow storage, disk I/O becomes the dominant bottleneck, making the

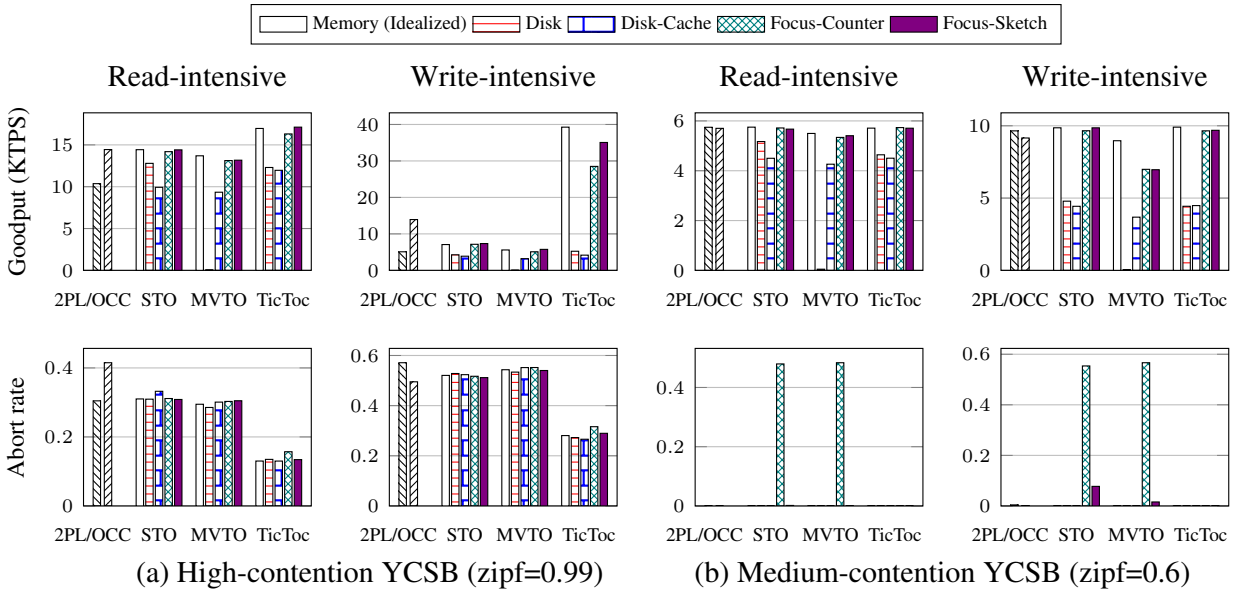
elimination of timestamp disk accesses provided by FPSketch particularly valuable. The results demonstrate how FPSketch performs under challenging storage conditions and reveal its effectiveness when storage characteristics fundamentally limit transaction throughput.

On slow storage media, the traditional CC methods, 2PL and KR-OCC, are the most relevant baselines because they do not depend on on-disk timestamp metadata; this avoids extra random I/O overhead on the critical path. Accordingly, FPSketch is primarily compared against 2PL and KR-OCC, and Disk/Disk-Cache timestamp variants are used as secondary references to quantify the penalty of persisting timestamps on slow devices.

The same workloads and evaluation methods as in Section 3.4 are used. For slow storage setups, both SATA SSD and HDD are used. The SATA SSD experiments are conducted on the same type of machine as the NVMe SSD tests, with only the storage hardware changed. However, for the HDD tests, different machines are used because CloudLab does not offer the same machines with HDDs. Using SATA SSD allows comparing results fairly with NVMe SSD and other faster storage. The HDD experiments demonstrate how FPSketch variants perform on a real HDD disk, helping understand its behavior in truly slow storage situations.

This section evaluates FPSketch on slow storage media, highlighting the following:

- FPSketch enables timestamp-based CC to outperform 2PL and KR-OCC in high-contention workloads. On SATA SSD, TicToc-Focus-Sketch is up to 6.89× and 2.52× faster than 2PL and KR-OCC, respectively; on HDD it achieves 1.8× the goodput of KR-OCC. Relative to Disk variants, FPSketch removes costly timestamp reads, reaching up to 569% (SATA) and 519% (HDD) higher goodput in write-intensive cases.
- Across mixed and long-transaction workloads on slow storage, FPSketch continues to exceed 2PL/KR-OCC. In mixed YCSB, it delivers up to 1.24× and 1.59× better goodput than 2PL and KR-OCC, respectively, while also improving up to 60% over Disk variants by eliminating timestamp I/O. Under medium contention on HDD, where I/O dominates, FPSketch still lifts goodput by up to 19% (read-intensive) and 121% (write-intensive) versus Disk.
- In TPC-C with 4 warehouses, STO-Focus-Sketch and TicToc-Focus-Sketch reach up to 1.25× and 1.18× better goodput than 2PL/KR-OCC on HDD, showing benefits across synthetic and realistic



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

Figure 4.1: YCSB small-transaction workloads results with 120 processing threads on slow SSD. 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-contention workload, several variants have abort rates very close to 0.

benchmarks.

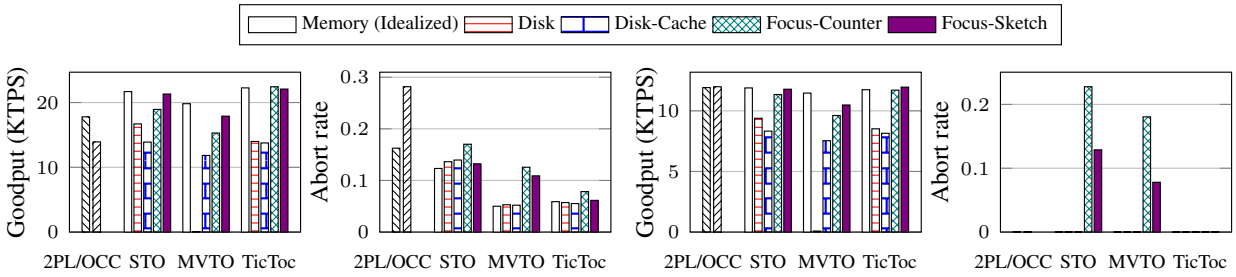
4.1.1 SATA SSD

YCSB Small-Transaction Workloads

Figure 4.1 presents the goodput of all timestamp-based concurrency control variants on four YCSB small-transaction workloads using slow SSD storage. The patterns in the results are similar to those shown in Section 3.4 for NVMe SSD. The FPSketch variants regularly achieve goodput very close to the Memory setup, which shows that they remain effective even when storage is slower.

In read-intensive workloads, the FPSketch variants (Focus-Counter and Focus-Sketch) outperform the slow-storage baselines: TicToc-Focus-Sketch is 1.65× and 1.19× faster than 2PL and KR-OCC, respectively, under high contention, while almost matching the Memory setup. They also improve goodput by as much as 39% compared to Disk variants when contention is high, and up to 23% in medium-contention situations.

For write-intensive workloads, the difference in goodput becomes even greater. The FPSketch variants



(a) High-contention (zipf=0.99) (b) Medium-contention (zipf=0.6)

Figure 4.2: YCSB mixed-transaction workloads results with 120 processing threads on slow SSD. 80% of transactions perform 2 reads and 2 writes, and 20% of transactions perform 28 reads. Note that, in the medium-contention workload, several variants have abort rates very close to 0.

again stay close to Memory and clearly exceed 2PL and KR-OCC: under high contention, TicToc-Focus-Sketch delivers 6.89× and 2.52× higher goodput than 2PL and KR-OCC, respectively. Relative to Disk variants, they boost goodput by up to 569% in high contention and 119% in medium contention.

TicToc is a timestamp-based CC protocol designed for high concurrency. Paired with FPSketch, it delivers the best performance across all workloads because FPSketch eliminates many on-disk timestamp lookups, reducing storage-induced stalls. The benefit is most pronounced in write-intensive scenarios: SplinterDB's fast write path allows operations to proceed quickly even with slow storage, but serializing transactions requires reading timestamps from disk, turning writes into reads. FPSketch effectively mitigates this overhead.

