DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Incremental Snapshotting in Transactional Dataflow SFaaS Systems

Author:
Nikolaos GAVALAS

Supervisor:
Dr. Asterios KATSIFODIMOS

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

Student number:	5514762	
Thesis committee:	Dr. A. Katsifodimos,	TU Delft, supervisor
	Dr. G. Gousios,	TU Delft
	PhD Candidate K. Psarakis,	TU Delft, daily supervisor

An electronic version of this thesis is available at
https://repository.tudelft.nl/.

June 19, 2023

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Incremental Snapshotting in Transactional Dataflow SFaaS Systems

by Nikolaos GAVALAS

The adoption of the serverless architecture and the Function-as-a-Service model has significantly increased in recent years, with more enterprises migrating their software and hardware to the cloud. However, most applications require state management, leading to the use of external databases. To alleviate the burden of state management, there are systems known as SFaaS (Stateful Function-as-a-Service) that provide stateful functions. Despite their benefits, SFaaS systems still face challenges such as the need for transactional logic. Stateful streaming dataflow engines offer promising capabilities for implementing transactional SFaaS systems due to their exactly-once message delivery guarantees and global state management. Key-value stores serve as embedded databases in this architecture, making it crucial to carefully evaluate available options for suitable types of key-value stores.

This work focuses on the implementation and evaluation of three distinct types of log-structured key-value stores within the context of serving as state-management backends for transactional dataflow systems. A key aspect of our implementations is the incorporation of efficient *incremental snapshotting* functionality. We explore the performance and suitability of these key-value stores in managing state and supporting transactional operations in dataflow systems.

Acknowledgements

I would like to express my gratitude to my supervisor, Asterios Katsifodimos, for providing me with the opportunity to demonstrate my abilities, trusting me, and offering invaluable feedback throughout the journey of my thesis.

Additionally, I extend my sincere appreciation and gratitude to my daily-supervisor, Kyriakos Psarakis, whose unwavering support, guidance and continuous feedback have been instrumental in making this work possible. I am also grateful to the members of the committee for generously dedicating their time to evaluate my work.

Lastly, I would like to thank my family and friends for their support throughout this endeavor.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud Computing has seen a dramatic rise in its adoption the recent years, with an increasing number of enterprises migrating their software and hardware to the cloud, and this trend is only expected to continue [Gens et al., 2019]. Historically, this shift towards managed infrastructure has been arguably inevitable, because with cloud computing the cost per unit of computation is minimized [Castro et al., 2019]. The drive for increased efficiency in computation has culminated in the emergence of the *serverless* architecture [Rajan, 2018].

In the serverless cloud computing execution model, applications are being developed as collections of fine-grained event-driven and stateless units of computation called *cloud functions*. Cloud providers offer the execution of serverless functions as a paid service, known as *Function-as-a-Service* or *FaaS* [Shafiei, Khonsari, and Mousavi, 2019].

While FaaS offerings prioritize scalability by being stateless, most applications require some form of state management, resulting in developers resorting to external databases for their applications' state-keeping. Several recent works have aimed to alleviate the burden of state management from application developers [Bykov et al., 2011; Burckhardt et al., 2021; Zhang et al., 2020] by enabling the transparent management of application state through external databases, thereby providing *stateful functions*, or *SFaaS*.

SFaaS systems ease the development of stateful applications, but they are not a panacea per se. Any programmer that develops distributed applications will eventually have to deal with fundamental potential issues such as network partitioning, system failures and the Byzantine generals messaging problem [Lamport, Shostak, and Pease, 2019]. These problems become especially hard to deal with when the application level requires implementing *transactional* logic, as transactions require extra guarantees. Transactions are sets of operations that must adhere to the ACID principles - Atomicity, Consistency, Isolation, and Durability [Gray and Reuter, 1992].

Consequently, developers often find themselves intermixing business logic with consistency checks, rollbacks, snapshots, and timeouts, resulting in systems that are highly intricate to maintain and prone to failures. This highlights the need for an intermediary layer that abstracts the distributed fault-tolerance logic and provides application developers with specific guarantees, both at the state-management level and the transactional level, if feasible.

