FASTER: An Embedded Concurrent Key-Value Store for State Management[*]

Badrish Chandramouli[‡], Guna Prasaad[◇†], Donald Kossmann[‡], Justin Levandoski[‡], James Hunter[‡], Mike Barnett[‡]

[‡]Microsoft Research [◇]University of Washington

badrishc@microsoft.com, guna@cs.washington.edu, {donaldk, justin.levandoski, jahunter, mbarnett}@microsoft.com

ABSTRACT

Over the last decade, there has been a tremendous growth in data-intensive applications and services in the cloud. Data is created on a variety of edge sources such as devices, and is processed by cloud applications to gain insights or make decisions. These applications are typically update intensive and involve a large amount of state beyond what can fit in main memory. However, they display significant temporal locality in their access pattern. We demonstrate FASTER, a new key-value store that combines a latch-free concurrent hash index with a *hybrid log*: a concurrent log-structured record store that spans main memory and storage, while supporting fast in-place updates in memory. FASTER achieves up to orders-of-magnitude better throughput than systems deployed widely today. It is built as an embedded high-level language component using dynamic code generation, and can work with any storage back-end such as local SSD or cloud storage. Our demonstration focuses on: (1) the ease with which cloud applications and state stores can deeply integrate state management into their high-level language logic at low overhead; and (2) the innovative system design and the resulting high performance, adaptability to varying memory capacities, durability, and natural caching properties of our system.

PVLDB Reference Format:
Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: An Embedded Concurrent Key-Value Store for State Management. *PVLDB*, 11 (12): 1930-1933, 2018.
DOI: https://doi.org/10.14778/3229863.3236227

1. INTRODUCTION

A variety of data is created on edge sources such as Internet-of-Things devices, mobile applications, browsers, and servers today. This data is processed by applications and services in the cloud to gain insights. Processing may include ad-hoc analysis of collected data in batches (e.g., in Hadoop and Spark), or realtime monitoring and processing as it arrives (e.g., in Streaming Dataflows [2, 3] and Actor-based Application Frameworks [1]). State management is a critical component in all such processing needs. Web applications may also issue a mix of record inserts, updates, and reads, usually supported via caching object stores such as Redis and Memcached.

State management is a difficult challenge for such applications and services, and has several unique characteristics:

- ***Large State Size***: The amount of state accessed by some applications can be very large, far exceeding the capacity of main memory. For example, a targeted search ads provider may maintain per-user, per-ad and clickthrough-rate statistics for billions of users. Even when it fits in memory, it is often cheaper [6] to retain infrequently accessed state on secondary storage.
- ***Update Intensity***: While reads and inserts are common, there are applications with significant update traffic. For example, a monitoring application receiving millions of CPU readings every second from sensors and devices may need to update a per-device aggregate for each reading.
- ***Temporal Locality***: Even though billions of state objects may be alive at any given point, only a small fraction is typically "hot" and accessed or updated frequently with a strong temporal locality. For instance, a search engine that tracks per-user statistics (averaged over one week) may have a billion users "alive" in the system, but only have a million users actively surfing at a given instant. Further, the hot set may drift over time: in our example, as users start and stop browsing sessions.
- ***Point Operations***: Given that state consists of a large number of independent objects that are inserted, updated, and queried, a system tuned for (hash-based) point operations is often sufficient. If range queries are infrequent, they can be served with simple workarounds such as indexing histograms of key ranges.

1.1 Existing Solutions

A simple solution adopted by many systems is to partition the state across multiple machines, and use pure in-memory data structures such as the Intel TBB Hash Map [11], that are optimized for concurrency and support *in-place updates* – where data is modified at its current memory location without creating a new copy elsewhere – to achieve high performance. However, the overall solution is expensive, often severely under-utilizes the resources on each machine, and makes failure recovery complicated.

