From FASTER to F2: Evolving Concurrent Key-Value Store Designs for Large Skewed Workloads

Konstantinos Kanellis*
University of Wisconsin-Madison
Madison, WI
kkanellis@cs.wisc.edu

Badrish Chandramouli
Microsoft Research
Redmond, WA
badrishc@microsoft.com

Ted Hart
Microsoft Research
Redmond, WA
tedhar@microsoft.com

Shivaram Venkataraman
University of Wisconsin-Madison
Madison, WI
shivaram@cs.wisc.edu

ABSTRACT

Modern large-scale services such as search engines, messaging platforms, and serverless functions, rely on key-value (KV) stores to maintain high performance at scale. When such services are deployed in constrained memory environments, they present challenging requirements: point operations requiring high throughput, working sets *much larger* than main memory, and natural *skew* in key access patterns. Traditional KV stores, based on LSM- and B-Trees, have been widely used to handle such use cases, but they often suffer from suboptimal use of modern hardware resources. The FASTER project, developed as a high-performance open-source KV storage library, has demonstrated remarkable success in both in-memory and hybrid storage environments. However, when tasked with serving large skewed workloads, it faced challenges, including high indexing and compactions overheads, and inefficient management of non-overlapping read-hot and write-hot working sets.

In this paper, we introduce F2 (for FASTER v2), an evolution of FASTER designed to meet the requirements of large skewed workloads common in industry applications. F2 adopts a two-tier record-oriented design to handle larger-than-memory skewed workloads, along with new concurrent latch-free mechanisms and components to maximize performance on modern hardware. To realize this design, F2 tackles key challenges and introduces several innovations, including new latch-free algorithms for multi-threaded log compaction, a two-level hash index to reduce indexing overhead for cold records, and a read-cache for serving read-hot records. Our evaluation shows that F2 achieves 2-11.9× better throughput compared to existing KV stores, effectively serving the target workload. F2 is open-source and available as part of the FASTER project.

PVLDB Reference Format:
Konstantinos Kanellis, Badrish Chandramouli, Ted Hart, and Shivaram Venkataraman. From FASTER to F2: Evolving Concurrent Key-Value Store Designs for Large Skewed Workloads. PVLDB, 18(12): 4910 - 4923, 2025.
doi:10.14778/3750601.3750615

*Work started during internship at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750615

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

1 INTRODUCTION

Modern large-scale services (search [13], messaging [9], serverless functions [5]) are heavy users of memory and storage. Real-world applications require caches and key-value (KV) stores that offer extremely high throughput at low latencies. Moreover, there is a strong need to deploy such systems in constrained memory environments to reduce the costs of large-scale online services [15, 51]. Many of these services focus on point reads, point updates, and atomic read-modify-writes as their target storage operations [11].

Around 2017, the FASTER project [11, 12] was started with the goal of addressing such use cases. FASTER is a key-value storage library that focuses on the problem of achieving bare-metal thread-scalable performance. It was built in C# along with a port to C++. Briefly, FASTER employs a thread-scalable hash index on top of a hybrid log: a record-oriented storage tier that spans main memory and secondary storage. FASTER was shown to saturate memory bandwidth for in-memory workloads, achieving up to 160M random read operations per second on a single machine. Further, it was shown to saturate disk IOPS for disk-oriented workloads, achieving up to 1M random read operations per second [11]. The project was open-sourced in 2018 and has, over the years, seen incredible traction in the research and open-source community, as well as usage in real-world industry scenarios. FASTER has around 1 million downloads on NuGet [40], 6.5k stars on GitHub, and over 570 forks.

We highlight two representative scenarios where the FASTER library was integrated into real applications. First, we built a new platform for serverless functions called Netherite [6, 7], which is a runtime for Microsoft's Azure Durable Functions: a service that allows for the deployment of large-scale stateful serverless applications [8]. In Netherite, FASTER is used to efficiently store, retrieve, and update the state of individual function invocations. This state is stored across main memory and Azure storage, and the active state is brought back to memory on demand. Second, we integrated FASTER into a streaming service for the purpose of saving and retrieving state related to large event records in long-running streaming computations such as temporal joins. Motivated by these use cases, we sought new application scenarios to both reduce existing costs and optimize the FASTER design further.

Based on our survey of a variety of services that depend on point-based storage access, we make several key workload observations:

- Indexing extremely large state with (1) limited memory and (2) the availability of multiple storage tiers (such as SSD, hard disks, and replicated cloud storage) is a common scenario [15, 24, 51].
- A natural skew in key access patterns exists for both reads and writes operations (e.g., Zipfian), and the working sets do not entirely fit in-memory (i.e., larger-than-memory) [9, 24, 38].
- The *read*-hot and *write*-hot working sets may not fully overlap, necessitating separate treatment for each working set [9, 48].
- Disk wearing due to excessive writing is a practical concern, due to the large amount of data (i.e., TBs) that is being processed [39].

These characteristics apply to use cases such as behavior targeted advertising [2, 10] in search engines (covered in more detail in Section 2), where most tracked users of a search engine—such as those who performed searches in the last 7 days—are inactive at any given moment. They also hold for use cases such as serverless functions, where the state of most serverless functions are cold and unused, but we want high performance for the active functions. In streaming systems, we may be tracking billions of records in a temporal join synopsis, but only a small fraction of records may be active (i.e., being joined to new streaming events) at a given time.

The industry's go-to storage solution for applications that access large skewed workloads has traditionally been Log Structured Merge (LSM) tree-based systems, such as RocksDB [9], which emphasize the judicious use of memory. This is achieved through a tiered architecture, in which small in-memory components absorb user updates (i.e., memtable) and maintain index metadata (e.g., filters), while large disk components store the actual data (i.e., LSM levels) [36]. Although this design enables LSM-based systems to store TBs of data, it comes at a high performance cost. In particular, LSM-based systems deliver main-memory performance far below what modern hardware can achieve [1, 11], while their disk-oriented performance is poor, due to their inability to fully utilize available bandwidth of NVMe disk devices (i.e., just 35% NVMe SSD utilization, Section 2.1). Other storage solutions such as B-tree based systems [30–32, 47] also do not adequately meet the requirements of these applications, mostly due to their page-oriented design and large write-amplification (i.e., 25-90×, Section 2.1).

Given the above, a natural question arose: *could we use FASTER to improve the throughput of workloads that exhibit the above characteristics?* When we tried to apply the original, unmodified FASTER design [11], we encountered several practical challenges:

- When memory resources are limited, background compaction (or garbage collection) in FASTER's single-log design causes increased disk writes (as live records are migrated to the tail), resulting in high disk wear or disruptions in user request processing. It also incurs transient memory spikes as candidate live records need to be tracked in memory during the process.
- FASTER's hash index tracks all live keys in the store and imposes a fixed 8 bytes per-key memory indexing overhead. For billions of keys, this leads to a prohibitively large memory footprint.
- FASTER effectively keeps write-hot records in memory. However, when read-hot and write-hot working sets are non-overlapping, read-hot records are either served from disk or brought into the log to be later flushed to disk, incurring additional I/O operations.

In this paper, we describe how we have evolved the original FASTER C++ design to a new compartmentalized architecture that aims to address all these challenges. The resulting system, F2 (for FASTER v2), adopts a two-tier log architecture that inherently handles skewed workloads with greater memory-efficiency, and couples it with high-performance latch-free mechanisms and components that are necessary to saturate the disk bandwidth of modern NVMe storage devices.

However, realizing this design in practice required overcoming key technical challenges. For instance, ensuring that compacting records across tiers in a CPU- and memory-efficient fashion is critical, while performing compaction concurrently to other user operations (like atomic RMWs) in a safe manner is non-trivial. To this end, we introduce a *lookup*-based record compaction method that achieves minimal memory and disk I/O overhead, enabling F2 to handle billion-key scale workloads (Section 5). This compaction method is based on our new *Conditional-Insert* primitive, which prevents (older) compacted records from overwriting newer versions of the same record (i.e., lost-updates), ensuring overall system correctness, and is multi-threaded, achieving much faster compaction times. To minimize indexing overhead for cold records, we introduce a concurrent *two-level index* design that spans both memory and disk (Section 6). Finally, we augment a dedicated *read-cache* that provides immediate access to disk-resident read-hot records without any additional I/O overhead (Section 7).

This results in a system that (1) maintains low memory overhead and (2) achieves high performance by exploiting both multi-core CPUs and NVMe storage devices, and (3) remains disk-friendly with minimal write amplification. We believe F2 is the first comprehensive KV store design to address the practical challenges that limit hash-based storage systems from handling real-world skewed larger-than-memory workloads.

We experimentally evaluate F2 on YCSB and real-world Mix-Graph [9] workloads against several modern key-value stores. We show that F2 achieves 2-11.9× better throughput compared to existing state-of-the-art systems (e.g., original FASTER, RocksDB, SplinterDB, KVell, LeanStore), when memory resources are limited. We also show that F2 matches or outperforms existing solutions even when handling less-skewed workloads, e.g., when 90% of operations access 33% of keys.

F2 is written in C++ as an evolution of the FASTER C++ codebase. It is now available in open-source as part of the FASTER project.[1].

2 ISSUES WITH LARGE SKEWED WORKLOADS

We start by describing a representative scenario that we identified through discussions with real-world platform builders who use key-value stores, and were considering solutions such as FASTER.