YCSB Mixed-Transaction Workloads

In Figure 4.2, both FPSketch variants show strong performance in mixed-transaction settings on slow SSD. In high-contention scenarios, FPSketch outperforms 2PL and KR-OCC by up to 1.24× and 1.59×, respectively. It also offers up to 60% higher goodput than the Disk variants by avoiding timestamp I/O.

Under medium contention, the goodputs of these systems are mainly limited by slow disk access. Unlike the evaluation with NVMe SSDs described in Section 3.4, the concurrency control method has little effect on overall goodput. However, since the Disk variants must fetch timestamps from the disk, there is extra overhead, resulting in lower goodput. FPSketch avoids this disk overhead and can improve goodput by up to 40%.

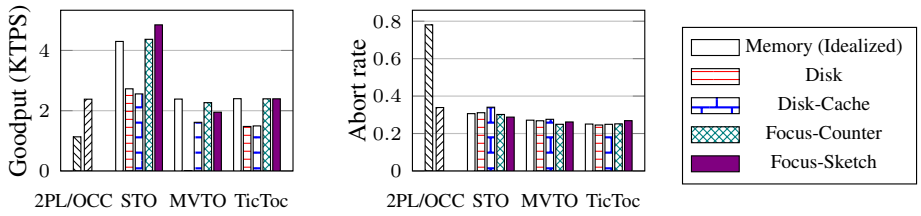


Figure 4.3: YCSB long-transaction workloads results with 120 processing threads on slow SSD. 5% of transactions perform 1000 reads and the rest of transactions perform 8 writes and 8 reads. The distribution is Zipfian with 0.9.

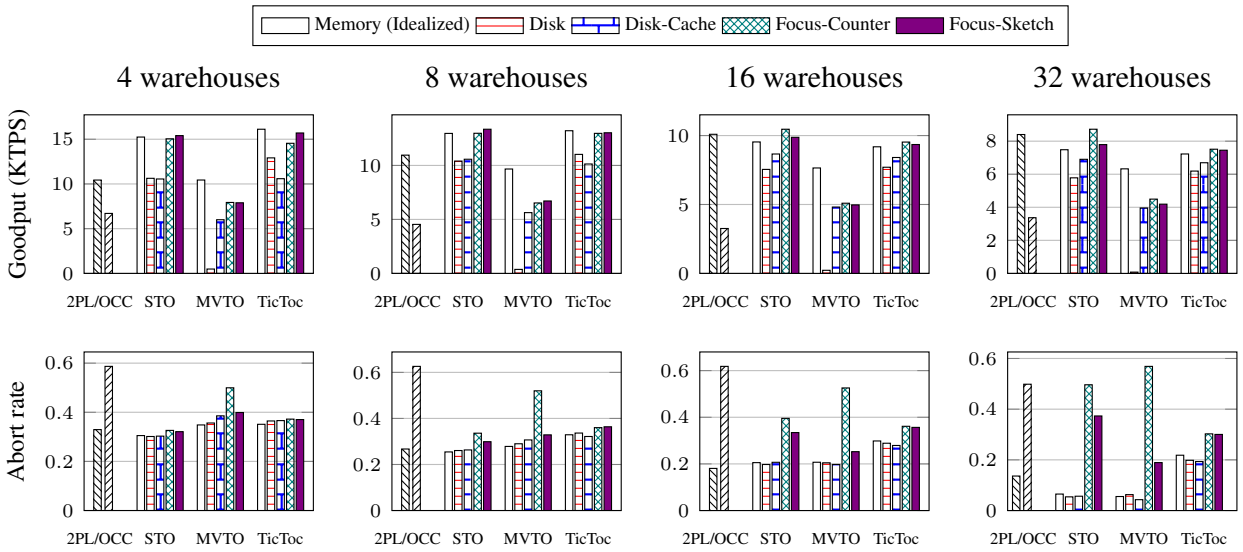


Figure 4.4: TPC-C results with 120 processing threads on slow SSD (More warehouses means less contention).

YCSB Long-Transaction Workloads

Figure 4.3 presents the goodput results for the long-transaction workload on slow SSD. The main pattern mirrors prior sections: the FPSketch variants maintain higher goodput than 2PL/KR-OCC and their Disk versions. Because the disk is slow, long transactions run more slowly and the differences between methods are smaller than with NVMe SSD (Figure 3.4). Even so, FPSketch continues to deliver the best performance among all tested protocols.

TPC-C Workloads

Figure 4.4 shows the goodput for the same TPC-C workloads described in Section 3.4.1. Compared to 2PL, FPSketch variants achieve better goodput in high-contention cases (4 and 8 warehouses) by avoiding on-disk timestamp metadata. At lower contention, 2PL can be competitive with TicToc-FPSketch on slow disks due to abort patterns: 2PL more often aborts shorter `Payment` transactions, whereas TicToc tends to abort longer `NewOrder` transactions. For example, in the 32-warehouse scenario, 93% of aborts in 2PL are `Payment`, while 79% in TicToc are `NewOrder`. Re-executing longer transactions harms TicToc goodput on slow disks, making TicToc-Focus-Sketch 0.89× as fast as 2PL at 32 warehouses, but 1.5× faster at 4 warehouses. Relative to Disk variants, FPSketch consistently removes timestamp I/O and improves goodput.

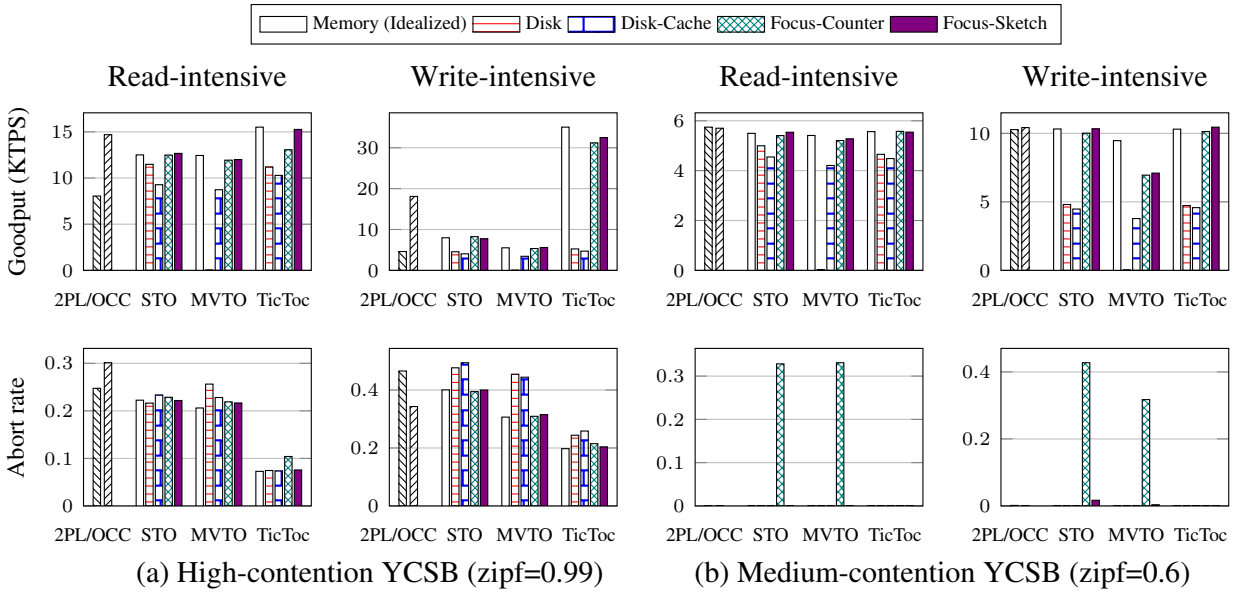
Having examined SATA SSD performance, the evaluation now turns to HDDs to evaluate FPSketch under even more challenging storage conditions. HDDs represent the slowest storage type in this evaluation, with latencies measured in milliseconds rather than microseconds. This allows observing how FPSketch's effectiveness scales as storage becomes increasingly I/O-bound and understanding its behavior in truly slow storage environments where disk access dominates all other factors.