Key-value stores are a popular alternative for state management. A key-value store is designed to handle *larger-than-memory* data and support failure recovery by storing data on secondary storage. Many such key-value stores [5, 10, 9] have been proposed in the past. However, these systems are usually optimized for blind updates, reads, and range scans, rather than point operations and *read-modify-write* (*RMW*) updates such as per-key aggregates, which are prevalent in our target applications. Hence, these systems do not scale to more than a few million updates per second, even when the hot-set fits entirely in main memory. Caching systems such as Re-

[†]Work performed during internship at Microsoft.
[*]Project website: `http://aka.ms/FASTER`

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236227

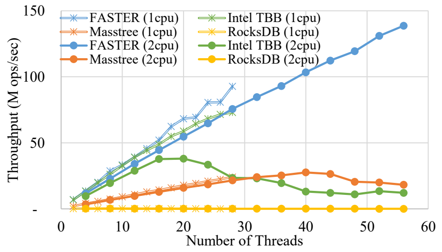


Figure 1: Throughput comparison.

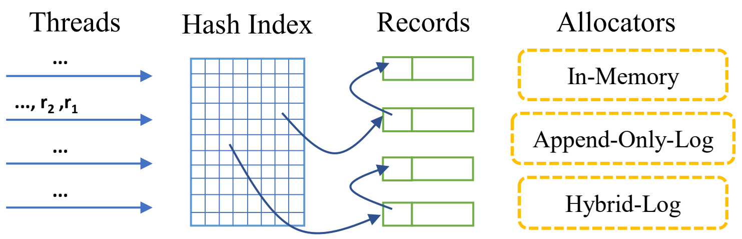


Figure 2: Overall FASTER architecture.

dis and Memcached are optimized for point operations, but are slow and depend on an external system such as a database or key-value store for storage and/or failure recovery. The combination of concurrency, in-place updates (in memory), and ability to handle data larger than memory is important for efficient state management in our target applications; but these features are not simultaneously met by existing systems.

1.2 Introducing FASTER

We have built a new concurrent key-value store called FASTER, designed to serve applications that involve update-intensive state management. The FASTER interface supports, in addition to reads, two types of state updates seen in practice: *blind updates*, where an old value is replaced by a new value blindly; and *read-modify-writes* (*RMWs*), where the value is atomically updated based on the current value and an input (optional). RMW updates enable us to support *partial* updates (e.g., updating a single field in the value) as well as mergeable incremental aggregates such as sum and count.

FASTER supports data larger than memory. When the data fits in memory, it can process 100s of millions of ops/sec. Fig. 1 shows throughput for the YCSB-A workload with RMW operations, using a Zipf key distribution with 8-byte keys and payloads, as compared to pure in-memory systems and key-value stores capable of handling larger data (see our paper [4] for details). FASTER scales well, and outperforms systems in both categories. Retaining high performance while supporting data larger than memory required a careful design and architecture. We propose to demonstrate the FASTER system and several of its novel features:

- FASTER uses a new epoch-based synchronization framework that facilitates lazy propagation of global changes to all threads via trigger actions. We will highlight the use of this framework to simplify the design of several system components.
- FASTER uses a new hash index design that is concurrent, latch-free, resizable, and cache-friendly. It can work with a variety of *memory allocators* for storing records, including in-memory allocators for operating in a pure in-memory setting, and log-structured memory allocators [12, 8] for handling data larger than memory.
- Log-structured allocators are based on the *read-copy-update* strategy, in which updates to a record are made on a new copy on the log. We find that such a design severely limits throughput and scalability in FASTER. As noted earlier, in-place updates are critical for reaching our target performance. Instead, FASTER uses `HybridLog`: a new hybrid log that seamlessly combines in-place updates with a traditional append-only log. The hybrid log organization allows FASTER to perform in-place updates of "hot" records and use read-copy-updates for colder records. Further, it acts as an efficient cache by shaping what resides in memory without any per-record or per-page statistics. Maintaining `HybridLog` concurrently and efficiently required us to solve novel technical challenges.
- FASTER supports durability, and can recover to a recent consistent point after failure. Recovery in FASTER is based on fuzzy index snapshots and periodic consistent points on `HybridLog`.