(Targeted Advertising) *Search engines perform* behavior targeted advertising *[2, 10], for which they store and track per-ad clicks and impressions as well as per-user sketch of ad activity, in a KV store. Queries, to retrieve the sketch for a given user or the number of clicks on a given ad, are point based. Updates are either blind inserts (e.g., insert a new ad into the system) or read-modify-writes (e.g., update the counter or sketch for a given ad or user). The number of users and ads being actively served may be large, with the aggregate data greater*

[1] https://github.com/microsoft/FASTER/tree/main/cc

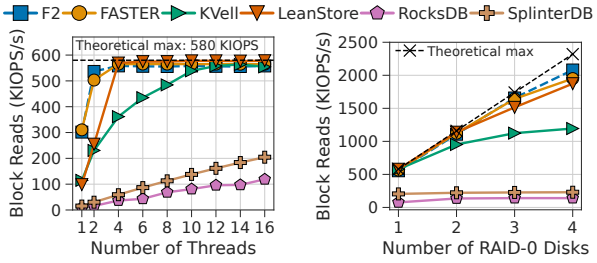

Figure 1: Out-of-memory performance for uniform read-only workload (30GiB database, 1GiB in-memory buffer), when (a) increasing the number of threads (left), and (b) increase the number of NVMe SSD disks in RAID-0 formation (right).

than the amount of memory available. For example, users who have interacted with the search engine over a 7-day period may constitute the entire set of tracked users. Further, a long tail of users and ads that are not actively being read or updated, still need to be available for immediate queries and updates. Both the read and write sets exhibit skewness. Finally, the set of users actively browsing and needing a lookup of their sketches (i.e., the read-hot keys) may be different from the ads being frequently shown and having their sketches updated (i.e., the write-hot keys).

This application scenario—and the ones described earlier such as serverless functions and streaming—exhibit several interesting workload characteristics. First, point operations and high throughput are still of paramount importance, and the working sets are *much larger* than main memory [23, 38]. Second, the total indexed data is often an order of magnitude larger than memory, with a large fraction of data being *rarely* updated or accessed [9, 48]. Third, memory is a *scarce* resource [15, 22, 33, 35, 51], periodic memory spikes are not acceptable (as services would have to provision for peak memory), and disk wearing due to excessive writes over the long term is a practical concern [39]. Finally, there is a natural skew in key access patterns for both reads and writes [3, 48], but the read and write working sets do not necessarily overlap.

2.1 Limitations of Existing KV Store Designs

We observe that existing KV stores, including LSM-based designs and B-tree based ones, do not fully address the requirements of the aforementioned workloads. We discuss their limitations below.

LSM-Tree-based Designs. Log-structured Merge (LSM) Trees [41] designs prioritize memory-efficiency, and they can store TBs of data through their tiered design. However, they fail to fully utilize the available NVMe SSD bandwidth [27]. To empirically show this, we perform a case study on a system equipped with a 16-core Intel Xeon CPU and four Samsung NVMe PM9A3 SSDs (detailed setup in Section 8.1), with a larger-than-memory (i.e., 30GiB with 1GiB in-memory buffer) uniform random read-only workload. As shown in Figure 1, we measure random read IOPS (4KiB blocks) for five popular key-values stores, when (a) increasing the number of threads used (left plot), and (b) when increasing the number of NVMe SSDs in RAID-0 configuration (right plot). We observe that both LSM-tree stores, RocksDB and SplinterDB, even when optimized and tuned properly, are not able to saturate I/O bandwidth.

One major factor is the widespread use of filters: given that each point lookup typically accesses numerous such filters, LSMs waste precious CPU cycles that could be used to issue more I/O [15]. Further, the benefits of filters can be diminished if the filters no longer fit in memory [15]. Prior work has found similar results: they [27] have shown that despite existing efforts, LSM stores fail to saturate SSD bandwidth and scale poorly [11] with increasing number of threads due to write-stalls caused by inefficient data flow across their components [50].

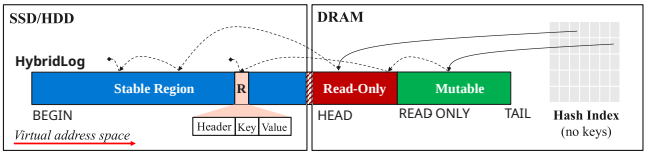

Figure 2: FASTER HybridLog and index architecture.

B-Tree-based Designs. B-Tree based storage designs [30–32] usually rely on in-memory structures (such as index pages, buffer pools, mapping table) to index the keys and cache the respective working set. Due to their CPU-optimized designs, they can achieve (i) linear thread scalability and (ii) disk IOPS saturation, even for modern NVMe storage devices.

KVell [31] employs a shared-nothing approach, where each thread uses a B-tree index to map keys to a page offset on disk. When memory resources are abundant, KVell saturates most of the available I/O bandwidth (Figure 1). However, when available memory is limited, parts of KVell's index are continuously paged out to disk [16], leading to high read/write-amplification (i.e., 25–90×) and low throughput (i.e., 3–10× drop), as shown in our evaluation (Table 1). LeanStore [1, 30] employs a B-tree indexing structure alongside an in-memory buffer manager to support larger-than-memory workloads. However, due to its page-oriented design, LeanStore performance degrades for skewed workloads. Specifically, given that hot records can be scattered around all pages, effectively caching the hot set in-memory is not possible. Perhaps more importantly, we find that the page-oriented design incurs high write-amplification (i.e., 30-65×), as for each user update, a disk block write is necessary (Table 1). Bw-Tree [32] uses log-structured writes using delta records for pages, and as a result incurs similar compaction overheads as LSM-Tree-based designs for pages with write-hot records.

3 ORIGINAL FASTER AND LIMITATIONS

We now provide background on the original FASTER design, including its components and internal mechanisms, before discussing its limitations in handling the above class of applications, motivating the need for evolving the design.

3.1 Design Overview

FASTER [11] is a log-structured, latch-free key-value store that targets point operations. As shown in Figure 2, it employs a hash index that chains records stored in a log, which spans both memory and disk (HybridLog). FASTER uses a lightweight epoch-based protection framework to facilitate cooperation across threads. Due to its log-oriented design, a garbage collection process is invoked periodically to shrink (compact) the log by removing stale tuples. This is performed using a *scan*-based compaction process that copies *live* records from the beginning of the HybridLog to its tail. As long as both the working record set and the index fit in memory, FASTER achieves high performance. Below, we detail its components.

Hash Index. At its core, FASTER consists of a latch-free in-memory hash table, which is divided into cacheline-sized buckets. Each bucket entry contains a pointer to a record whose key hashes to that bucket. Each record points to another record, forming a logical linked list of records with common significant key hash bits (i.e., *hash chain*). Each bucket entry contains additional bits from the associated records' key hash, increasing hashing resolution and further disambiguating what records the bucket entry points without full key comparisons. A bucket occupies 8 bytes of in-memory space. FASTER defines four user operations: `Read`, `Upsert`, `RMW`, and `Delete`. Latch-free algorithms are used to add/remove entries in the index and to add records at the hash chain tail.

Hybrid Log. Each record pointed to by the hash table is stored in a log that spans disk and main memory, called HybridLog. Each record consists of an 8 byte header, a key, and a value. This header, among other information, stores a pointer to the previous address, the log address of the previous record in the hash chain (linked list). The log itself is divided into three contiguous regions: ① *mutable*, ② *read-only*, and ③ *stable* regions. The mutable and the read-only regions reside in-memory, while the stable one resides on disk. Records in the mutable region can be atomically updated in-place. Records in the read-only and stable regions are immutable, and use read-copy-updates (RCU) to the tail, adding to the hash chain tail via a compare-and-swap (CAS) op at the corresponding hash entry, thus providing linearizability guarantees [4].

This log design allows write-hot records to be accessed and updated very quickly, while scaling to many threads. As the tail grows, older log pages need to be flushed to disk and ultimately evicted from main memory. This is achieved in a latch-free manner by tracking several increasing addresses, i.e., a `BEGIN` address that tracks the first valid address in HybridLog, a `TAIL` that tracks the tail of the HybridLog address space, etc.

Epoch Framework. FASTER uses an epoch-based framework [11], which enables synchronization across threads in a lazy fashion, without using fine-grained latches. Most of the time, threads perform operations independently (e.g., update a record). However, some system-wide events (e.g., flushing log pages to disk) necessitate thread synchronization, i.e., to avoid accessing invalid memory regions or stale data. This is achieved using a global epoch counter, and thread-local ones, where the latter are periodically synced to the global counter. This mechanism allows actions to be executed *only after* all threads have agreed to a common view of the world.

Log Compaction. FASTER employs a *scan*-based compaction process to perform garbage collection. Here, the oldest disk-resident log records are read from disk and all potentially live records are stored in a temporary in-memory store. A full scan of the rest of the log confirms which of these records are indeed live. These live records are then inserted into the log tail. Finally, the `BEGIN` address is moved forward, effectively truncating the log.