4.1.2 HDD

Hard disk drives (HDDs) are still widely used today for storing very large amounts of data because they are inexpensive and can hold a lot of information. For these experiments, a different machine is used since CloudLab does not provide the same machine with an HDD. The system used for the HDD evaluation has a 36-core Intel(R) Xeon(R) Gold 6154 CPU running at 3.00GHz (with 2 threads per core), 192GiB of DDR4 2666 MHz memory, and an 893GiB SCSI HDD. In these tests, 60 threads are used because the version of SplinterDB used requires keeping some CPU cores available.

YCSB Small-Transaction Workloads

Figure 4.5 shows the results of running YCSB small-transaction workloads with 60 processing threads on an HDD. In the high-contention, read-intensive workload, KR-OCC reaches about 14 KTPS. Because disk I/O is a major bottleneck with slow storage, KR-OCC performs better than other timestamp-based concurrency control (CC) methods except for TicToc. TicToc-Focus-Sketch has the highest goodput in this



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

Figure 4.5: YCSB small-transaction workloads results with 60 processing threads on HDD. 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-contention workload, several variants have abort rates very close to 0.

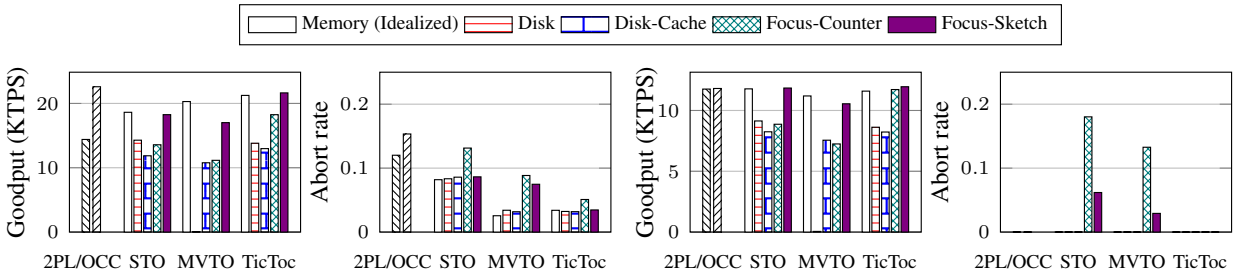
workload, running 1.04× faster than KR-OCC. In addition, Focus-Sketch increases TicToc's goodput by 36% compared to TicToc-Disk.

Similar to the results in Figure 4.1, when the workload is high-contention and write-intensive, FPSketch keeps TicToc's high performance. TicToc-Focus-Sketch achieves 1.8× the speed of KR-OCC and increases goodput by 519% compared to TicToc-Disk.

For workloads with medium contention, all methods are mainly limited by I/O. The Disk versions have extra I/O overhead to get timestamps. FPSketch removes this overhead, increasing goodput by up to 19% and 121% for the read-intensive and the write-intensive workloads, respectively.

YCSB Mixed-Transaction Workloads

Figure 4.6 shows the results of running YCSB mixed-transaction workloads with 60 processing threads on an HDD. The results are similar to the read-intensive workload results on HDD in Figure 4.5 because 20% of transactions perform 28 reads. TicToc-Focus-Sketch is 0.96× slower than KR-OCC but it increases goodput by 56.6% compared to TicToc-Disk.



(a) High-contention (zipf=0.99) (b) Medium-contention (zipf=0.6)

Figure 4.6: YCSB mixed-transaction workloads results with 60 processing threads on HDD. 80% of transactions perform 2 reads and 2 writes, and 20% of transactions perform 28 reads. Note that, in the medium-contention workload, several variants have abort rates very close to 0.

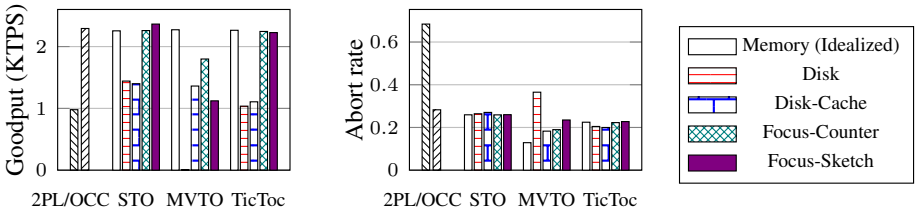


Figure 4.7: YCSB long-transaction workloads results with 60 processing threads on HDD. 5% of transactions perform 1000 reads and the rest of transactions perform 8 writes and 8 reads. The distribution is Zipfian with 0.9.

YCSB Long-Transaction Workload

As the slow disk is a major bottleneck in read-intensive workloads, all methods except 2PL show similar goodput, while 2PL causes higher abort rates. FPSketch variants show their effectiveness in removing the need to access timestamps on disk, resulting in higher goodput. However, MVTO with FPSketch incurs extra write operations as described in Section 3.3.3, resulting in lower goodput.

TPC-C Workloads

Figure 4.8 presents the goodput results for the TPC-C workloads described in Section 3.4.1, but now running on an HDD. As seen with the YCSB experiments on HDD, FPSketch performs very well in high-contention situations with a small number of warehouses. When contention is lower, however, the performance of all concurrency control methods becomes similar because the speed of the slow hard drive limits throughput. With 4 warehouses, STO-Focus-Sketch and TicToc-Focus-Sketch achieve up to 1.25× and 1.18× better

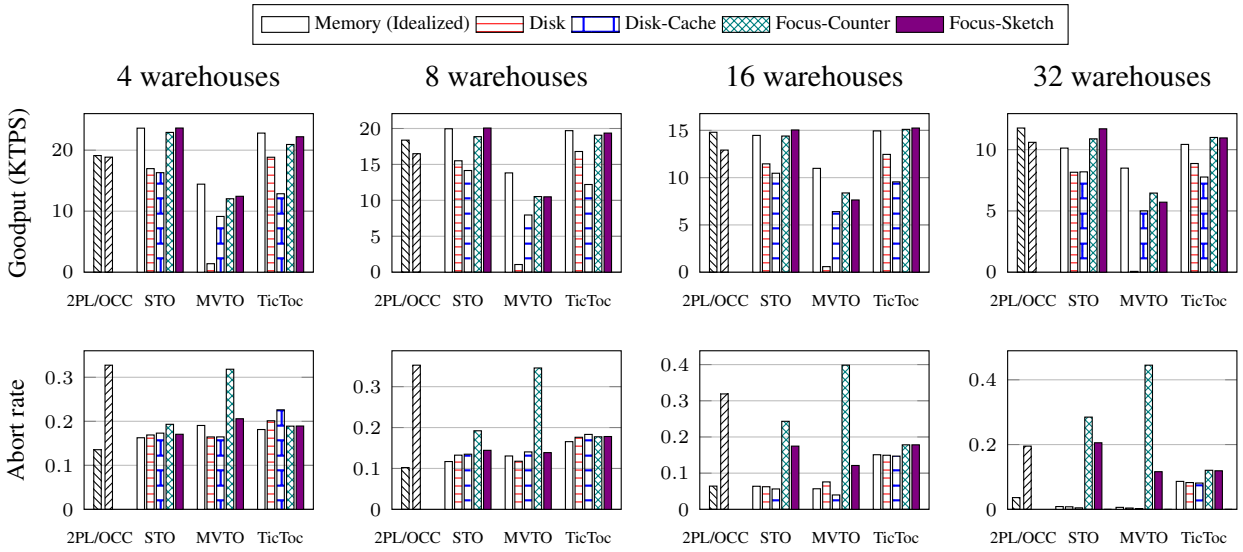


Figure 4.8: TPC-C results with 60 processing threads on HDD (More warehouses means less contention).

goodput than 2PL/KR-OCC, respectively.

4.2 Fast Storage

Having demonstrated FPSketch's effectiveness on slow storage, the evaluation now examines its performance on fast storage media, such as CXL-based SSDs with DRAM-like latency. The evaluation reveals a fundamental shift in performance characteristics: as storage becomes faster, the system transitions from being I/O-bound to CPU-bound, making the overhead of the FPSketch data structure itself more visible relative to storage access costs. These results provide insights into how FPSketch variants scale as storage technologies continue to evolve toward lower latencies.