SFaaS systems built on top of *stateful streaming dataflow engines* such as Apache Flink StateFun [Carbone et al., 2015] make excellent candidates for implementing *transactional SFaaS* systems, primarily for two reasons [Heus et al., 2022]:

1. They offer *exactly-once* message delivery semantics, eliminating the need for identifying lost messages and resending them, and also guarantee the message delivery order - the communication channels between the distributed components are FIFO.

2. They fully manage the system's global distributed state by periodically creating consistent snapshots and recovering them upon failures. This is especially important for implementing transactions, since for failed transactions there needs to be a rollback mechanism to guarantee the Atomicity property.

Dataflow SFaaS systems are comprised of multiple worker processes, with each of them keeping a partition of the global state locally [Carbone et al., 2015]. The state is represented as key-value pairs, making key-value stores an ideal choice as embedded databases for this task.

As the key-value store is a critical component of this architecture, it is essential to carefully evaluate the available options of suitable types of key-value stores and motivate our selection. Towards this end, in this study, we implement three different kinds of key-value stores, evaluate their performance within transactional dataflow systems and conduct a comprehensive comparative analysis among them.

1.1 Design Requirements

In a (transactional) dataflow SFaaS system, the key-value stores need to have specific properties to be considered suitable. These properties, extending those mentioned in the work of Chandramouli et al., 2018, are:

1. *Incremental snapshots* [Carbone et al., 2017]. When the dataflow engine requests a worker to create a snapshot of its state, the state backend (the key-value store) will dump the state and save it. As this process happens many times during the execution of a workflow, to ensure fault-tolerance and fast state recovery, it is imperative that it is done efficiently, building on previous snapshots.

   The naive solution is to save the whole state every time, but if there is a way to only save the updates on the state at each step, incrementally, it would definitely be more efficient. However, saving only the updates on each step, would make recovery very slow, as the state would need to be rebuilt from the very beginning in case of a system failure. In this work, we propose a solution that combines *fast incremental snapshots with low recovery times*.

2. *State recovery to a previous version from previous snapshots (rollback)* [Chandy and Lamport, 1985; Carbone et al., 2017]. Upon execution, the dataflow coordinator process may request the workers to restore some previous version of their state, so that the system can go back to some consistent global state and "replay" events to recover from some failure.

3. *Larger-than-memory data (spill-to-disk)*. When dealing with large volumes of data, it is expected that during execution the state will exceed in size the amount that can be stored in memory. Hence, it is essential that the key-value store employs persistent storage when necessary to handle states larger than the available memory.

4. *Update-intensity*. In dataflow systems, changes to the state are typically characterized by the volume of updates rather than inserts or deletes. This is particularly evident in workflows that involve data aggregations or analytics, and it holds even more significance in systems that support transactions. Transactional systems often involve frequent operations like value increments. As a result, the state backend needs to be well-suited for update-heavy workloads.

5. *Locality*. In real-world dataflow applications, access to data is rarely uniformly distributed. Keys that are “alive” at any moment may be of many orders of magnitude, but it’s usually a subset of those that are “hot” at some given time, i.e. accessed or updated frequently. The hot set may drift as time passes but the strong temporal locality property is maintained.

6. *Point operations*. A key-value store for our use-case should be optimal for point operations, i.e. operations associated with a single key, as opposed to range operations. Since state updates rarely operate on ranges of keys, we can leverage this knowledge to our advantage.

1.2 Research Questions

At this juncture we can outline the main research questions of this work. The first research question is:

> **RQ1**: Which type or types of key-value stores are more fitting as embedded state stores in the worker processes of transactional dataflow SFaaS systems?

To address this question in alignment with the design requirements outlined in subsection 1.1, our approach involves several steps. Firstly, we will survey and examine existing key-value store designs, considering their suitability for our purposes. Next, we will carefully narrow down our options and provide a compelling rationale for our chosen selections. Subsequently, we will proceed to implement the most promising candidates, and study them in depth, which leads us to the second and third research questions:

> **RQ2**: How do changes in the parameters of each selected type of key-value store affect its performance?

> **RQ3**: In the selected types of key-value stores, which are the trade-offs that determine their operation? In which general use-cases does each of them perform better?