3.2 Challenges with Large Skewed Workloads

Single-Log Design Implications. The original FASTER design uses a single log, which creates new challenges with large skewed workloads. One such issue is caused by the vast difference in popularity of compacted (cold) vs. in-memory (hot) records. In particular, during garbage collection (i.e., log compaction), cold records located

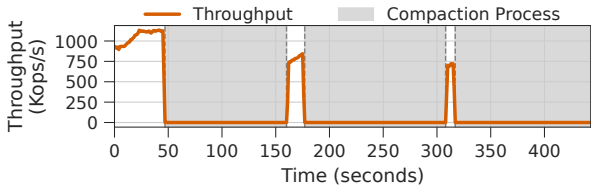

Figure 3: FASTER performance over time for a larger-than-memory RMW YCSB-F workload. Once log size budget is reached, a log compaction process copies cold tuples to the tail of the log. Yet, this results in hot tuples to be evicted to disk (over-and-over), significantly degrading performance.

at the beginning of the log are copied to the log tail. This not only increases tail contention with incoming user updates (leading to performance degradation), but also causes hot records (stored in-memory) to be flushed to disk, increasing write amplification. Notice that having cold records occupy the in-memory log region reduces the number of write-hot records that can be in-place updated in the mutable region. Thus, when log compaction finishes, hot records are appended to the tail of the log, further increasing the overall HybridLog size. This in turn triggers another log compaction process and, as shown in Figure 3, can lead to a "death spiral" behavior, where the system is entirely preoccupied with background compaction operations rather than serving user requests.

Inspired by LSM-tree based designs, F2 addresses these issues by introducing a separate log tier that stores write-cold records, eliminating log tail contention and "death spiral" behavior (Section 4). F2 also addresses key technical challenges to support efficient, latch-free user operations (e.g., RMWs) in this tiered design (Section 5.3).

Inefficient Record Compaction Mechanism. As discussed in Section 3.1, the original FASTER design employed a scan-based record compaction process to identify live records and compact them to its log tail. This scan-based approach has two drawbacks. First, the original implementation was single-threaded, limiting the maximum record compaction rate. Second, it requires (i) additional memory resources (i.e., a temporary memory buffer) to store potentially live records, and (ii) a full scan of the HybridLog to reason about record liveness. This approach not only led to transient memory spikes, but also wasted memory and I/O bandwidth resources, which could have been used instead to cache more hot records in-memory and perform more user I/O operations, respectively.

F2 solves both problems by introducing a *lookup*-based compaction mechanism, based on our new *Conditional-Insert* primitive (Section 5.1). This new compaction process is multi-threaded and utilizes only minimal memory and disk bandwidth resources to reason about record liveness (Section 5.2).

Large Indexing Overhead. FASTER's hash index tracks all keys in the store and incurs a fixed, per-key memory overhead of 8 bytes. Although this overhead is manageable for small datasets, when dealing with billions of keys, this leads to a prohibitively large memory footprint (e.g., 64GiB to index 8 billion records). One might try to constrain the index size, e.g., by restricting the number of buckets, yet, this creates too many hash collisions, making point reads require multiple I/O ops to follow the hash chains on disk.

To address this limitation, F2 introduces a separate two-level hash index that spans both memory and disk, and can index billions of (cold) records with minimal memory footprint (Section 6).

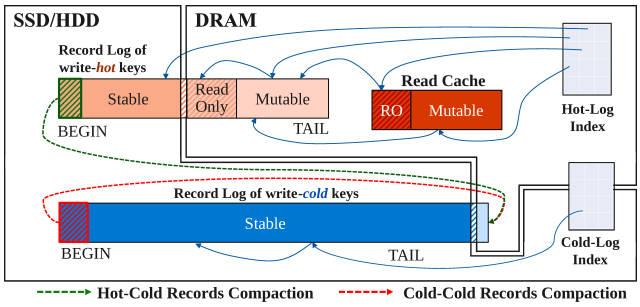

Figure 4: F2 Architecture

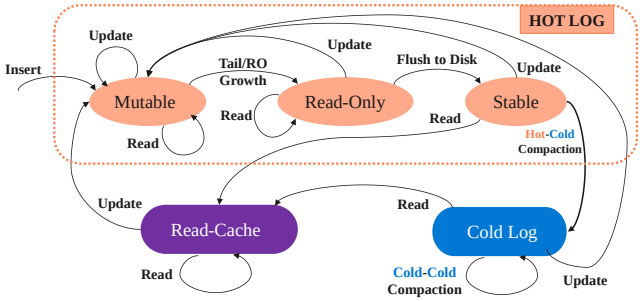

Figure 5: Lifecycle of a Record in F2

Suboptimal Handling of Read-Hot Records. In FASTER, write-hot records are effectively in-place-updated in memory through the HybridLog design. However, when read- and write-hot working sets are non-overlapping, read-hot records may get flushed to disk to make space for write-hot records and vice versa. This leads to poor performance: reads incur I/O and writes incur tail growth.

To effectively handle both read- and write-hot working sets, F2 introduces a dedicated in-memory read-cache that provides immediate access to read-hot records (Section 7).

4 F2 OVERVIEW

F2 is a concurrent, lock-free, key-value store that can serve larger-than-memory skewed workloads with ① low memory footprint, ② saturation of the available NVMe SSD bandwidth, and ③ minimal disk wear. F2 supports point `Reads`, `Upserts`, `Deletes` (using tombstone markers), and atomic updates in the form of read-modify-writes (`RMWs`). F2's design also provides linearizable semantics.

Figure 4 depicts F2's architecture. First, F2 incorporates a log-structured record store that keeps write-hot keys (i.e., *hot log*), alongside its respective hash index (i.e., *hot-log index*). Second, it integrates a separate record log for storing write-cold keys (i.e., *cold log*), alongside its respective hash index (i.e., *cold-log index*). Finally, a *read-cache* lies between the hot-log index and the hot log, which maintains a set of disk-resident read-hot records in a separate in-memory store. F2 automatically places records on the appropriate component, based on their observed read/write hotness.

4.1 Components Overview

Hot-Log Index. The hot-log index employs a lock-free hash table design, similar to the one of the original FASTER, that is stored in memory. Each bucket entry contains a pointer (i.e., address) to a record, whose key hashes to this entry. However, this record may now reside in either the hot log or read-cache. Each record points to a (previous) record (if any) in the hot log, forming a hash chain. Using this hash chain, F2 accesses records with matching key, stored in hot log or in read-cache.

Hot Log. The goal of the hot log is to enable write-hot records to be retrieved and updated promptly, even when many threads are concurrently operating on the log. Therefore, the hot log is an instance of FASTER's HybridLog, which is coupled with the hot-log index. The hot-log index, indexes only the hot-log (and read-cache) records, requiring fewer memory resources. Although the organization of the hot-HybridLog is largely the same (e.g., insertions at the tail of the log, in-place updates or RCU), we significantly alter its compaction behavior (as we discuss in Section 5).

Cold-Log Index. The goal of the cold-log index is to reduce the memory resources needed to index cold records. It is based upon a *two-level* index design, and consists of a (small) in-memory structure and a (large) on-disk one. The core idea is to group multiple hash index entries together, to create *hash chunks*, and then index these chunks in-memory, while storing the actual chunks on-disk (more details in Section 6).

Cold Log. The addition of the cold log enables the physical separation of write-hot and write-cold records. Cold-log organization is similar to hot-log, with the exception that (almost) all records now reside on disk. Accessing a cold-log record requires two I/O ops: one for retrieving the hash chain from the cold-log index, and one for reading the actual record from the log. Yet, the cold-log integration eliminates the contention at the hot-log tail, fixing the "death-spiral" behavior of the original FASTER design.

Read Cache. While write-hot records reside in the in-memory part of the hot log, write-cold records do not. This leads to poor performance for disk-resident *read*-hot records, as F2 has to perform one (or more) I/O operations each time this record is requested. The read-cache allows immediate access to read-hot records, even if they are stored on disk (more in Section 7).

4.2 Lifecycle of a Record

Figure 5 depicts the lifecycle of a record in F2. The user first inserts a record into the store, by creating a new record in the hot log *tail*. Initially, the record is created in the mutable region (in-memory), and any subsequent updates are performed in-place. As other records are appended to the log, our record eventually moves to the read-only region. If a user issues an update for this key, we perform an RCU to append the updated record to the hot log tail. As long as our record is in-memory (read-only or mutable hot-log region), read-cache is not used.

When our record has not been updated for a while, it is eventually flushed to disk (i.e., stable region), as the in-memory regions are populated with newer records. Here, a user update will result in RCU to the hot-log mutable region, while a user read will copy the record to the in-memory read-cache (after being fetched from disk). This allows future user reads to be served directly from read-cache, avoiding any extra I/O operations. As long as the user issues reads for our record, it remains inside the read cache (i.e., write-cold, read-hot record). Yet, if no reads occur for some time, our read-cached record is ultimately evicted.

Assuming no further updates, our record ultimately ends up in the back (i.e., `BEGIN`) of the hot log. As other records are being

appended, the hot log grows larger, which necessitates moving the write-cold records from the hot-cold to the cold-log. This is achieved through a background *hot-cold compaction* process (green arrow in Figure 4). During this process, *live* records from the beginning of the hot log (i.e., compacted region) are copied to the *tail* of the cold log. Once all live records have been copied, the hot log is truncated, invalidating all records in the compacted region. Note that during hot-cold compaction, the hot log tail remains intact, and is able to fully accommodate other incoming user requests.

Once the hot-cold compaction finishes, our record now resides in the cold log. In fact, because the hot set for a skewed workload is relatively small, most records end up in the cold-log. The cold log resides (almost) entirely on disk, as keeping those (write-cold, read-cold) records in-memory does not bring any benefits. At this point, a user update request creates a new record to the mutable region of the hot log, while a read request copies our disk-resident record into the read cache (causing it to become read-hot write-cold). Assuming that no such requests take place, our record remains cold, and eventually arrives at the back (i.e., `BEGIN`) of the cold-log.