This section presents FPSketch's performance on fast storage, revealing the following:

- On fast storage (simulated DRAM-like latency storage), FPSketch demonstrates dramatic performance advantages over traditional concurrency control across all workload types. In high-contention, read-intensive workloads, TicToc-Focus-Sketch achieves up to 3.55× and 21.5× higher goodput than 2PL and KR-OCC, respectively, with up to 286% improvement over TicToc-Disk. In write-intensive workloads, TicToc-Focus-Sketch is 2.12× and 3.16× faster than 2PL and KR-OCC, with 200% improvement over TicToc-Disk. For mixed-transaction workloads, TicToc-Focus-Sketch achieves 1.78×

and 24.3× higher throughput than 2PL and KR-OCC, with 77% improvement over TicToc-Disk. TPC-C results show consistent improvements: TicToc-Focus-Sketch achieves 34–45% higher goodput than TicToc-Disk across all contention levels, while STO-Focus-Sketch achieves up to 11.9–70.6× more goodput than KR-OCC (and up to 1.41× more than 2PL) with 107–150% improvement over STO-Disk, demonstrating that timestamp-based methods excel when storage latency is no longer the bottleneck.

- As storage becomes faster, FPSketch variants exhibit overhead from hash table operations, memory allocation, and sketch eviction that create a performance gap compared to the idealized Memory variant (39.5% and 22.5% lower for TicToc-Focus-Sketch and STO-Focus-Sketch, respectively). CPU cycles become the limiting factor rather than storage latency, revealing opportunities for future optimization, particularly for medium-contention workloads where overheads are more visible.

4.2.1 CXL Simulation Methodology

To evaluate FPSketch's effectiveness on next-generation storage technologies, CXL-based SSDs storage is simulated using block ramdisk [48]. This subsection details the simulation methodology, the assumptions made, and how these results relate to real CXL hardware.

Simulation Setup

Block ramdisk is used on the same CloudLab machine described in Section 3.4.1 to simulate CXL-based SSDs with DRAM-like latency. Block ramdisk provides storage access through the standard block device interface while storing data in RAM, effectively creating a software-based storage layer that mimics the low latency characteristics of memory-like storage technologies. This approach allows evaluating FPSketch's behavior when storage access times are dramatically reduced without requiring specialized CXL hardware that may not be readily available in experimental environments.

Assumptions and Limitations

The simulation makes several important assumptions that should be considered when interpreting results:

- **Interface semantics**: Block ramdisk uses the standard block device interface, whereas real CXL storage may support byte-addressable access through the CXL.mem protocol. This difference may affect the relative performance of different access patterns, though the fundamental latency characteristics the simulation seeks to capture are preserved.
- **Bandwidth characteristics**: Real CXL storage provides high bandwidth (up to 16 GB/s on PCIe 5.0 x4), whereas the simulation is limited by system RAM bandwidth. However, since the evaluation focuses on latency-bound operations (random I/O for timestamp access), bandwidth differences have minimal impact on the findings.
- **Remote memory effects**: Real CXL storage may exhibit NUMA-like behavior with remote memory access characteristics. The simulation does not capture these effects, though FPSketch's effectiveness in avoiding remote access overhead remains valid.
- **CPU overhead**: Block ramdisk still incurs some software overhead from the block device stack, whereas real CXL storage with direct memory access may have different overhead characteristics. However, this overhead is minimal compared to actual storage latency and does not materially affect the conclusions.

Experimental Configuration

In these fast storage experiments, only STO and TicToc protocols are evaluated. MVTO runs out of memory on the machine with 256GB of memory because its write operations keep creating new versions of keys, making it impractical for evaluation on fast storage where write rates are extremely high. 120 processing threads are run for all experiments, matching the configuration used in the NVMe SSD evaluations. All other experimental parameters (workloads, database size, cache configurations) remain consistent with the evaluations described in Section 3.4.1 to ensure fair comparison across storage types.

The following sections present results showing that FPSketch variants significantly outperform 2PL and KR-OCC on fast storage, achieving dramatic improvements in high-contention scenarios while maintaining effectiveness across all workload types. However, the relative overhead of FPSketch operations becomes more pronounced, revealing areas for future optimization.

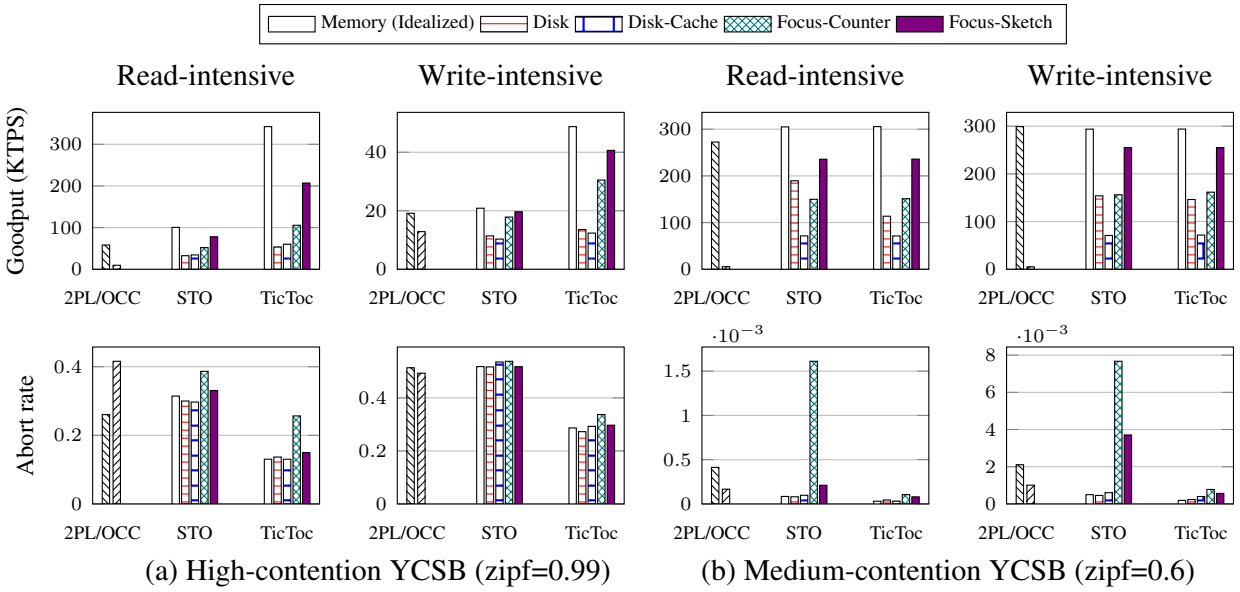


Figure 4.9: YCSB small-transaction workloads results with 120 processing threads on fast storage. In write-intensive workloads, each transaction performs 8 reads and 8 writes. In read-intensive workloads, each transaction performs 15 reads and 1 write. Note that, in the medium-contention workload, several variants have abort rates very close to 0.

YCSB Small-Transaction Workloads

Figure 4.9 presents the goodput results for all timestamp-based concurrency control (CC) variants across four small-transaction YCSB workloads when using fast storage.

The first key result is that timestamp-based CC methods work much better with fast storage than 2PL and KR-OCC. This improvement happens because timestamp-based methods allow more transactions to run at the same time (higher concurrency) and result in fewer aborts, whereas 2PL and KR-OCC start to experience CPU limitations when storage is no longer slow. KR-OCC is especially affected, since it needs more synchronization to check for conflicts when committing transactions.

The second main finding is that splitting timestamp storage using FPSketch is effective. Although Disk variants store timestamps on SplinterDB with fast storage, they are still slower than pure in-memory dedicated timestamp storage. This is because accessing timestamps through SplinterDB is more complex than accessing them from memory directly.

A third observation is that FPSketch variants do not reach the performance of the idealized Memory counterpart as storage becomes faster. This is due to the overhead from the FPSketch data structure itself.