Next, we will proceed with a thorough evaluation of the implemented key-value stores by integrating them into a transactional dataflow system. During this evaluation, we will specifically focus on assessing the efficiency of the incremental snapshotting functionality and its impact. Thus, our fourth research question is formulated as follows:

> **RQ4**: How does the performance of a key-value store that incorporates incremental snapshotting functionality compare to that of a "naive" in-memory key-value store, which captures snapshots of its entire state at each step, in terms of snapshot creation time?

Ultimately, we will be able to address the final research question:

> **RQ5**: Is there a key-value store that clearly stands out as the superior choice for state management?

1.3 Contributions

We summarize this work's contributions in the following points:

1. *Design and implementation of Three Key-Value Stores*: To ensure a fair comparison and level playing field, we have implemented three distinct key-value store implementations. Each implementation adheres to the same programming language and incorporates similar design choices for shared functionality, such as data encoding and data structures. By keeping these aspects consistent, we can isolate the differences in the key-value store logic and facilitate accurate comparisons.
2. *Experimental Analysis*: In order to address the research questions outlined in section 1.2, we have conducted a series of experiments. These experiments focus on analyzing the parameters of each implemented key-value store and exploring the trade-offs inherent in their designs, particularly in terms of resource utilization. By systematically examining these aspects, we aim to gain a deeper understanding of the strengths and weaknesses of each key-value store implementation.
3. *Comprehensive Comparison*: Building upon the experimental analysis, we have conducted a comprehensive comparison among the implemented key-value. This comparison encompasses various factors, including the effectiveness of incremental snapshotting, which plays a vital role in state management. Ultimately, our goal is to determine whether one key-value store emerges as the optimal choice for our specific use case. By thoroughly evaluating the performance and capabilities of each implementation, we aim to provide insights and make informed recommendations for state management in transactional dataflow systems.

1.4 Outline

The rest of the thesis is structured as follows:

Chapter 2 provides a review of the existing literature and related work in the field. It explores previous research, methodologies, and advancements in key-value stores and state management within transactional dataflow systems. This chapter establishes a solid foundation for our own study.

In chapter 3: we delve into comprehensive descriptions of the internal workings of each type of key-value store. We provide in-depth insights into their underlying mechanisms, data structures, and algorithms. Furthermore, we discuss the specific implementation details and design decisions that pertain to each key-value store type. By thoroughly understanding the intricacies of each implementation, we lay the groundwork for subsequent evaluations and comparisons.

Chapter 4 is dedicated to the evaluation of our implemented key-value stores. We conduct a series of benchmarks and comparisons to assess their performance and capabilities. This includes integrating the key-value stores into a transactional dataflow system to simulate real-world usage scenarios. By rigorously evaluating their performance, scalability, and efficiency, we gain valuable insights into the strengths and limitations of each implementation. We discuss the obtained results and analyze the implications they have on state management in transactional dataflow systems.

In the final chapter, 5, we provide a comprehensive summary of our research and findings. We present our conclusions based on the evaluation and comparisons performed. We also address the research questions posed earlier in the thesis and provide insightful answers. Additionally, we discuss potential directions for future research and highlight areas that require further exploration and development.

Chapter 2

Related Work

2.1 Transactional Dataflow Systems

Transactional dataflow systems are a class of distributed systems designed to handle large-scale data processing with transactional guarantees. They provide a programming model that allows developers to write declarative, data-driven computations that automatically handle fault tolerance, scalability, and consistency.

Transactional dataflow SFaaS (Stateful Function-as-a-Service) systems are cloud-based systems that provide a serverless platform for processing large-scale data with transactional guarantees. These systems allow users to write and deploy stateful individual functions or small pieces of code that are triggered in response to events, such as incoming data or scheduled tasks. They are build on top of dataflow systems because they provide fault tolerance, scalability and consistency out-of-the-box.

One of the most prominent transactional dataflow SFaaS system is Apache Flink's [Carbone et al., 2015] StateFun, the architecture of which is shown in figure 2.1 (credited to Heus et al., 2022).