As the cold log is populated with more records, older non-live records need to be garbage-collected. To do so, we employ another background process, i.e., *cold-cold compaction*. This process copies *live* cold log-resident records from the back (i.e., `BEGIN`) of the cold log to *its tail* (red arrow in Figure 4). Once all live records have been copied, we truncate the cold log, completely removing non-live records from F2. Notice how both hot-cold and cold-cold compactions copy records to the *cold*-log tail, avoiding any tail contention in the hot-log.

4.3 Implementation and Usage

F2 is exposed as an embedded library implemented as part of the FASTER C++ code-base [26], with around 11k SLOC. As in the original FASTER implementation, we leverage template meta-programming to avoid runtime overheads, and employ a large set of tests to check for correctness under concurrent execution.

F2's API for performing user operations is identical to the FASTER one, enabling existing users to seamlessly transition to F2. Given its more flexible design, F2 provides additional options for users to configuration, to meet the application (and environment) demands. Listing 1 describes how one could configure and initialize F2.

While optimally tuning the parameters of any KV store is a challenging task [28], we provide here some guidelines for configuring F2. First, to avoid multiple I/O ops when retrieving disk-resident records, we recommend sizing the hot- (cold-) log indexes based on the expected number of *unique* keys for each log. For example, indexing 1B records, with 125M being hot, requires at least 1GiB (16M buckets × 64B each) for the hot-log index, and 256MiB (i.e., index 28M hash chunks of 256B each using 3.5M buckets) for the cold-log index. Second, for read-heavy workloads, we recommend trading-off in-memory hot-log space for read-cache one (and vice versa). For instance, in Section 8.2 we show that properly setting (i.e., increasing) the read-cache size can improve F2's throughput by 19-27%. Finally, the hot/cold log in-memory organization can be configured based on suggestions from the original FASTER paper [11] (e.g., set log mutable region to 90% of its in-memory size).

The next three sections cover details of each major component in F2, starting with the tiered record log in Section 5.

```
// Define F2Kv instance
using HotIndex = MemHashIndex<Disk>;
using ColdIndex = ColdIndex<Disk>
using F2KvInst = F2Kv<Key, Value, Disk, HotIndex, ColdIndex>;

// Configuration
ReadCacheConfig rc_config { <mem_hlog_sz> };
F2KvInst::HotIndexConfig hi_config{ <hash_table_sz> };
F2KvInst::ColdIndexConfig ci_config{ <hash_table_sz>, <mem_hlog_sz>};

// Initialize F2Kv
F2KvInst store {
    hi_config, <hot_log_mem_sz>, <hot_log_disk_fp>,
    ci_config, <cold_log_mem_sz>, <cold_log_disk_fp>,
    rc_config };
```

Listing 1: F2 C++ Initialization Code Example

5 TIERED RECORD LOGS

In F2's tiered design, live records undergo continuous compactions through hot-cold and cold-cold compaction processes. With hot (and cold) logs potentially managing millions (billion) keys, performing log compaction in a CPU-optimized, memory-efficient manner is of utmost importance.

To this end, we introduce a new primitive, *Conditional-Insert* (`CI`), that is used as a building block for F2's lookup-based compaction algorithm and user `RMW`. Then, we describe how threads perform record compaction and user operations, emphasizing on correctness issues under concurrent execution. Finally, we highlight such an issue that arises when `Reads` are performed concurrently with cold-cold compaction, and explain how it is addressed.

5.1 Conditional-Insert Primitive

Our goal is to develop a primitive that can append a record to a log, only if no other record with a matching key has been appended in the meantime. In other words, we want to insert this key *conditional* on no newer insert happening during the process. More formally, given a record R, stored in a record log (i.e., *source* log) and a `START` address in the log, `CI` appends the record to the tail of *target* record log (same or different to the source log), only if there exist no record(s) with a matching key in the `(START, TAIL]` address range of the source log. If a record exists in this range, the operation aborts (i.e., becomes a no-op). Figure 6 depicts these two possible outcomes (i.e., success, abort). For ease of exposition, we initially make two assumptions: (1) the source log is the same as the target log, and (2) the `START` address matches the log address of R we wish to append to log tail.

Conditional-Insert is implemented as follows. First, we perform a lookup at the index of the source log, to find the entry corresponding to R's key, and we store a copy of that `entry` in the operation context. The index entry contains the log address of the most-recent record (of this hash chain) in the log. Starting from this address, we follow this hash chain backwards, possibly issuing read I/O request(s). If at any point during this backwards search, we encounter a record that matches our key, we promptly *abort* (i.e., *non*-live record). Otherwise, we arrive either at the end of the hash chain or at some address outside the search range (i.e., `address` < `START`), meaning we can now copy R to the target log tail.

Appending R to the log tail must now ensure that no other newer records with the same key were inserted in the meantime (e.g., by a concurrent user or compaction operations). Otherwise, it is possible

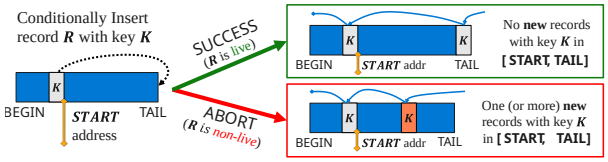

Figure 6: *Conditional-Insert* op and possible outcomes.

that we accidentally overwrite a newer version with an older one (i.e., lost update anomaly). Hence, we leverage the previously saved index `entry`. In particular, we first write the record to the log, and then perform an index update (using an atomic CAS operation), expecting that the index `entry` remained unchanged. If CAS fails, then newer records were inserted in this hash chain. In this case, we invalidate our written log record, and restart our search on the hash chain, but *only* check the newly-introduced records on the hash chain. As before, if we encounter a (newer) record with a matching key, we abort; otherwise we try to update the index entry again. We repeat the above process, until either succeeding at appending the record (i.e., record is live), or aborting altogether. This ensures that in either case, no newer record(s) have been overwritten.

5.2 Lookup-based Compaction

We now describe how we can leverage the `CI` primitive to make compactions faster and more memory-efficient. Given a source log and a (potentially different) target log, lookup-based compaction process consists of the following phases:

① **Copying phase**: Starting from the beginning of the source log (oldest records), we sequentially scan a fixed `[BEGIN, UNTIL]` range. This range represents a specific percentage of the entire log (e.g., 10%). For each record scanned, we issue a Conditional-Insert operation. If the record is live, then a copy is atomically created at the tail of the target log. Note that during this phase, copies of the *same* live records may exist in both logs (for hot-cold compaction) or in both ends of the cold log (for cold-cold compaction).

② **Truncation phase**: Once we process all keys in this source log region (i.e., all live records have been copied to target log), we *truncate* the source log, by atomically setting the `BEGIN` address to `UNTIL`. Then, all hash index entries that point to invalid addresses (i.e., `address < BEGIN`), are invalidated using CAS operations.

Cold-Cold Compaction. Our initial assumptions with `CI` were: (1) the source log is the same as the target log, and (2) the `START` address is the log address of the record we wish to append to the log tail. Notice how with these assumptions, `CI` can safely compact a *live* cold-log record to the cold-log tail, even when newer records are being appended to the same tail (e.g., hot-cold compaction).

Hot-Cold Compaction. Here we consider the case where the source log (i.e., hot log) differs from the target log (i.e., cold log), relaxing our first assumption. As before, we follow the hash chain backwards for any key matches, ultimately exploring the entire hot log address range. We are now ready to copy our record to the cold log tail. Since the records in the cold log are older by-design, they naturally satisfy our invariant and we can just issue an `Upsert` to cold log. The only implication here is that we might `Upsert` non-live keys (e.g., if newer records entered the hot log in the meantime). While these superfluous writes might lead to slightly more disk operations, correctness is still ensured. We late relax our second assumption, when discussing user `RMW`s (Section 5.3).

Concurrent Conditional-Insert. Our key invariant is satisfied, even in the presence of concurrent `CI` ops. The only concern here is a possible record re-ordering at the target log (as a result of non-sequential record processing), that might lead to overwriting of newer records. First, we note that when compaction threads operate on records with different keys, records re-ordering poses no correctness issues (i.e., different hash chains). Now consider a scenario where two threads, T_1 and T_2, operate on two different records with the same key, R_1 and R_2. We know that both records are part of the same hash chain, and thus one record, e.g., R_1, is located in front (i.e., higher log address) of the other, e.g., R_2, in the hash chain. This suggests that R_1 is live, while R_2 is not. When the two threads call `CI`, *only* T_1's request succeeds. This is because by following the hash chain backwards, T_2 inevitably encounters R_1 (i.e., same key) and thus aborts. Notice how T_1 does not encounter R_2 at all, since it is located before R_1 in the log. Generalizing this to many threads operating on multiple records with the same key, it follows that *exactly one* record for each key is compacted.

Multi-threaded Compaction. To achieve shorter compaction times, multiple threads can participate in the compaction process, issuing concurrent `CI` ops on different records. Participating threads coordinate using the epoch protection framework (overhead is negligible), and correctness under concurrent `CI` ops is always ensured (see above discussion). During copying phase, we employ an in-memory circular buffer that is populated with several (32MiB) log pages; initially the first pages in the `[START, END]` range. Records residing inside the log pages are distributed to compaction threads using fetch-and-add atomic operations, while the next log page(s) (if any) are prefetched from the disk to avoid any processing stalls.