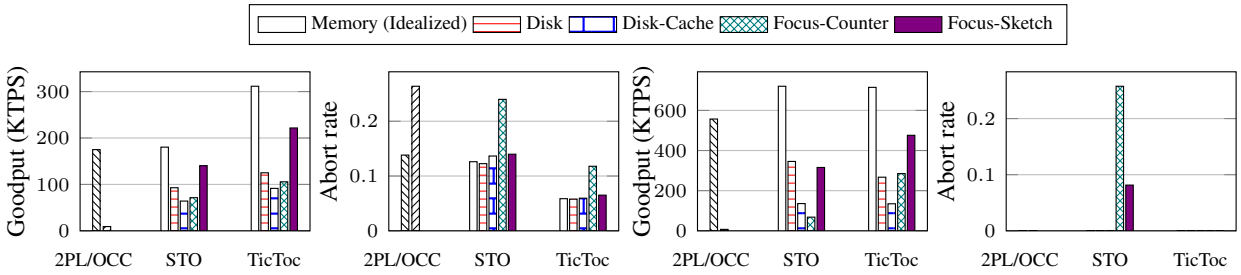
On slow storage, these overheads are negligible compared to disk I/O latency, but on fast storage they become significant. The overhead has several components: (1) Hash table operations: FPSketch must create and delete entries in a hash table, which requires allocating and copying memory for keys based on their reference count. In the high-contention, read-intensive workload for TicToc-Focus-Sketch, roughly 80% of `IncRef` calls are responsible for creating new entries in the hash table, which increases overhead. This dynamic memory allocation for hash table entries and key storage incurs CPU cycles and potential cache misses. (2) Sketch eviction: Focus-Counter has additional overhead because it needs to evict keys from the hash table to the sketch when reference counts reach zero, a step that does not occur in the Focus-Sketch variant. These overheads collectively create a performance gap between Memory and FPSketch variants that grows larger on fast storage, where CPU cycles become the limiting factor rather than storage latency.

In the high-contention, read-intensive workload, TicToc-Focus-Sketch is 3.55× and 21.5× faster than 2PL and KR-OCC, respectively. STO-Focus-Sketch is 1.39× faster than 2PL and 8.1× faster than KR-OCC. TicToc-Focus-Sketch increases goodput by 286% compared to TicToc-Disk. STO-Focus-Sketch increases goodput by 139% compared to STO-Disk. However, compared to the Memory variant, TicToc-Focus-Sketch has 39.5% lower goodput, and STO-Focus-Sketch is 22.5% lower.

For the write-intensive workload, similar trends appear but the differences are smaller. This is because the storage latency is less important for write operations in write-optimized data structures like LSM-trees. In the high-contention, write-intensive case, TicToc-Focus-Sketch is 2.12× and 3.16× faster than 2PL and KR-OCC, respectively. STO-Focus-Sketch is closer to 2PL and 1.53× faster than KR-OCC. TicToc-Focus-Sketch shows a 200% improvement over TicToc-Disk, and STO-Focus-Sketch improves by 72.2% over STO-Disk.

The Focus-Counter variant removes the I/O overhead from accessing timestamps on disk and so reaches higher goodput than 2PL and KR-OCC. However, it still has less goodput than Focus-Sketch variants because it needs to do extra work to evict keys and also has a higher abort rate.

In workloads with medium contention, these overheads play a clearer role. Since there is less contention, the protocol algorithm itself, rather than transaction retries, affects goodput more strongly. In these cases, 2PL slightly outperforms FPSketch variants because: (1) with reduced contention, lock conflicts become rare, minimizing 2PL's traditional weakness; (2) 2PL avoids the hash table operations and memory



(a) High-contention (zipf=0.99) (b) Medium-contention (zipf=0.6)

Figure 4.10: YCSB mixed-transaction workloads results with 120 processing threads on fast storage. 80% of transactions perform 2 reads and 2 writes, and 20% of transactions perform 28 reads. Note that, in the medium-contention workload, several variants have abort rates very close to 0.

allocation overheads inherent to FPSketch; and (3) the relative cost of lock acquisition and release becomes smaller compared to FPSketch's timestamp management operations when storage is fast. This result demonstrates that while FPSketch provides substantial benefits, there is room for optimization to reduce overhead, particularly for medium-contention workloads on fast storage.

YCSB Mixed-Transaction Workloads

Figure 4.10 presents the goodput outcomes for all timestamp-based CC mechanisms across two mixed-transaction YCSB workloads under fast storage conditions.

In the high-contention scenario, the trends of the different CC mechanisms are similar to those observed in the small-transaction workloads, but the performance gaps are even larger. This is because mixed-transaction workloads have a higher fraction of short transactions compared to the small-transaction workloads. Looking at the numbers, TicToc-Focus-Sketch achieves 1.78× and 24.3× higher throughput than 2PL and KR-OCC, respectively. STO-Focus-Sketch achieves 1.13× and 15.37× higher throughput than 2PL and KR-OCC, respectively. TicToc-Focus-Sketch also reaches 77% higher goodput than TicToc-Disk, and STO-Focus-Sketch is 51% faster than STO-Disk. However, when compared against the Memory configuration, both TicToc-Focus-Sketch and STO-Focus-Sketch have 28% and 22% lower goodput, respectively.

For the medium-contention scenario, there are two main points. First, similar to the small-transaction experiments, 2PL still outperforms the FPSketch variants, although the timestamp-based CC mechanisms running with Memory configuration are still able to outperform 2PL. This shows that timestamping itself is

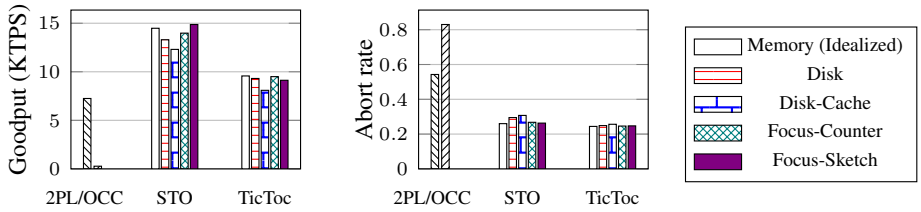


Figure 4.11: YCSB long-transaction workloads results with 120 processing threads on fast storage. 5% of transactions perform 1000 reads and the rest of transactions perform 8 writes and 8 reads. The distribution is Zipfian with 0.9.

a strong approach. Second, in this situation, a Disk-based configuration can outperform its FPSketch variant: STO-Disk achieves 1.09× faster than STO-Focus-Sketch, while TicToc-Focus-Sketch still outperforms TicToc-Disk. This is the result of differences in abort handling: STO aborts transactions immediately upon timestamp order break, while TicToc waits until commit time to make this decision. This leads to more over-approximation of timestamps in FPSketch for STO, especially since mixed-transaction workloads have more short transactions. With bigger sketch size than the evaluation setup, the abort rate of STO-Focus-Sketch decreases. For example, increasing the sketch size for STO-Focus-Sketch from the current 32KiB to 256KiB raises goodput to approximately 520 KTPS, improving goodput by 50.4% compared to STO-Disk. Thus, the more memory for sketch is necessary for STO if transactions are short and their operations are executed quickly.

YCSB Long-Transaction Workload

The results for long-transaction workloads in Figure 4.11 are similar to those on other types of storage. On fast storage, STO and TicToc with approximate timestamp storage reach goodput almost equal to their Memory versions. This shows that approximate timestamps work well for both large and small transactions on fast storage as well.