Implemented as a high-level-language component in C# using dynamic code generation, FASTER is easy to integrate and use in an application, and avoids the overhead of the typical socket-based interfaces of systems such as Redis and Memcached. We have also ported FASTER to C++ for use in other scenarios. FASTER blurs the line between traditional key-value stores and update-only "state stores" used in streaming systems (e.g., the Spark State Store [2]).

In this paper, we overview the system and describe our demonstration, and refer readers to our research paper [4] for the technical details. In particular, our demonstration focuses on: (1) the ease with which cloud applications and state stores can deeply integrate state management into their high-level language logic at low overhead; and (2) our innovative system design and the resulting high performance, adaptability to varying memory capacities, durability, and natural caching properties of our system.

2. SYSTEM OVERVIEW

Fig. 2 shows the overall architecture of FASTER. Threads perform operations on FASTER using epoch protection (Sec. 3.1) to manage concurrency. FASTER consists of a *hash index* that holds pointers to key-value records and a *record allocator* that allocates and manages individual records. The index (Sec. 3.2) provides efficient hash-based access to hash buckets, which are cache-line-sized arrays of hash bucket entries. Each entry includes some metadata and an address provided by a record allocator. The record allocator stores and manages individual records. Hash collisions that are not resolved at the index level are handled by organizing records as a linked-list. Each record consists of a record header, key, and value, and points to the previous record in the linked-list. While FASTER can work with standard in-memory and log-structured allocators, we propose the `HybridLog` allocator (Sec. 3.3) that combines log-structuring with in-place updates to handle larger-than-memory data without losing performance when the hot data fits in memory.

User Interface

In addition to the standard `get-put` interface supported by key-value stores, FASTER supports advanced user-defined updates. We use dynamic code generation to integrate the update logic provided as user-defined delegates during compile time into the store, resulting in a highly efficient store with native support for advanced updates. The generated FASTER run-time interface consists of the following operations:

- `Read`: Read the value corresponding to a key.
- `Upsert`: Blindly replace the value corresponding to a key with a new value. Insert as new, if the key does not exist.
- `RMW`: Update the value of a key based on the existing value and an input (optional) using the update logic provided by the user during compile-time. We call this a *Read-Modify-Write (RMW)*

operation. The user also provides an initial value for the update, which is used when a key does not exist in the store.

- `Delete`: Delete a key from the store.

3. MAJOR SYSTEM COMPONENTS

We next describe the three main components of FASTER: an epoch protection framework that forms our threading model, our concurrent in-memory hash index, and our `HybridLog` record allocator. Together, they form the core of the FASTER key-value store that is the focus of our demonstration.

3.1 Epoch Protection Framework

FASTER threads perform operations independently with no synchronization most of the time. At the same time, they need to agree on a common mechanism to synchronize on shared system state. To achieve these goals, we extend the idea of *multi-threaded epoch protection* [7]. Briefly, the system has a global current epoch value that is periodically incremented. Threads hold on to a particular epoch to perform a set of operations. There is a global notion of a *safe* epoch, such that no threads are active in that epoch or earlier. While systems like the Bw-Tree [9] use epochs for specific purposes such as garbage collection, we extend it to a generic framework by adding the ability to associate callbacks, called *trigger actions*, when an epoch becomes safe.

Epochs with trigger actions can be used to simplify lazy synchronization in parallel systems. Consider a canonical example, where a function `active-now` must be invoked when a shared variable `status` is updated to `active`. A thread updates `status` to `active` atomically and bumps the epoch with `active-now` as the trigger action for the current epoch. Not all threads will observe this change in `status` immediately. However, all of them are guaranteed to have observed it when they refresh their epochs (due to sequential memory consistency using memory fences). Thus, `active-now` will be invoked only after all threads see the status to be active and hence is safe. We use the epoch framework in FASTER to coordinate system operations such as memory-safe garbage collection, index resizing, circular buffer maintenance, page flushing, shared log page boundary maintenance, and checkpointing for recovery. This allows us to provide FASTER threads with unrestricted latch-free access to shared memory locations in short bursts for user operations such as reads and updates.