Summary. Unlike FASTER's scan-based compaction, F2's lookup-based compaction (i) requires minimal memory resources (three log pages, or 96MiB), (ii) is multi-threaded, enabling much faster compaction times (i.e., 5.2× as shown in Section 8.2), and (iii) performs only the absolute necessary disk operations to determine record liveness. Hence, F2 can even compact billion-key logs, which would otherwise be infeasible with the original FASTER.

5.3 User Operations

Upsert and Delete. We implement `Upserts`/`Deletes` as follows. First, we perform a lookup in hot-log index, then append the new record to hot log tail, and finally CAS the index entry to point to the our newly-appended record. If CAS fails, we mark the record as invalid and retry. In `Delete`, a tombstone record is *always* inserted, even when the entry for the key does not exist in the hot-log index, as (non-tombstone) valid records may still exist in the cold log.

Read. We first issue a `Read` op in the hot log (i.e., most recent records). If a record is found there, we return it to the user. If a tombstone record is found, we return `NOT_FOUND`. If no record is found in the hot log, we then issue a `Read` to the cold log. As before, we either return a valid record, or `NOT_FOUND`. The above algorithm provides correct results in most cases. Yet, under concurrent cold-cold compaction, a `Read` might return `NOT_FOUND` even if a record exists. We discuss this anomaly in Section 5.4. We later explain how `Reads` are modified when using a read-cache (Section 7).

Read-Modify-Write. A user RMW operation *atomically* updates the value of a key based on user-provided logic, or inserts a record

Algorithm 1: User Read-Modify-Write (RMW) in F2

```
1  function Status Rmw(key):
2      start_addr = hot_log.index.FindEntry(key).address;
       // Try RMW record in hot log
3      rmw_status = hot_log.Rmw(key, create_if_not_exists=false);
4      if rmw_status ≠ Status.NOT_FOUND then
5          return rmw_status // Record updated!
       // No record in hot log – try Read from cold log
6      read_status = cold_log.Read(key, record);
7      if read_status == Status.OK then
8          new_value = UpdateValue(key, input, record.value);
9      else new_value = InitialValue(key, input) ;
       // Check if stored start log address is valid
10     if start_addr < hot_log.begin_address then
11         goto RETRY;
       // Append updated value; abort if new records
12     ci_status = hot_log.ConditionalInsert(key, new_value, start_addr);
13     if ci_status == Status.OK then
14         return Status.OK;
15     RETRY:
16     return Rmw(key);
```

with an initial value if the key does not exist. The first step in RMW is to locate the most-recent record with a matching key. In F2, this record may reside in either hot or cold log.

Algorithm 1 details how user RMWs are performed. First, we issue a RMW request to the hot log (L3). Since write-hot records are usually stored in the hot log, this makes the common case fast, as we can quickly return upon updating (possibly in-place) the record (L5). If no record matching our key exists in hot log, we refrain from creating a new record. Instead, we issue a Read request to the cold log (L6), as a record may exist there. In this case, we update its value (L8) using the user-provided logic (i.e., UpdateValue); otherwise (L9) we use the initial value (i.e., InitialValue). Finally, we try to append the updated record to the hot log tail (L12).

Concurrent to our RMW operation, newer records with the same key might have been appended by the user, causing a change in the hash chain. To ensure that we always use the most-recent record for our key, we leverage the CI primitive we presented earlier. More specifically, at the very start of the user RMW (L2) we fetch and store the address where the hash chain begins in the log (i.e., start_addr). We later use that address to determine whether any new records has been inserted in the (start_addr,TAIL] range (L12). If this is the case, we abort the user RMW operation and retry again. Note that the hot log RMW request (L3) will now most likely succeed, since a record now exists in the hot log (assuming small chance of hash collisions). Otherwise, CI successfully inserts the updated record to the tail of hot log. In the rare case where the range (start_addr,TAIL] is invalid (L10), caused by log truncation (e.g., due to concurrent hot-cold compaction), we retry from the start.

5.4 False-Absence Anomaly

A cold log Read traverses the entire log, by following the hash chain. However, it is possible to fail locating a record that is indeed present in the log, incorrectly returning NOT_FOUND.

Consider the scenario depicted in Figure 7, where a Read operation is issued in the cold log (after a failed search in the hot log) for a given key, K_1. At the same time a concurrent cold-cold compaction is being executed. Assume that only a single record

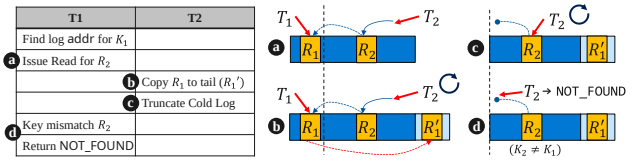

Figure 7: False-Absence Anomaly Scenario

R_1 for key K_1 exists in the cold log, and it is located in at the very beginning of the cold log. The following events transpire in-order. First, thread T_1 issues a Read, and thus performs a lookup in the cold-log index to find the log address of the first record in the hash chain (i.e., R_2). T_1 then issues a read I/O request to fetch R_2 from disk. Unbeknownst to T_1 at the time, R_2 has a different key $K_2 \neq K_1$, i.e., due to a hash collision. While R_2 is being read from disk, thread T_2 performing the compaction, manages to copy all live records to the tail of the cold log. Thus, a copy of R_1, R'_1 has been written to the tail. T_2 then proceeds and truncates the log, invalidating R_1. Then, R_2 is finally fetched from disk. T_1 only now realizes that R_2's key $K_2 \neq K_1$. Now, it follows the hash chain backwards, only to find that the previous address, originally pointing to R_1, is now invalid (due to log truncation). Thus, T_1's Read op returns NOT_FOUND, as it deduces that no record with key K_1 exists in either hot or cold log. However, this is clearly incorrect, as R'_1 exists in the cold log tail.

This anomaly occurs because T_1 is not aware of the concurrent cold-cold compaction. To address this, one might try to employ some locking scheme (e.g., where cold log truncation is done only when no cold-log Reads are active), or temporarily store every live record in compacted region into a separate in-memory store. However, the former introduces starvation issues (e.g., for constant stream of cold-log requests), while the latter introduces additional memory overhead. Instead, we fix this issue by employing a shared atomic counter that tracks the number of completed log truncations. On every cold-log Read request, we now first fetch and store (in the operation context) the number of log truncations. We then follow the respective hash chain backwards, as before. If we ultimately find no record with a matching key, we then check if a log truncation took place, i.e., by comparing the current counter value with the one we stored previously. If a log truncation indeed occurred, we traverse just the *newly*-introduced part of the hash chain (if any) to check whether a record was indeed compacted. While this scheme introduces some extra work to few cold-log Read ops to ensure correctness, log truncations are infrequent, thus the common case remains unaffected.

6 INDEXING COLD RECORDS

When F2 handles larger-than-memory skewed workloads, most records end up in the cold log. Yet, indexing this many records using solely in-memory structures incurs large memory overheads. For instance, indexing a billion keys using a design similar to the hot-log index requires at least 8GiB. Other systems require even more memory, as they store extra metadata (e.g., 19GiB for KVell).

To this end, we introduce a *two-level* hash index design, as shown in Figure 8. The core idea is to perform in-memory indexing at a coarse-grained level. Specifically, we first group multiple hash index entries together, to create *hash chunks*. Each hash chunk holds a fixed power-of-two number of entries (e.g., 32). Then, we use an in-memory hash table to index these chunks (1st level), while the

actual chunks are stored in a log-structure disk store (2nd level). To facilitate concurrent updates on the index chunks, we leverage a HybridLog instance, configured with a small in-memory region.

This cold-log index design comes with three main benefits. (1) By reusing existing lock-free components (i.e., HybridLog), we can easily ensure multi-threaded correctness. (2) The hash chunk size allows us to control the in-memory overhead of the index: increasing (decreasing) hash chunk size results in fewer (more) hash chunks required to index the same number of keys, leading to a smaller (larger) in-memory hash table. Finally, (3) by adopting a log-structure design to store hash chunks, we can now use small chunks (i.e., less than 4KiB disk block size), leading to low write amplification, as a single disk block write updates many chunks. We note that our cold-log index design could be replaced by any other concurrent memory-efficient index structure as well.

Finding an Entry. Finding a cold-log hash entry consists of (i) reading the hash chunk from the hash chunk log, and (ii) extracting the respective hash entry. Given a key, we first compute its hash value, h, and then identify the respective hash chunk using a subset of h's bits. Then, we issue a Read in the hash chunk log. Once the chunk is read, we identify the hash entry for our key (different subset of h's bits). Finally, we issue a Read in the cold-log, as this hash entry now points to a cold-log hash chain.

Modifying an Entry. Modifying a cold-log hash entry consists of (i) reading the hash chunk from the hash chunk log, (ii) applying our update to the specific hash entry, and (iii) writing the entire hash chunk back to the hash chunk log. Notice how this process naturally fits the functionality of a RMW. Thus, given a record key, we first identify the respective hash chunk (from the key hash), and then issue an *atomic* RMW to the hash chunk log; if no hash chunk exists, we create an empty one (i.e., all entries are invalid).

Configuration. We configure the index so that we use less than 1B of memory per cold key. We set the size of each hash chunk to 256B, resulting in 32 hash index entries being stored in a single chunk. Smaller chunk sizes (e.g. 64B) trade off less write amplification for a larger index memory footprint, and vice-versa. We set the number of chunks based on the expected number of unique cold keys, e.g., for 250M keys, we can use 250M/32 ≈ 8M chunks (i.e., one hash entry per key). Indexing 8M chunks in-memory requires just 64MiB.