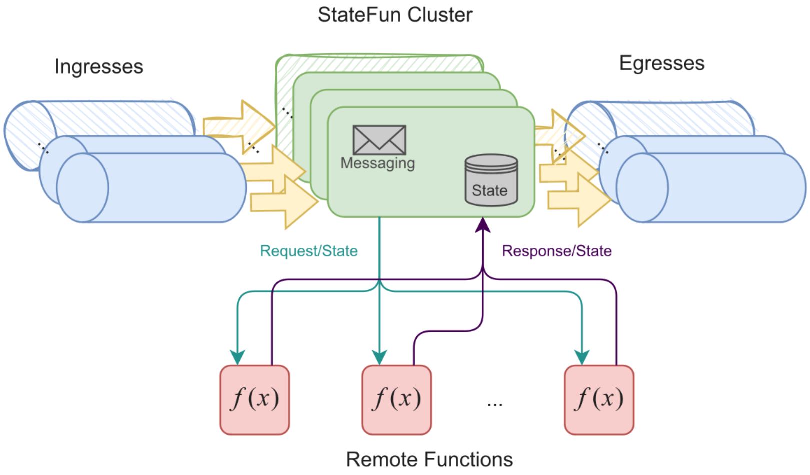


FIGURE 2.1: Architecture of Apache StateFun

Remote functions are executed in the nodes of the StateFun cluster, and each node saves its state into an embedded key-value store, as the state can be modelled effectively by a collection of key-value pairs.

In relation to the current work, this is the model architecture for which we will optimize our key-value stores. More concretely, we assume that the key-value stores are to be used as embedded key-value stores in a similar cluster, and that there exists some reliable remote storage in the cloud to store our snapshots.

2.2 Key-value stores

A key-value store is a type of database that uses a simple key-value data model to store data. In a key-value store, data is represented as a collection of key-value pairs, where each key is a unique identifier that is associated with a corresponding value.

Key-value stores are designed for efficient and fast access to data, making them suitable for use cases where high performance and low latency are critical.

There are various types of key-value stores, each of which is optimized for specific use cases and applications. A fundamental factor that determines the properties of a key-value store is its backend, i.e. the data structures that power it. The main backends for key-value stores are B-Trees, LSM-Trees, and on-disk hash-tables if they store their data on disk, or other tree-based or hash-based data structures if they store their data in memory. Of course, there are also hybrids that combine other types.

2.2.1 Types of key-value store backends

B-Trees

A B-tree is a data structure used to store and organize data in a sorted manner, allowing for efficient search, insertion, and deletion operations [Comer, 1979]. It is a balanced tree structure, meaning that the height of the tree is kept relatively low compared to the number of elements it contains, which in turn ensures fast access and modification times.

The B-tree consists of nodes, each containing a number of keys and pointers to child nodes. The keys are sorted in ascending order within each node, and the pointers are used to traverse the tree and locate the desired key or node. The number of keys and pointers in each node is fixed, and typically determined by the size of a disk block or page. An example of how key-value lookups work in B-Trees can be found in figure 2.2.

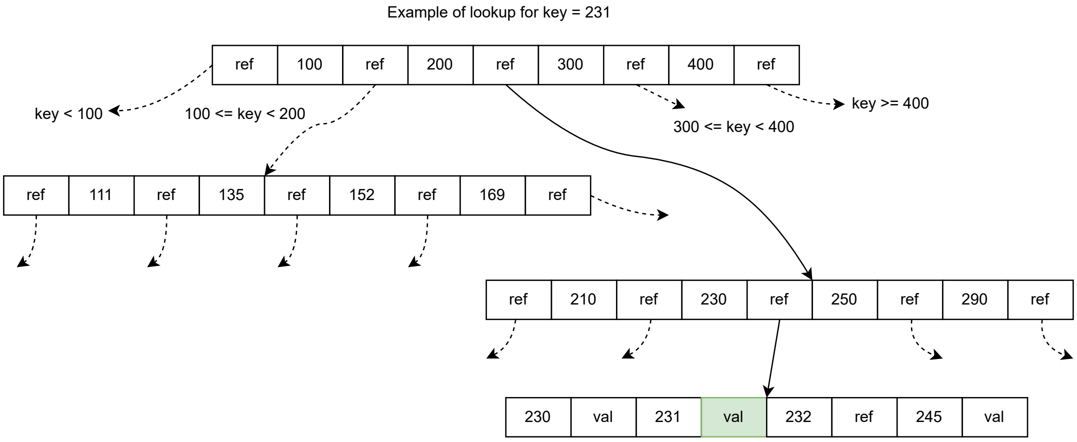


FIGURE 2.2: Key lookup example in a B-Tree.