3.2 Hash Index

The FASTER index is a cache-aligned array of 2^k *hash buckets*. Each hash bucket consists of seven 8-byte hash bucket *entries* and one 8-byte entry that serves as an overflow bucket pointer. The 8-byte entries allow us to operate latch-free using compare-and-swap operations. Each entry consists of a 48-bit address (provided by the record allocator) and a 15-bit tag, which is used to increase our effective hashing resolution to $k+15$ bits. Note that the index does not store keys; this keeps the index small and allows us to retain it entirely in main memory. Keys whose hash values map to the same array offset and tag are organized as a reverse linked-list pointed to by the index entry.

Reads and deletes are straightforward latch-free operations, but inserts into the index are trickier to carry out in a latch-free manner while preserving the invariant of exactly on index entry per bucket and tag. We use a two-phase protocol along with a *tentative bit* to solve the problem. The index also supports resizing on-the-fly: we use our epoch framework with a sequence of phases to achieve latch-free resizing without blocking, while retaining performance in the common case where the index size is stable. Our paper [4] covers the details on our solutions to these challenges.

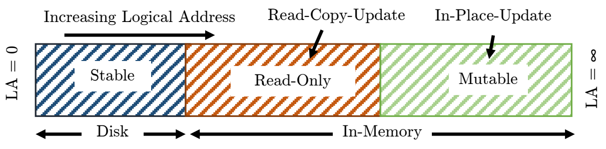


Figure 3: Logical address space in `HybridLog`.

3.3 Hybrid Log Allocator

Log-structured allocators are commonly used for organizing data that spills to secondary storage, but we find that such an allocator is unable to scale beyond a few million operations per second, due to the overhead of read-copy-update and the pressure on the tail of the log. We address this challenge using `HybridLog`, a novel data structure that combines in-place updates (in memory) and log-structured organization (on disk), while providing latch-free concurrent access to records. `HybridLog` spans memory and secondary storage, where the in-memory portion acts as a cache for hot records and adapts to a changing hot set.

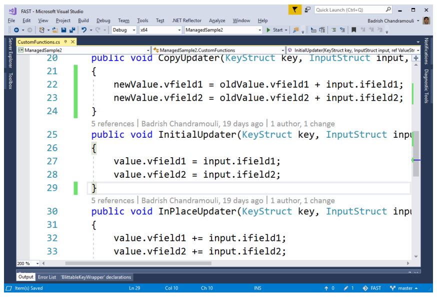
We first define a global logical address space that spans main memory and secondary storage. The logical address space is divided into three contiguous regions: (1) *stable region* (2) *read-only region* and (3) *mutable region* as shown in Fig. 3. The stable region is the part of logical address space that is on secondary storage. The in-memory portion is composed of read-only and mutable regions. Records in the mutable region can be modified in-place, while records in the read-only region cannot. We use read-copy-update to modify a record in the read-only region: a new copy is created at the tail (in the mutable region) and then updated. Further updates to such a record are performed in-place, as long as it stays in the mutable region. This organization provides good caching shaping behavior without requiring fine-grained statistics, as the read-only region provides a second chance for hot records to go back to the tail before they are evicted.

The regions of `HybridLog` are demarcated using three offsets maintained by the system: (1) the *head offset*, which tracks the lowest logical address that is available in memory; (2) the *read-only offset*, which tracks the boundary between the read-only and mutable regions in memory; and (3) the *tail offset*, which points to the next free address at the tail of the log.

Threads use an atomic fetch-and-add operation on the tail offset for new allocations. Since updates and reads are very frequent, locking the head and read-only offsets to determine how to process these operations would cause severe performance degradation. Instead, we leverage epoch protection and let threads use a cached value of these offsets, updated at epoch boundaries. Thus, each thread has its own view of `HybridLog` regions, that may diverge from the true system view. This divergence leads to subtle concurrency issues. For example, consider two threads issuing RMW operations. The first thread may consider a logical address to be in the read-only region and therefore copy it to the tail. In parallel, the second thread may view the same address to be in the mutable region, and update it in place, leading to a *lost update* by the second thread. We identify this case by defining a *safe read-only offset* that is guaranteed to have been seen by all threads. Updates in the *fuzzy region* between the safe and true read-only offsets are handled carefully by delaying the update until it is safe to perform. Our paper [4] has the details on these and other concurrency challenges.