7 SERVING ON-DISK READ-HOT RECORDS

F2's two-tier design physically separates write-hot records from write-cold ones. However, this leads to poor performance when serving disk-resident read-hot records, as F2 would have to perform I/O each time such records are requested. To address this, we introduce a dedicated, in-memory read cache.

We tightly integrate the read-cache with the hot-log index, by extending the hot-log index chains to span both the read-cache and the hot-log (Figure 4). Specifically, a hot-log index entry may now (optionally) point directly to a record in the read-cache, before continuing to point one (or more) hot-log records. The entry may also point to the hot-log directly, bypassing the read-cache altogether. Read-cache keeps *replicas* of records originally located in the hot or cold log; the original records are never removed from their position in the log. Records already residing in-memory (e.g., hot-log in-memory region) are never inserted in the read-cache.

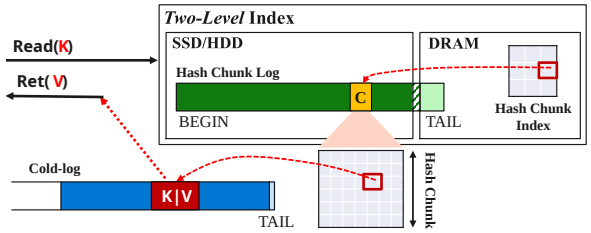

Figure 8: Cold-log Index architecture, and Read op lifecycle.

Read-cache is implemented as a separate in-memory HybridLog, and thus contains only a mutable and a read-only (RO) region. Records are inserted at the log tail (i.e., mutable region), and eventually evicted at the head (i.e., RO region). If a record residing in the RO region is requested again by the user, we update its presence in the read-cache, by copying it to the log tail. This gives our record a second-chance (similar to Second-Change FIFO cache [46]), and further ensures that the most read-hot records are never evicted.

The key invariant that the read-cache should satisfy is that for a given key, it should keep its *most-recent* record. We next describe how user operations are modified to ensure this.

Upsert, RMW, Delete. When inserting/updating a record for a given key to F2's hot log, we need to invalidate the (older) read cache-resident entry (if any). Since the hot-log and read-cache share the same hash chains, we can do so by traversing the record hash chain, invalidating any potential records located in the read cache (by setting a bit in the record header), and then proceed with the request as before (Section 5.3).

Reads Given a key, we first identify its respective hash chain (by consulting the hot-log index), and then follow the chain backwards as before. If we encounter a valid record with a matching key in read-cache (i.e., most-recent), we promptly return it to the user. Otherwise, we continue the search in the hot (and possibly in the cold log). If we find a disk-resident record in either log, we first try to insert it to the read-cache; this might fail if another thread concurrently updates the same key. Ultimately, we either return a valid record to the user or NOT_FOUND.

Records Eviction When the read-cache becomes full, records need to be evicted to make space for other (read-hotter) records. The only challenge is to ensure that both during, and after the eviction process, the system always remains in a consistent state. Thus, the records eviction process involves re-arranging hash chains that point to soon-to-be-evicted read-cache records, in a lock-free manner, without impeding incoming user ops.

Eviction is performed on a per-page granularity. For each record in the page, we first determine if it is valid; if it is, we skip the record. Otherwise, it means that the respective hash index entry points to this soon-to-be-evicted record, and thus needs to be modified to point to the hot log instead. To do so, we retrieve the next address in the hash chain, which always points to the hot log (i.e., at most one record per hash chain resides in read cache). Then, we CAS this address to the hash bucket entry; if we succeed, we move to the next record. Otherwise, some other thread altered the hash chain in the meantime, e.g., by inserting a new record in the chain. Therefore, our record has been invalidated, and we again move to the next record. Note that multiple threads can participate in this process safely, as no two evicted records share the same hash chain.

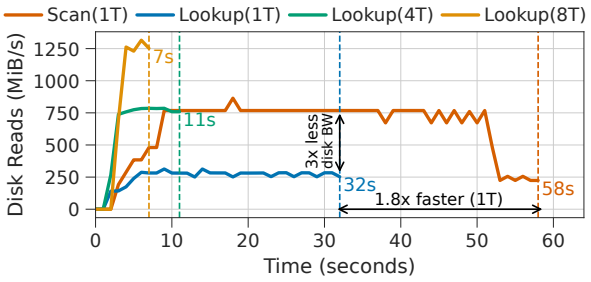

Figure 9: FASTER's scan-based vs F2's lookup-based single-log compaction disk read throughput, when compacting 2GiB (out of 30GiB). For same target disk BW, lookup-based finishes 5.2× faster and uses 25× less memory (120MiB vs 3GiB).

8 EVALUATION

F2 was recently merged into the FASTER code-base, and we do not yet have production workloads to test it on. However, we find that we can emulate the characteristics of larger-than-memory skewed workloads with available YCSB and MixGraph benchmarks. In this section, we evaluate F2 along multiple dimensions and and compare against state-of-the-art KV stores. In particular, we show that:

- F2's lookup-based compaction is 5.2× faster than FASTER's scan-based one and uses 25× less memory. F2's read-cache improves throughput by up to 1.27× for read-heavy workloads (Section 8.2).
- F2 provides meaningful speedups (2–11.9×) across YCSB and real-world MixGraph workloads, with on average 1.4-2.9× lower write amplification than LSM stores (Section 8.3).
- F2 remains robust to varying degrees of key access skewness, outperforming the best SoTA stores by on average 2.6× (1.8×) for high (low) skewness levels (Section 8.4).
- F2 outperforms the best systems on small memory budgets (2.5-10% of DB size) by 2.7× on average, while matching the best system for larger (≥ 20% of DB size) budgets (Section 8.5).

8.1 Experimental Setup

System. We conduct all experiments on CloudLab [25], using a `sm110p` node that is equipped with a 16-core Intel Xeon Silver 4314 CPU, 128GiB of RAM, and four Samsung PM9A3 NVMe SSD (PCIe v4.0) devices, running Linux kernel v5.4. For our experiments, we use all four NVMe disks in RAID-0 formation (using `mdadm`) with `ext4` filesystem and disk block size of 4KiB.

YCSB. We use YCSB [17] workloads with 250 million keys (8B keys, 100B values) and run YCSB-A (50% reads, 50% updates), YCSB-B (95% reads, 5% updates), YCSB-C, (100% reads), YCSB-F, (50% reads, 50% RMWs), YCSB-D read-latest workload (95% reads, 5% inserts), and a custom YCSB-W (5% reads, 95% updates). We model the skewness of real-world workloads with a *Zipf* distribution, using the *YCSB default* $\theta = 0.99$ or equivalently $\alpha = 1/(1-\theta) = 100$, and other skewness factors ($\alpha \in [3, 1000]$). With the default YCSB skewness factor ($\theta = 0.99$ / $\alpha = 100$), 90% of accesses go to 18% of records.

MixGraph. We employ two MixGraph (MG) benchmarks with 250M keys, i.e., All-Dist (AD) and Prefix-Dist (PD), which were developed to emulate the real-world workloads observed in Meta's production services [9]. They use 48B keys with variable-sized values, and consist of 83% reads, 14% writes (and 3% seeks, which we skip). All-Dist clusters hot keys close together in the key-space, while Prefix-Dist further partitions the key-space into smaller hot/cold key-ranges and issues more requests to hot ranges.

Measurements & Resources. For each experiment, we load the dataset into the system (e.g., 30GiB), warm it up with 25M ops, and then run each workload for 300M ops measuring system throughput. We report average throughput in thousands of requests per second. Unless otherwise noted, we set the available memory to 10% of the dataset set, i.e., 3GiB for YCSB 250M key-value dataset, and 4GiB for MixGraph one. We also manually pin user threads to hardware cores. The above is done via the exposed user configuration parameters of each system and further enforced via Linux `cgroup` [14]. This setup is similar to prior works [9, 16, 49].

Baseline Systems We compare F2 against several state-of-the-art systems including, SplinterDB [15, 16] (commit `1939a12`), RocksDB v8.11.4 [9], FASTER [11], Kvell [31] (commit `af10b7a`), and LeanStore [30] (commit `26d4a46`, `io` branch). We configure all systems to use Direct I/O disk ops, and disable any persistent layer (e.g., write-ahead logging), compression, and checksums (if supported).

For all baselines, we set their memory-related parameters as recommended by their documentations (for the given memory budget), and apply all available point operation optimizations. For RocksDB, we enable Bloom Filters (with 10 bits) and use data block hash index [29, 44], using the `OptimizeForPointLookup()` option. SplinterDB employs quotient filters, which we enable.

We configure FASTER with fixed 1GiB hash index (8 tag bits) and 1.75GiB HybridLog in-memory region (90% mutable like in [11]). For most experiments, we replace FASTER's original scan-based compaction with F2's lookup-based one, to avoid exceeding the memory budget during compaction process.