TPC-C Workloads

Figure 4.12 shows the TPC-C benchmark results on fast storage. The figure includes both goodput and abort rates for different concurrency control methods as the number of warehouses changes (which also changes the contention level). Similar to the YCSB results, both Focus-Sketch and Focus-Counter always

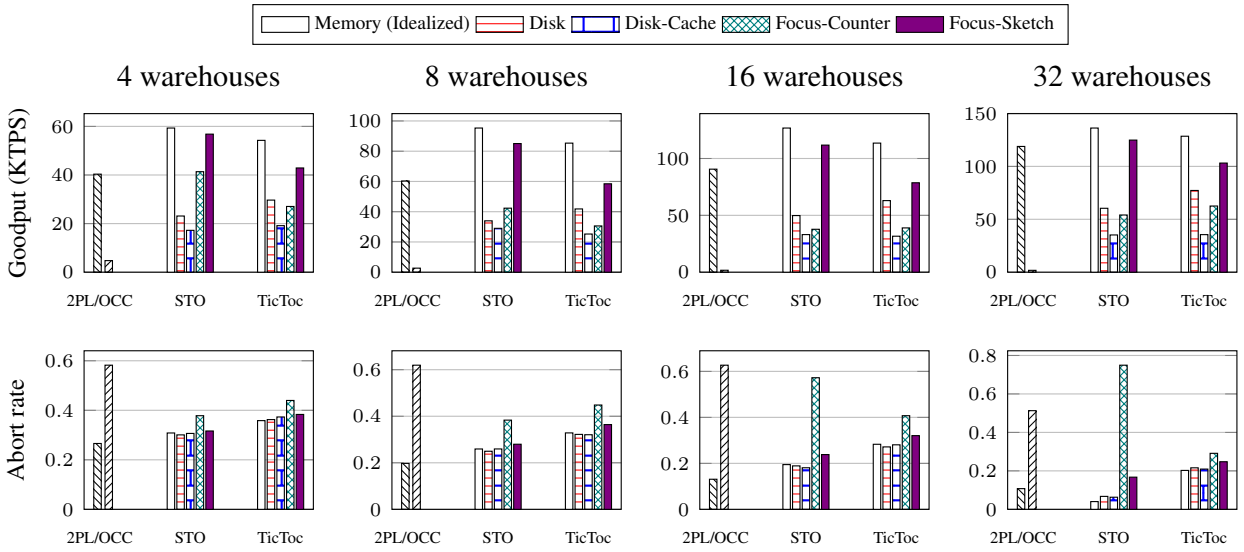


Figure 4.12: TPC-C results with 120 processing threads on fast storage (More warehouses means less contention).

have lower goodput than the Memory version because of extra overheads, but they are still much better than the Disk-based methods.

TicToc-Focus-Sketch achieves up to 9 to 58.3× higher goodput than KR-OCC (though it can be slower than 2PL at lower contention levels with 0.87–1.06× relative performance), and about 34–45% higher goodput than TicToc-Disk at all contention levels. STO-Focus-Sketch achieves up to 11.9 to 70.6× more goodput than KR-OCC and up to 1.41× more than 2PL, and about 107–150% improvement over STO-Disk in all cases. The TPC-C benchmark consists of both short and long transactions, and STO achieves higher goodput than TicToc for long transactions. This matches earlier findings from the YCSB long-transaction workload in Figure 4.11.

Comparing Focus-Sketch and Focus-Counter, Focus-Sketch always has better goodput at every contention level. This is because Focus-Sketch has no overhead from sketch evictions. This indicates that Focus-Sketch is a good option for deployment on fast storage.

4.3 Summary

This chapter presents a comprehensive evaluation of FPSketch across diverse storage media, from traditional slow storage (SATA SSD and HDD) to modern fast storage simulating CXL-based SSDs with DRAM-like

latency. The experimental study reveals several important findings and lessons that guide the adoption and future development of FPSketch.

4.3.1 Key Findings

The evaluation demonstrates that FPSketch remains effective across a wide range of storage technologies. On slow storage media, including SATA SSD and HDD, FPSketch enables timestamp-based protocols to outperform traditional concurrency control methods. In high-contention, write-intensive workloads on SATA SSD, TicToc-Focus-Sketch achieves up to 6.89× and 2.52× higher goodput than two-phase locking (2PL) and KR-OCC, respectively, while in read-intensive workloads it reaches 1.65× and 1.19× faster than 2PL and KR-OCC. On HDD, TicToc-Focus-Sketch achieves up to 1.8× the goodput of KR-OCC in write-intensive workloads, and in TPC-C with 4 warehouses, STO-Focus-Sketch and TicToc-Focus-Sketch reach up to 1.25× and 1.18× better goodput than 2PL/KR-OCC. Also, FPSketch continues to provide substantial improvements over traditional methods by eliminating the overhead of accessing timestamps from disk.

On fast storage media, the experiments reveal a different set of characteristics. Timestamp-based concurrency control methods, when combined with FPSketch, significantly outperform 2PL and KR-OCC because they enable higher concurrency and result in fewer transaction aborts. For instance, TicToc-Focus-Sketch achieves up to 3.55× and 21.5× higher goodput than 2PL and KR-OCC, respectively, in high-contention, read-intensive workloads. However, FPSketch variants do not reach the performance of the idealized Memory configuration due to overhead from their operations, memory allocation, and key management in the FPSketch data structure itself.

4.3.2 Lessons Learned

The experimental study yields several important lessons. First, the effectiveness of FPSketch varies with storage characteristics, and the appropriate baselines vary accordingly. On slow storage (I/O-bound), 2PL and KR-OCC are the right baselines because they avoid on-disk timestamp metadata; in this regime, FPSketch enables timestamp-based CC to exceed those baselines by eliminating timestamp I/O. On fast storage (CPU-bound), while FPSketch still outperforms Disk-based approaches and surpasses 2PL/KR-OCC by scaling concurrency, the relative overhead of the FPSketch data structure becomes more visible. This

suggests future optimizations should focus on reducing CPU overheads, particularly for high-performance storage environments.

Second, Focus-Sketch consistently outperforms Focus-Counter across all storage types and workloads. The additional overhead of evicting keys to the sketch in Focus-Counter creates a performance gap that becomes more pronounced as storage becomes faster. Moreover, Focus-Sketch also outperforms Disk-based configurations in the vast majority of scenarios, making it the clear preferred choice among all timestamp storage options. Only in rare cases with medium-contention scenarios on fast storage can Disk-based configurations occasionally match or slightly exceed Focus-Sketch performance, when the overhead of maintaining the sketch outweighs the benefits of avoiding disk accesses. In practice, these cases are limited, and Focus-Sketch offers superior performance while still eliminating the need to access timestamps from persistent storage.

Third, these results highlight the importance of transaction characteristics in determining FPSketch's effectiveness. Short transactions with quick operations benefit more from FPSketch, especially on slow storage. For long transactions, the impact is smaller because the transaction execution time dominates, but FPSketch still provides improvements by reducing timestamp access overhead. The mixed-transaction workloads demonstrate that FPSketch maintains its effectiveness even when workloads contain both short and long transactions.

Fourth, sketch capacity and adaptivity matter as storage becomes faster. Performance is sensitive to the size of the sketch, especially for STO on fast storage where immediate validation increases pressure on approximate timestamps. Larger sketches reduce over-approximation and aborts (e.g., increasing the sketch from 32KiB to 256KiB substantially improved STO-Focus-Sketch in mixed workloads). A simple autotuning policy that adjusts sketch size using online abort/eviction telemetry within a small memory budget can deliver robust performance across workloads.

Finally, CPU and NUMA considerations dominate on fast storage. Once the system is CPU-bound, allocator and hash table costs in FPSketch become critical. Using slab/arena allocation for entries and keys, pre-sizing and per-core/socket sharding of the hash table, reducing key copies, and NUMA-aware placement can close part of the gap to the Memory configuration.

4.3.3 Deployment Guidelines

Based on this comprehensive evaluation, the following deployment guidelines for FPSketch are offered:

Variant selection: Focus-Sketch should be preferred over Focus-Counter in virtually all scenarios, as it consistently outperforms Focus-Counter across all storage types and workloads while requiring similar memory footprint. The only exception is when memory constraints are extremely tight and the workload has very low contention, where Focus-Counter's ability to evict keys might provide marginal memory savings.

Storage-specific recommendations: On slow storage (SATA SSD and HDD), FPSketch provides dramatic improvements against traditional concurrency control methods and should be adopted whenever timestamp-based CC methods are used. The benefits are most pronounced for high-contention workloads. On fast storage (CXL-based SSDs or similar future innovations), FPSketch remains beneficial, particularly for high-contention scenarios where timestamp-based methods significantly outperform 2PL and KR-OCC. However, the overhead becomes more visible, suggesting future optimization opportunities.

Workload considerations: FPSketch is highly beneficial for the vast majority of practical workloads. Small transactions with high contention see the biggest wins; Long transactions also benefit, though proportionally less because transaction execution time dominates. Mixed workloads benefit substantially from FPSketch across all storage types.

Memory requirements: FPSketch requires minimal memory (32KiB in the experiments with an 80GB database), making it practical for deployment even in memory-constrained environments. The memory footprint remains consistent across storage types, as the sketch size is workload-dependent rather than storage-dependent.