B-trees are commonly used in database systems (especially relational database systems), file systems, and other applications that require fast and efficient access to large amounts of data stored on disk or in memory. However, the B-tree significantly escalates the I/O costs of transactions, as it necessitates real-time maintenance of the index. Consequently, this results in a considerable rise in the overall system cost, reaching up to a fifty percent increase in I/O operations [O'Neil et al., 1996].

Log-Structured Merge-Trees

The Log-structured Merge-Tree [O'Neil et al., 1996] (or LSM-Tree for short) is another popular data-structure used in modern database systems.

At its core, the LSM-tree consists of two main components: a memory component, often called the memtable, that serves as an in-memory buffer and a series of on-disk components, typically referred to as levels. The levels themselves are comprised of immutable SSTables, short for sorted-string tables, or just "runs". The writes to the LSM-Tree are flushed directly to disk, and the runs are then merged periodically to garbage-collect overwritten records. The LSM-Trees' internals are analyzed in detail in Chapter 3.

In comparison to the B-Trees, LSM-Trees have several advantages (or trade-offs to be more accurate):

- The LSM-trees excel in workloads with heavy write operations (inserts-updates-deletes). Since writes are initially buffered in the memtable and flushed to disk periodically, LSM-trees minimize disk I/O operations, resulting in significantly faster write performance compared to B-trees. B-trees, on the other hand, require immediate disk writes for every update, which can be a performance bottleneck in write-intensive workloads. However, B-Trees typically are more suitable for read-intensive workloads.

- LSM-Trees are usually more space-efficient, leading to less disk space usage. B-Trees often suffer from fragmentation, where deleted or updated entries leave behind empty or partially-filled nodes. LSM-trees consolidate data during the compaction process, eliminating duplicates and reclaiming space, leading to improved space utilization. In addition, data is LSM-Trees can be relatively easily compressed, leading to even more efficient space utilization.

- In storage systems, a phenomenon known as *write amplification* [Hu et al., 2009; Dong et al., 2017] occurs, which can have a detrimental effect on disk durability and performance, especially in SSD drives. Write amplification refers to the situation where a single database write operation triggers multiple physical writes to the disk. This phenomenon is more pronounced in B-trees in comparison to the LSM-trees, because multiple random page writes are needed to update a single value and the index. LSM-Trees are less susceptible to this phenomenon because they write data sequentially, leading to less susceptibility to write amplification. Sequential writes also lead to a performance boost, especially in rotational HDD disks.

- Due to the way LSM-Trees organize their data immutably into levels, they allow for fast recovery, and most importantly for efficient incremental snapshots. *All the recent writes are located in higher levels of the LSM-Tree and therefore when taking a snapshot we can exclude the lower levels if they have been included in a previous snapshot*. With a B-Tree, incremental snapshots would be challenging to achieve because of their in-place updates. We would need to maintain additional data-structures to keep track of what exactly was changed, and make these data-structures persistent as well.

While log-structured storage offers numerous benefits, it does have a drawback related to the compaction process, which can occasionally impact the performance of concurrent read and write operations. As disks have limited bandwidth, allocating a significant portion of it to merging operations can adversely affect data writes. This can result in a slight reduction in throughput and average response time. The impact is typically minimal but in certain cases, particularly at higher percentiles, queries to log-structured storage engines may experience relatively high response times. In such scenarios, B-trees tend to offer more predictable and consistent performance.

Additionally, in B-trees each key exists in precisely one location within the index, unlike log-structured storage engines where multiple copies of the same key may reside in different segments. This characteristic makes B-trees appealing in databases that aim to provide robust transactional semantics. Many relational databases, for instance, implement transaction isolation by applying locks to key ranges. In a B-tree index, these locks can be directly associated with the tree, making it easier to manage and enforce transactional consistency. For our use case, this characteristic is not important, because all transactional logic is handled at higher levels by the transactional dataflow system.

Fractal Trees

Fractal Trees are a type of indexing data structure that are designed to provide high performance and scalability in multi-core environments. They are primarily based on B-Trees.

The key idea behind Fractal Trees is to split the index into a set of smaller indexes, each of which is optimized for a specific data access pattern. This allows the system to scale horizontally across multiple cores and nodes, while also providing high performance for a wide range of workloads.