F2 Configuration. Unless otherwise noted, we configure F2 as per our guidelines (see Section 4.3). The mutable region of the hot log is set to 90% of the log in-memory region. We use 512MiB memory budget (4M hash buckets) for the hot-log index and assign 512MiB to read-cache. We only use 64MiB for the in-memory region of the cold-log and configure the cold-log index to use 256B hash chunks that are indexed in-memory using another 64MiB. The remaining budget (≈ 1.75GiB) is mostly used for the in-memory region of the hot log. To trigger both hot-cold and cold-cold compactions, we set the disk budget of the hot (cold) record log to 125% of database size: 5GiB (35GiB) for YCSB and 7.5GiB (42.5GiB) for MixGraph. Note that we use the same F2 configuration for all workloads to perform a *fair* comparison against the baselines.

8.2 Comparing F2 with the Original FASTER

Lookup- vs Scan-based Compaction. We compare F2's lookup-based compaction with our original scan-based compaction, when compacting 2GiB worth of records from a single log (to its tail). As shown in Figure 9, lookup-based compaction can lead to faster compaction times, i.e., 1.8× when using a single thread, or 5.2× when using the same disk bandwidth (with 4 threads). More importantly, compactions consume 25× less memory: we measure the peak memory utilization during lookup-based compaction at 120MiB, compared to 3GiB for scan-based compaction. This is because the lookup-based approach only stores a fixed amount of data in-memory (i.e., several log pages), and not a growing temporary memory buffer of live records (as the scan-based approach does).

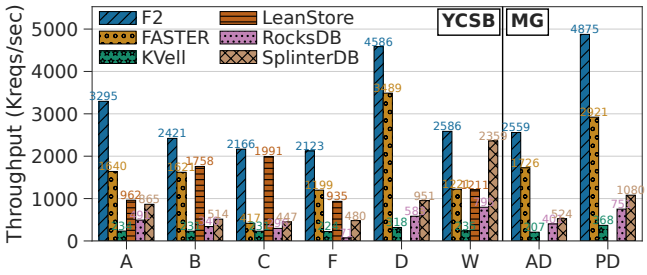

Figure 10: Throughput of F2 and baselines on all workloads.

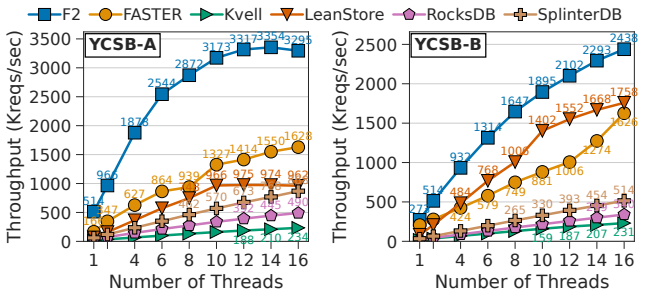

Figure 11: Thread scaling of F2 and baselines systems for Zipfian YCSB-A (left) and YCSB-B (right).

Impact of Read-Cache. We now evaluate the impact of F2's read-cache in system throughput. We observe that compared to the default 512MiB read-cache, F2 can deliver up to 19% (27%) higher throughput for read-heavy YCSB-B (YCSB-C), if read-cache is configured properly. Moreover, for the read-only YCSB-C, we see that even a small cache (of 256MiB) can provide almost a 8.3× speedup over not using one. Interestingly, for read-cache sizes of 1.5GiB (or more), an (increasingly larger) number of cold-resident read-hot records can now be stored in-memory. This occurs because the read-cache is now able to keep in-memory an (increasingly larger) subset of read-hot records that actually reside in the cold log.

8.3 Comparison to Baselines

System Throughput. We now evaluate F2 performance compared to the baseline systems, when using all 16 CPU cores and with the available memory as 10% of the workload dataset (i.e., 3GiB). Figure 10 shows system throughput (Kops/sec) for YCSB and MixGraph workloads. F2 outperforms FASTER (2.1×), LeanStore (2×), KVell (11.9×), SplinterDB (4.6×), and RocksDB (11.8×) on average. For update-heavy YCSB-A, F2 speedups stem from its tiered log-structured design, which enables fast ingestion speeds, while keeping the most-frequently accessed keys in the in-memory part of hot-log. For read-heavy MixGraph (YCSB-B, YCSB-C), F2 is 1.5–4.8× faster than FASTER and SplinterDB, while matching LeanStore for YCSB (its driver does not support MixGraph), mostly due to its read-cache and smaller hash chains (i.e., less read I/O).

KVell's poor performance is attributed to its large in-memory index, parts of which are continuously being swapped out to disk. LeanStore's page-oriented buffer pool cannot keep the hot records in-memory, as these are not always clustered together in the same page; however, it saturates the I/O bandwidth, achieving good performance. FASTER's "death-spiral" effect is observed on update-intensive YCSB-A/F/W workloads: i.e., continuous compaction of write-cold records to log tail, leading to the eviction of hot ones.

Table 1: Average user request latency (us) for F2 and baselines.

	Workload	FASTER	LeanStore	RocksDB	SplinterDB	F2
Read	YCSB-A	41.5	254.3	53.0	42.1	41.7
	YCSB-B	47.2	122.0	50.7	42.1	45.3
	YCSB-W	40.9	311.8	63.1	58.8	39.2
Write	YCSB-A	4.2	262.8	10.3	3.9	3.4
	YCSB-B	4.8	128.8	6.7	3.9	3.9
	YCSB-W	4.1	324.9	17.3	3.5	3.3

Table 2: Disk read (RA) and write (WA) amplification for F2 and baselines on YCSB-A, YCSB-B, YCSB-W, and MG-PD.

	Workload	FASTER	KVell	LeanStore	RocksDB	SplinterDB	F2
RA	YCSB-A	7.23	91.93	66.87	21.47	17.09	6.41
	YCSB-B	5.03	48.71	35.22	16.51	15.75	5.5
	YCSB-W	38.6	95.32	72.23	109.12	52.97	53.68
	MG-PD	1.91	33.72	-	5.23	4.61	1.84
WA	YCSB-A	2.62	31.17	34.47	5.28	2.18	1.23
	YCSB-B	1.21	33.79	38.73	2.64	2.52	1.77
	YCSB-W	1.75	31.48	32.87	5.81	1.85	1.48
	MG-PD	2.26	32.01	-	2.52	2.38	2.43

Overall, F2 provides meaningful speedups (2–11.9×) across many YCSB and real-world MixGraph workloads.

Latency. We now evaluate F2 request latency compared to baselines. We use a single thread to avoid any potential additional disk controller delays caused by larger I/O queue depth. Table 2 lists the average latency (in microseconds) for three YCSB workloads. We observe that F2's average read/write request latency is on par with the best baselines, FASTER and SplinterDB, while achieving 1.4× (3.5×) lower read (write) latency compared to RocksDB.

I/O Amplification. Table 1 lists disk read and write amplification for several as measured by `proc/io`. We observe that F2 reads 2.5–2.9× less bytes from disk compared to SplinterDB for read-intensive YCSB-B and MG-PD, due to the in-memory region of F2's hot log and its read-cache that provide immediate access to hot records. For update-intensive YCSB-A/W, we see that F2 writes 1.3-1.7× fewer bytes to disk, compared to the best-performing system, SplinterDB. This is attributed to the in-place update region of F2's hot log, which avoids writing stale values to disk for write-hot keys, as well as its log-structured design, which aggregates multiple records (or hash chunks), before writing them to disk in larger (4KiB) blocks. Note that even with F2's cold-log index writing hash chunks to disk, F2 writes a comparable number of bytes to disk compared to FASTER, and is more disk-friendly compared to LSM-based systems. Unsurprisingly, page-oriented designs, i.e., LeanStore, KVell, incur high write amplification (i.e., ≥ 30× for 8B key, 100B value records).

Overall, F2 achieves minimal disk wear, i.e., 1.3–3.9× lower write amplification than LSM-based stores, on average.

Thread Scaling. We now evaluate system throughput by varying the number of threads. Figure 11 shows throughput for YCSB-A and YCSB-B. For YCSB-A, we observe that F2 scales linearly from 1 to 6 threads, but between 10-12 threads the scaling flattens out. LeanStore manages to saturate the disk bandwidth with 10 threads; adding more threads do not result in better performance due to inefficient record caching. Both RocksDB and SplinterDB show good scaling, with the latter showing superior ingestion behavior, yet they cannot saturate disk bandwidth. Finally, for YCSB-B, F2, LeanStore and FASTER achieve good thread scaling, and saturate 85–90% of the disk bandwidth with 16 threads.

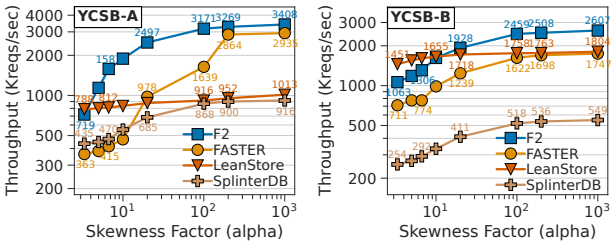

Figure 12: Throughput on YCSB-A (left), YCSB-B (right), when varying Zipf Skewness Factor (α). Axes in log scale.

8.4 Varying the Workload Skewness

F2 targets larger-than-memory workloads with skewed key distribution. To better understand how F2 behaves under different skewed distributions, we experiment with varying the Zipfian skewness factor α, from 3 to 1000 (higher values mean more skewed accesses). When α=100 (YCSB default), 90% of accesses go to 18% of records; for α=10, 90% of accesses go to 33% of records. As before, we use 16 threads, and set the memory budget to 3GiB. Figure 12 shows total system throughput for YCSB-A, YCSB-B (axes in log scale).