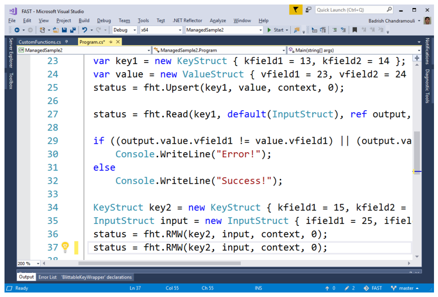
4. DEMONSTRATION WALKTHROUGH

We structure the demonstration of the C# version of FASTER in four parts. The C++ version of FASTER will also be made available at the demo, for interested audience members.


FAST - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test .NET Reflector Analyze Window Help
Debug x64 ManagedSample2 Start
CustomFunctions.cs
ManagedSample2
ManagedSample2.CustomFunctions
InitialUpdater(KeyStruct key, InputStruct input, ref ValueStr
Output Error List 'BlittableKeyWrapper' declarations
Item(s) Saved Ln 29 Col 10 Ch 10 INS FAST master

```
public void CopyUpdater(KeyStruct key, InputStruct input,
{
    newValue.vfield1 = oldValue.vfield1 + input.ifield1;
    newValue.vfield2 = oldValue.vfield2 + input.ifield2;
}
5 references | Badrish Chandramouli, 19 days ago | 1 author, 1 change
public void InitialUpdater(KeyStruct key, InputStruct inpu
{
    value.vfield1 = input.ifield1;
    value.vfield2 = input.ifield2;
}
5 references | Badrish Chandramouli, 19 days ago | 1 author, 1 change
public void InPlaceUpdater(KeyStruct key, InputStruct inpu
{
    value.vfield1 += input.ifield1;
    value.vfield2 += input.ifield2;
```

(a) Compile-time callback definitions.


FAST - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test .NET Reflector Analyze Window Help
Debug x64 ManagedSample2 Start
CustomFunctions.cs Program.cs*
ManagedSample2
ManagedSample2.Program
Main(string[] args)
Output Error List 'BlittableKeyWrapper' declarations
Ready Ln 37 Col 55 Ch 55 INS FAST master

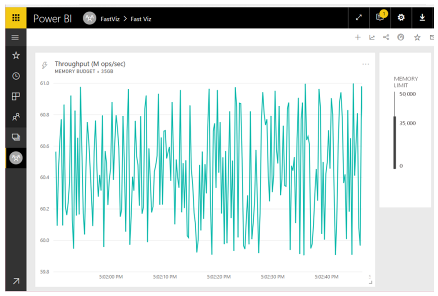
```
var key1 = new KeyStruct { kfield1 = 13, kfield2 = 14 };
var value = new ValueStruct { vfield1 = 23, vfield2 = 24
status = fht.Upsert(key1, value, context, 0);

status = fht.Read(key1, default(InputStruct), ref output,

if ((output.value.vfield1 != value.vfield1) || (output.va
    Console.WriteLine("Error!");
else
    Console.WriteLine("Success!");

KeyStruct key2 = new KeyStruct { kfield1 = 15, kfield2 =
InputStruct input = new InputStruct { ifield1 = 25, ifiel
status = fht.RMW(key2, input, context, 0);
status = fht.RMW(key2, input, context, 0);
```

(b) Run-time interface and use.



(c) Run-time performance dashboard.

Figure 4: Screenshots of aspects of the system that will be demonstrated to visitors.