Concurrency control pairing: TicToc with Focus-Sketch consistently provides the best performance across all evaluated scenarios, making it the recommended combination for new deployments. STO with Focus-Sketch is also excellent, particularly for long transactions. MVTO can benefit from FPSketch on slow storage, but may face memory limitations on fast storage with write-intensive workloads.

In conclusion, FPSketch proves to be a valuable technique for enabling modern timestamp-based concurrency control methods on persistent storage across a wide spectrum of storage technologies. While there are opportunities for further optimization, particularly for fast storage environments, the experimental evidence strongly supports the adoption of FPSketch in production systems that use various storage media.

Chapter 5

Related Work

5.1 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 3.1; 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 [55].

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.

5.2 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, 14, 20, 24, 33, 35, 45, 46]. For example, RocksDB supports both pessimistic and optimistic concur-

rency 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 [55]. 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.

5.3 Timestamp Access Acceleration

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 [27] and Umbra [22] highlight the critical role of efficient page cache management in boosting performance, as on-disk timestamps can be fetched quickly once loaded. By contrast, This 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. Transactional systems leveraging FPSketch can attain even stronger performance gains when paired with robust page cache management.

5.4 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 [28] 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 [51], 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.

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.

5.5 Approximation with Sketches

In networking, FlexSwitch [44] 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.

Chapter 6

Conclusion and Future Work

This dissertation addresses a fundamental challenge in modern database systems: enabling efficient timestamp-based concurrency control in disk-based transactional databases. Through both theoretical analysis and experimental evaluation, the research demonstrates that approximate timestamp storage enables high-performance transaction processing in disk-based databases across diverse storage media technologies.

It presents two primary contributions. First, it introduces FPSketch, a novel approximate timestamp storage system that enables efficient timestamp-based concurrency control in disk-based databases. FPSketch combines a hash table for exact timestamps of active keys with a sketch for approximate upper bounds of inactive keys, achieving the performance benefits of fully in-memory timestamp storage while requiring only minimal memory—as little as 32KiB for an 80GB database. Second, it presents a comprehensive analytical and experimental evaluation demonstrating that FPSketch remains effective across a wide spectrum of storage technologies, from traditional hard disk drives with millisecond latencies to emerging CXL-based storage approaching DRAM-like speeds.

The FPSketch System

The first contribution establishes FPSketch as a practical solution to the metadata storage problem in timestamp-based concurrency control. Its design is grounded in a key insight: for timestamp-based protocols like STO, MVTO, and TicToc, overapproximating timestamps does not violate correctness—it may cause harmless extra aborts but preserves serializability. This insight allows utilizing approximate data structures while

maintaining all correctness guarantees.

FPSketch's hybrid architecture, combining a foveated region (hash table) and peripheral region (sketch), ensures that active keys maintain exact timestamps throughout their transaction lifetime while inactive keys can be safely approximated. It formally proves that this approach satisfies the necessary properties for correctness with all three protocols studied. The evaluation demonstrates that FPSketch-based implementations achieve dramatic performance improvements: TicToc with Focus-Sketch improves goodput by up to 5.9× over disk-based timestamp storage, up to 14× over traditional 2PL, and reaches performance close to an idealized in-memory system.

Broad Applicability Across the Storage Spectrum

The second contribution establishes the universal applicability of the approximate timestamp storage approach. This comprehensive evaluation across HDDs, SATA SSDs, NVMe SSDs, and simulated CXL-based storage reveals that FPSketch's benefits scale with the fundamental gap between local memory and remote storage access. This finding ensures that FPSketch will remain valuable as storage technology continues to evolve.

On slow storage (HDDs and SATA SSDs), where disk I/O dominates performance, FPSketch beats traditional concurrency control methods like 2PL and KR-OCC by eliminating the overhead of accessing timestamps from disk. TicToc-Focus-Sketch is up to 6.89× and 2.52× faster than 2PL and KR-OCC. FPSketch shows the effectiveness in disk I/O reduction by improving goodput by as much as 569% (SATA SSD) and 519% (HDD) in write-intensive cases. Therefore, FPSketch enables high-performance timestamp-based concurrency control on slow storage.

On fast storage (simulated DRAM-like speed of CXL-based SSDs), the performance characteristics undergo a fundamental shift: the system transitions from being I/O-bound to CPU-bound as storage latencies approach the single-digit microsecond range. This transition makes the overhead of FPSketch's data structure operations (hash table management, memory allocation, and sketch operations) more visible relative to storage access costs. However, FPSketch continues to provide substantial benefits by eliminating timestamp access overhead through persistent storage, and more importantly, timestamp-based concurrency control methods combined with FPSketch significantly outperform traditional approaches. Specifically,

TicToc-Focus-Sketch achieves up to 3.55× and 21.5× higher goodput than 2PL and KR-OCC, respectively, in high-contention workloads on fast storage, demonstrating that timestamp-based methods enable higher concurrency and fewer aborts when storage is no longer the limiting factor. While FPSketch variants do not reach the performance of idealized in-memory timestamp storage due to their internal overhead, they still deliver dramatic improvements over disk-based approaches and traditional concurrency control methods across the entire storage spectrum.

6.1 Key Findings and Insights

This research yields several important findings that guide both current deployment and future development:

Performance Characteristics

Across all evaluated storage technologies and workloads, FPSketch consistently outperforms traditional concurrency control methods and disk-based timestamp storage. The Focus-Sketch variant achieves goodput close to the idealized Memory configuration while using only a tiny fraction of memory. TicToc with Focus-Sketch emerges as the clear performance leader, significantly outperforming other concurrency control methods across all scenarios.

The memory efficiency of FPSketch is remarkable: a 32KiB sketch suffices for an 80GB database, representing less than 0.00004% of the database size. This efficiency makes FPSketch practical even in memory-constrained environments where storing all timestamps in RAM would be infeasible.

Storage-Dependent Behavior

The evaluation reveals that the nature of FPSketch's advantages changes fundamentally as storage performance improves. On slow storage, traditional concurrency control methods like 2PL and KR-OCC are commonly employed because they do not depend on on-disk timestamp metadata; this avoids extra random I/O, which is a major bottleneck on slow storage. FPSketch outperforms the traditional CC methods by effectively eliminating I/O bottlenecks. On fast storage, while FPSketch still effectively eliminates the overhead of accessing timestamps from disk, the CPU overhead from FPSketch operations becomes more visible, creating optimization opportunities for future work.

Workload Sensitivity

FPSketch demonstrates consistent effectiveness across diverse workloads, from small transactions typical of OLTP systems to long transactions. High-contention, write-intensive workloads show the largest improvements, but even medium-contention scenarios benefit substantially. Mixed workloads containing both short and long transactions maintain FPSketch's effectiveness, demonstrating its robustness.

Protocol Compatibility

The analytical and experimental evaluations confirm that FPSketch correctly integrates with STO, MVTO, and TicToc without any algorithmic changes to these protocols. This plug-and-play compatibility makes adoption straightforward and suggests that FPSketch could similarly integrate with current and future timestamp-based concurrency control protocols.

6.2 Broader Implications

The success of FPSketch points to a broader principle: approximate metadata management can enable high-performance system designs that would otherwise be impractical. The key insight—that overapproximation preserves correctness for many concurrency control protocols—may find application beyond timestamp storage.

As storage technologies continue evolving toward faster, more memory-like interfaces, the distinction between memory and storage blurs. FPSketch demonstrates how application-specific caching strategies can bridge this gap, providing a template for managing other types of frequently accessed metadata in future systems.

The evaluation also highlights an important shift in database system design. Traditional systems optimized for disk I/O as the dominant bottleneck. Modern systems must optimize for CPU efficiency and memory hierarchy utilization. FPSketch exemplifies this new design paradigm by prioritizing CPU and memory efficiency while maintaining correctness guarantees.

6.3 Future Work

While FPSketch demonstrates strong performance across diverse storage technologies, several directions offer opportunities for further improvement and broader application.