For high skewness factors ($\alpha \geq 200$), F2 performs 3.4× (1.4×) better for YCSB-A (YCSB-B) compared to LeanStore, and matches or outperforms FASTER (e.g., by 1.5× for YCSB-B), due to its effective physical separation of hot and cold records, minimizing compaction and user-related disk operations. As we decrease workload skewness ($\alpha \leq 50$), F2's performance gracefully degrades, as the hot set now spills over to disk (and subsequently to cold-log for $\alpha \leq 20$). For update-intensive YCSB-A, F2's fast ingestion capability and efficient lookup-based compaction manage to retain superior performance, even for less-skewed workload (i.e., $\alpha = 3$). For read-intensive YCSB-B, F2's performance degrades, as for more requests now F2 needs to issue two additional I/O (i.e., cold-log resident records). Yet, even with its moderately-sized read-cache (i.e., 512MiB), F2 outperforms the original FASTER by almost 2×.

8.5 Varying Memory Budgets

F2 aims for high-performance even when deployed on constrained memory environments. Here, we experiment with varying memory budgets, ranging from 750MiB to 7.5GiB (2.5–25% of our 250M dataset), using YCSB-A and YCSB-B. We use 16 threads, and configure each system to adhere with the memory limit (we also impose this limit via Linux `cgroups`). Specifically, for F2 we only change the size of the in-memory region of the hot log based on the available budget, while keeping everything else constant (e.g., read-cache). When operating on the lowest 750MiB budget, we disable the read-cache (to make space for the hot-log index and in-memory hot-log region). For LeanStore/SplinterDB (FASTER) we adjust the size of the in-memory buffer-pool/cache (log) based on memory budget.

Figure 13 shows throughput of the best systems as we increase the available memory budget. Given a minuscule memory budget of 750MiB (2.5% of 30GiB), F2 achieves 36% (83%) of the performance for YCSB-A (YCSB-B) when given 4× more memory, while still performing 1.73× (2.14×) better than the best system on the same budget, LeanStore. When using such small budgets (2.5-5%) on YCSB-A even the hottest records do not fit in-memory, forcing F2 to perform mostly I/O operations. Once we give slightly larger budget of 2.25GiB ($\geq$ 7.5%), F2 sees a performance jump of 2×, as most

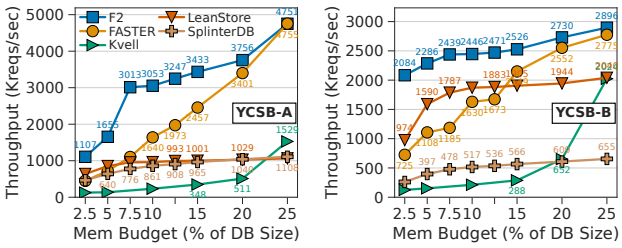

Figure 13: Throughput on memory-scarce environments. Memory budget is 2.5% – 25% of database size (30GiB). Zipfian YCSB-A (left), YCSB-B (right). X-axis in log scale.

hot records are now in-memory, leading to 3.1-4.7× better throughput compared to LeanStore/SplinterDB, while always matching or outperforming FASTER. On YCSB-B, F2 quickly saturates the disk bandwidth at 2.25GiB (7.5%), and any further gains stem from serving in-memory hot records (i.e., due to larger in-memory hot-log size). LeanStore and FASTER, while slow on small budgets, manage to perform well on higher ones, with FASTER even matching F2's performance for budgets of $\geq$ 6GiB (20-25%).

In summary, F2 outperforms the best systems on small-moderate memory budgets and matches their performance on larger ones.

9 RELATED WORK

Memory-efficient Designs. Log-structured Merge (LSM) Trees [41] designs prioritize memory-efficiency, and they can store TBs of data. They have been widely adopted as the storage layer for many popular key-value stores [9, 16, 34, 43]. Researchers have proposed many optimizations for LSM-Trees, including better compaction algorithms [21, 43, 45] (or policies for different tiers [18, 19]), and smaller and more-performant filters [15, 20, 37]. State-of-the-art LSM-based systems, like SplinterDB [15, 16], integrate additional optimizations aimed at improving concurrency and I/O bandwidth utilization, like STBϵ-tree, flush-then-compact compaction, quotient mapplets [15, 42], which further reduce write amplification.

B-Tree based Designs. KVell [31] uses a B-Tree to map every key to a page offset on disk. Disk pages are cached in-memory using a dedicated page cache. In KVell, each thread is responsible for handling requests only for a subset of the key space, eliminating thread contention. LeanStore [30] is optimized for modern NVMe SSDs and multi-core CPUs, and uses a B-Tree alongside an in-memory page buffer manager to support larger-than-memory workloads. Its key idea is pointer swizzling: cached pages are directly accessible via pointers, avoiding the indirection necessary in traditional buffer manager designs. It employs additional techniques (e.g., optimistic locking, contention split) to improve concurrency [1]. Bw-Tree [32] uses log-structured writes using delta records for pages, as discussed in Section 2. Recent work [52] proposes a migration process that clusters hot (cold) records together to create hot (cold) pages, by moving records across pages, improving caching effectiveness.

10 CONCLUSION

This paper describes our journey from the original FASTER library to F2 (for FASTER v2), an evolved key-value store design that targets large skewed workloads. F2 addresses the limitations of existing systems that prevent them from serving such workloads effectively. F2 is open-sourced and available as part of the FASTER project.

REFERENCES

[1] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. The evolution of leanstore. In *BTW 2023*, pages 259–281. Gesellschaft für Informatik eV, 2023.

[2] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. Microsoft cep server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558–1561, August 2009.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 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*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[4] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[5] Azure Stream Analytics. https://azure.microsoft.com/en-us/products/stream-analytics, February 2025.

[6] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8):1591–1604, 2022.

[7] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: efficient execution of serverless workflows. *The VLDB Journal*, 34(2):25, 2025.

[8] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: Semantics for stateful serverless. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[9] Zhichao Cao and Siying Dong. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.

[10] Badrish Chandramouli, Jonathan Goldstein, and Songyun Duan. Temporal analytics on big data for web advertising. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, page 90–101, USA, 2012. IEEE Computer Society.

[11] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[12] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: an embedded concurrent key-value store for state management. *Proceedings of the VLDB Endowment*, 11(12):1930–1933, 2018.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[14] Control Group v2. https://docs.kernel.org/admin-guide/cgroup-v2.html, October 2023.

[15] Alex Conway, Martín Farach-Colton, and Rob Johnson. Splinterdb and maplets: Improving the tradeoffs in key-value store compaction policy. *Proc. ACM Manag. Data*, 1(1), may 2023.

[16] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63. USENIX Association, July 2020.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[18] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 505–520, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Niv Dayan and Moshe Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 365–378, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. Spooky: granulating lsm-tree compactions correctly. *Proc. VLDB Endow.*, 15(11):3071–3084, July 2022.

[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 25–36, New York, NY, USA, 2011. Association for Computing Machinery.

[23] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment*, 6(14):1942–1953, 2013.

[24] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), October 2021.

[25] 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. The design and operation of cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1–14, USA, 2019. USENIX Association.

[26] FASTER: Fast persistent recoverable log and key-value store + cache. https://github.com/microsoft/FASTER, October 2023.

[27] Gabriel Haas and Viktor Leis. What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines. *Proc. VLDB Endow.*, 16(9):2090–2102, May 2023.

[28] Andy Huynh, Harshal A Chaudhari, Evimaria Terzi, and Manos Athanassoulis. Endure: a robust tuning paradigm for lsm trees under workload uncertainty. *arXiv preprint arXiv:2110.13801*, 2021.

[29] Improving Point-Lookup Using Data Block Hash Index. https://rocksdb.org/blog/2018/08/23/data-block-hash-index.html, August 2018.

[30] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196. IEEE, 2018.

[31] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[32] Justin Levandoski, David Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, April 2013.

[33] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 26–37, 2013.

[34] LevelDB. https://github.com/google/leveldb, October 2023.

[35] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 1–13, New York, NY, USA, 2011. Association for Computing Machinery.

[36] Chen Luo and Michael J. Carey. Lsm-based storage techniques: A survey. *The VLDB Journal*, 29(1):393–418, jul 2019.

[37] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2071–2086, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Lin Ma, Joy Arulraj, Sam Zhao, Andrew Pavlo, Subramanya R. Dulloor, Michael J. Giardino, Jeff Parkhurst, Jason L. Gardner, Kshitij Doshi, and Stanley Zdonik. Larger-than-memory data management on modern storage hardware for in-memory oltp database systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 12, USA, 2012. USENIX Association.

[40] NuGet Gallery. https://nuget.org/, July 2025.

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

[42] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 775–787, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[44] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, october 2023.

[45] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. Compactionary: A dictionary for lsm compactions. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2429–2432, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*, volume 2. Prentice Hall Englewood Cliffs, 1997.

[47] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[48] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[49] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. Treeline: An update-in-place key-value store for modern storage. *Proc. VLDB Endow.*, 16(1):99–112, sep 2022.

[50] Jinghuan Yu, Sam H. Noh, Young-ri Choi, and Chun Jason Xue. Adoc: automatically harmonizing dataflow between components in log-structured key-value stores for improved performance. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST'23, USA, 2023. USENIX Association.

[51] Huanchen Zhang. Memory-efficient search trees for database management systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 9, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. Two is better than one: The case for 2-tree for skewed data sets. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, Online Proceedings*, 2023.