4.1 Compile-Time Definitions

We will first demonstrate the FASTER *compile-time interface*, depicted in Fig. 4a, which accepts user-defined read and update logic in the form of side-effect-free callback functions. Users can either choose to let the system handle record-level concurrency, or provide advanced logic for single- and multi-threaded access. For instance, when used as a partitioned state store, users can avoid any synchronization when accessing and updating records.

4.2 Run-Time Interface

Next, we will demonstrate the *run-time interface*, depicted in Fig. 4b, and summarized below. This interface is dynamically code-generated for an application, for the specified read and update logic and the data-types for keys and values.

```
Status Read(Key*, Input*, Output*, Context*, long);
Status Upsert(Key*, Value*, Context*, long);
Status RMW(Key*, Input*, Context*, long);
Status Delete(Key*, Context*, long);
```

All calls include a `long` argument for a sequence number used in recovery. For a read, the user provides an input that is used by the read callback function, to read the appropriate field into the output. With RMW, the input is used along with the old value, by the update callback function, to compute the new value. Contexts are used when operations go asynchronous, e.g., due to I/O. The use of this interface will be demonstrated in two settings: (1) as a key-value store used in a stand-alone cloud application that manages per-device state; and (2) as the state store of an aggregation operator in a streaming dataflow pipeline.

4.3 Run-Time Behavior and Performance

We will demonstrate the run-time behavior and performance (see Fig. 4c) of FASTER using two workload scenarios.
1) Key-Value Store: We will run a benchmark YCSB workload with varying percentage of reads, blind updates, and RMW operations. This benchmark will show how FASTER performs in terms of throughput, storage read and write IOPS, and memory utilization. We will cover two cases: (a) when data fits in memory; and (b) as the memory budget is varied, as depicted in Fig. 4c.
2) Streaming State Store: We will demonstrate FASTER being used as the state store for a streaming aggregate query (e.g., per-key sum). The query operates over a key space that may exceed main memory. The operator may generate its own output, or our log may be flushed on demand as the query output.

4.4 Recovery

FASTER employs a low-overhead, non-blocking recovery solution. We will demonstrate the durability of FASTER by killing the computation midway and restarting it, to show that FASTER can recover quickly to a consistent point, and continue processing at high throughput after a failure, using a recent index and log snapshot. The key demonstration here is the light-weight persistence enabled by FASTER as a high-level language component.

5. CONCLUSIONS

We demonstrate FASTER, a new concurrent key-value store for managing application state. FASTER is based on a latch-free index that works with `HybridLog`, a concurrent log that combines an in-place updatable region with log-structuring, to optimize for the hot set without any fine-grained caching statistics. FASTER achieves better throughput – up to 160M operations per second on one machine – than systems deployed widely today, and outperforms in-memory data structures when the workload fits in memory. Our demonstration focuses on: (1) the ease with which cloud applications and state stores can integrate state management into their high-level language logic; and (2) the innovative system design and the resulting high performance, adaptability to varying memory capacities, durability, and caching properties of our system.

6. REFERENCES

[1] Orleans - Distributed Virtual Actor Model for .NET. `https://github.com/dotnet/orleans`, 2018. [Online; accessed 23-Jul-2018].

[2] Apache Software Foundation. Spark State Store: A new framework for state management for computing streaming aggregates. `https://issues.apache.org/jira/browse/SPARK-13809`, 2017. [Online; accessed 23-Jul-2018].

[3] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[4] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 275–290, New York, NY, USA, 2018. ACM.

[5] Facebook Open Source. RocksDB. `http://rocksdb.org/`, 2017. [Online; accessed 23-Jul-2018].

[6] J. Gray and G. Graefe. The Five-minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *SIGMOD Rec.*, 26(4):63–68, Dec. 1997.

[7] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, Sept. 1980.

[8] J. Levandoski, D. Lomet, and S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10):877–888, 2013.

[9] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 302–313, Washington, DC, USA, 2013. IEEE Computer Society.

[10] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[11] Intel Corporation. Intel Threading Building Blocks. `https://www.threadingbuildingblocks.org/`, 2017. [Online; accessed 23-Jul-2018].

[12] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.