Optimizations for Fast Storage

The evaluation reveals that on fast storage (CXL-based or similar), FPSketch incurs CPU overhead from sketch operations, memory allocation, and key management that prevents it from fully matching the idealized Memory configuration. Future work could explore several optimization strategies:

First, *lock-free and wait-free algorithms* could reduce synchronization overhead in the sketch and hash table operations. Current implementations use per-bucket locks which, while correct, may become bottlenecks on fast storage where operations complete in nanoseconds.

Second, *custom memory allocators* optimized for the access patterns of FPSketch could reduce allocation overhead. Fast storage environments are CPU-bound, making efficient memory management critical.

Third, *hardware acceleration* could leverage modern CPU features like SIMD instructions for bulk sketch operations or hardware transactional memory for lock-free updates. Exploring how emerging CPU architectures can accelerate FPSketch operations presents an interesting research direction.

Finally, *adaptive sketch sizing* could dynamically adjust sketch size based on workload characteristics, reducing overhead for low-contention scenarios while maintaining accuracy for high-contention workloads.

Evaluation on Real CXL-based SSDs

While the evaluation includes a simulation of CXL-class latencies, an important next step is to evaluate FPSketch on production CXL-connected devices. Real hardware introduces effects that are difficult to capture in simulation, including PCIe/CXL fabric contention, device firmware policies (e.g., interrupt moderation, internal queueing, and thermal throttling), host driver and I/O stack interactions, and tail-latency behaviors under bursty workloads. A systematic experimental campaign on commercially available CXL-based SSDs and memory expanders would (1) validate the analytical model and calibrate constants, (2) quantify head- and tail-latency distributions and CPU overheads at varying queue depths, NUMA placements, and PCIe topologies, and (3) surface optimization opportunities specific to CXL (e.g., larger submission queues,

batching, and polling). These results would strengthen external validity and refine guidance for deploying FPSketch on next-generation storage.

Integration with Existing Database Systems

While FPSketch is realized in SplinterDB, integration with other database systems would validate broader applicability. Integrating FPSketch with other database systems like LSM-tree based systems like RocksDB, B-tree systems like PostgreSQL, or other database architectures could reveal system-specific optimization opportunities.

Beyond Timestamps: General Approximate Metadata

The principle underlying FPSketch —that approximate metadata can preserve correctness for certain protocols—may apply beyond timestamps. Could similar techniques optimize storage of locks, version numbers, conflict detection metadata, or other concurrency control state?

Exploring the space of metadata that can be safely approximated could yield additional optimization opportunities. This direction requires identifying metadata properties that allow approximation without violating correctness.

Workload-Aware Optimization

The evaluation demonstrates that FPSketch's effectiveness varies with workload characteristics. Future work could develop *adaptive FPSketch* variants that automatically adjust their behavior based on observed workload patterns. For example, sketch size could adapt to contention levels, eviction policies could optimize for observed access patterns, or the hash table could resize based on active key counts.

Machine learning techniques could potentially optimize FPSketch parameters based on historical workload data, automatically tuning for best performance without manual intervention.

6.4 Concluding Remarks

This dissertation demonstrates that approximate timestamp storage enables efficient timestamp-based concurrency control in disk-based databases while requiring minimal memory. Through the design and evalu-

ation of FPSketch, the research establishes that this approach remains effective across a wide spectrum of storage technologies, from traditional hard drives to emerging memory-like storage.

The key contribution is not merely a new data structure, but a demonstration that approximate metadata management can unlock high-performance system designs that would otherwise be impractical. As storage technology continues evolving and the gap between memory and storage narrows, techniques like FPSketch that optimize metadata access will become increasingly important for achieving optimal database performance.

FPSketch represents a practical, deployable solution to a fundamental challenge in modern database systems. By requiring only 32KiB of memory for an 80GB database while achieving performance close to idealized in-memory systems, FPSketch makes advanced concurrency control protocols accessible to real-world database deployments. The universal applicability of FPSketch across storage technologies ensures its relevance both today and as storage continues evolving.

More broadly, the dissertation contributes to a paradigm shift in transactional system design, from optimizing for disk I/O to optimizing for CPU efficiency and memory hierarchy utilization. As this shift continues, approximate metadata management techniques will play an increasingly central role in high-performance transaction processing in disk-based databases.

Bibliography

[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*.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64.

[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:*.

[4] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*, volume 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):128–141.

[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*, SODA '03, page 546–554, USA. Society for Industrial and Applied Mathematics.

[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, pages 209–223. USENIX Association.

[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*, SIGMOD '22, page 19–33, New York, NY, USA. Association for Computing Machinery.

[9] Howard Chu. 2015. LMDB: The Lightning Memory-Mapped Database Manager.

[10] Compute Express Link Consortium. 2025. Compute express link. `https://computeexpresslink.org/`.

[11] 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).

[12] 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)*, pages 49–63.

[13] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75.

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

[15] Transaction Processing Performance Council. 2010. Tpc benchmark c standard specification. Technical Report TPC-C Rev. 5.11.0, Transaction Processing Performance Council.

[16] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An introduction to the compute express link (cxl) interconnect. *ACM Computing Surveys*, 56(11):1–37.

[17] 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)*, pages 1–14, Renton, WA. USENIX Association.

[18] 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)*, pages 343–360, Boston, MA. USENIX Association.

[19] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633.

[20] Facebook, Inc. 2013. Rocksdb. `https://github.com/facebook/rocksdb`.

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

[22] 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):2797–2810.

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

[24] Google, Inc. 2019. Leveldb. `https://github.com/google/leveldb`.

[25] 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*, pages 658–670.

[26] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317.

[27] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-tree: A modern read-write-optimized concurrent larger-than-memory range index. *Proc. VLDB Endow.*, 17(11):3442–3455.

[28] Cary Huang. 2024. A deeper look inside postgresql visibility check mechanism.

[29] Myoungsoo Jung. 2022. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51.

[30] 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*, HPDC '24, page 186–199, New York, NY, USA. Association for Computing Machinery.

[31] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226.

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

[33] Pranab Mazumdar, Sourabh Agarwal, Amit Banerjee, Pranab Mazumdar, Sourabh Agarwal, and Amit Banerjee. 2016. Azure sql database. *Pro SQL Server on Microsoft Azure*, pages 129–156.

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

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

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385.

[37] 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*.

[38] PostgreSQL. 2025. PostgreSQL Source Code: src/backend/access/transam/slru.c. `https://github.com/postgres/postgres/blob/master/src/backend/access/transam/slru.c`.

[39] 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*, PPoPP '23, page 39–51, New York, NY, USA. Association for Computing Machinery.

[40] Redis. 2022. Redis. `https://redis.io`.

[41] David P. Reed. 1983. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23.

[42] 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*, pages 1583–1598. ACM.

[43] Samsung. 2013. Samsung solid state drive. White paper, Samsung Electronics Co., Ltd.

[44] 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)*, pages 67–82, Boston, MA. USENIX Association.

[45] 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*.

[46] 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*, page 491–518. Association for Computing Machinery and Morgan & Claypool.

[47] 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*, pages 809–822. USENIX Association.

[48] The Linux Kernel Developers. 2020. Using the RAM disk block device with Linux. https://www.kernel.org/doc/html/latest/admin-guide/blockdev/ramdisk.html.

[49] 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/.

[50] 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*.

[51] 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*, SIGMOD '17, page 1041–1052, New York, NY, USA. Association for Computing Machinery.

[52] 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*, pages 82–93. Springer.

[53] Shao-Peng Yang, Kyung-In Kim, Eujin Lee, Tae-Gun Song, Byung-Chul Tak, Venkatanathan Varadarajan, Ajay S. Shrivastava, Sung-Soon Lee, Chan-Ik Choi, Hak-Joo Oh, Hyunsun Cho, Seung-Wook Kim, Sung-Jae Lee, and Jung-Hoon Kim. 2023. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 599–613. USENIX Association.

[54] 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).

[55] 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*, pages 1629–1642.

[56] Yuhong Zhong, Daniel S Berger, Pantea Zardoshti, Enrique Saurez, Jacob Nelson, Dan RK Ports, Antonis Psistakis, Joshua Fried, and Asaf Cidon. 2025. Oasis: Pooling pcie devices over cxl to boost utilization. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 101–119.