Like B-Trees, there are good for transactions at the database level, because each key exists in only one copy in the tree. Compared to LSM-Trees, they can offer some advantage in terms of mitigating write amplification [Kuszmaul, 2014], but the advantage is insignificant in leveled many-runs-per-level LSM-Trees (which is the kind of LSM-Tree presented and implemented in Chapter 3).

On-disk hash-tables

On-disk hash-tables, also known as persistent hash-tables, are data structures that allow efficient storage and retrieval of key-value pairs on disk. On-disk hash-tables use hashing algorithms to map keys to specific locations on the disk, enabling fast retrieval of values associated with the keys. The hash-table is typically divided into fixed-size buckets or blocks, each containing a certain number of key-value pairs. One prominent example of a database that uses on-disk hash-tables is the *GNU dbm* project.

In-memory key-value stores

In-memory key-value stores, as the name implies, store and retrieve data entirely in main memory, providing fast and efficient access to key-value pairs. Unlike disk-based storage systems, which store data on hard drives, in-memory key-value stores keep the entire dataset in RAM, eliminating the latency associated with disk I/O operations, allowing for extremely low access times, making them ideal for applications that require high-performance data retrieval, such as caching, real-time analytics, and session management. However, the limited capacity of RAM restricts the size of the dataset that can be stored in-memory, making these stores more suitable for smaller to moderate-sized datasets that can fit within the available memory. A well-known commercial in-memory key-value store is *Redis*.

Hybrids

There are databases that leverage more than one data structure to store and retrieve data. Microsoft's FASTER Chandramouli et al., 2018 for instance uses in-memory components with log-structured on-disk storage to combine the best between two worlds. We analyze FASTER in detail in chapter 3, as it is one of the implemented stores.

2.2.2 Key-value stores in dataflow systems

The most prominent key-value store used as embedded key-value store in distributed streaming/dataflow systems is *RocksDB*, an LSM-Tree-based store. It is used in Apache Spark Structured Streaming [Armbrust et al., 2018], Apache Flink [Carbone et al., 2015], *Apache Kafka* and Apache Samza [Noghabi et al., 2017]. In the work of Kalavri and Liagouris, 2020, the authors have also integrated FASTER in a dataflow system seamlessly.

2.3 Incremental Snapshots

Snapshots play a vital role in distributed systems, as they enable the creation of a consistent snapshot representing the global state of the system. However, achieving this consistency is challenging due to the absence of globally shared memory and a synchronized global clock.

Extensive research has been conducted on snapshotting algorithms, as documented in literature such as the work by Chandy and Lamport, 1985. While these algorithms have received significant attention, there has been limited exploration of efficient incremental snapshots, which aim to leverage previous snapshots to avoid redundant work.

The problem of incremental snapshots can be seen as an extension of the broader challenge of distributed state synchronization, which focuses on maintaining consistency across distributed systems. In the next sections we present the approach of some commercial systems to this problem, as well as some data structures commonly used for state synchronization.

2.3.1 Incremental Snapshots in Distributed Systems

Apache Flink [Carbone et al., 2015] leverages the properties of the log-structured storage and the concept of *delta maps* (see section 2.3.2) [Carbone et al., 2017] for incremental snapshots, although as for the time this related work was published, it hasn't been implemented. This approach is the one we generally follow in implementing the incremental snapshotting functionality in the key-value stores in Chapter 3.

In the work of Fraser, 2009, the author introduces the concept of *Differential Synchronization* for synchronizing changes to a single document edited by multiple users in parallel. The approach is interesting but too high-level for our use-case.

2.3.2 Data structures for efficient state synchronization

Low-level synchronization mechanisms usually make use of specific data structures. In the following subsections, we examine three of the most prominent ones.

Delta maps

A delta map is a data structure or mechanism used to track and represent changes or differences between two versions of a dataset or state. It provides a way to efficiently transmit and apply updates across distributed nodes without transferring the entire dataset.

Delta maps are usually implemented with version vectors or logs, depending on the problem and the application. The synchronization approach with delta maps using version vectors is more carefully examined in Chapter 3 because it is the approach we will use ourselves for detecting the changes needed to be synchronized between the key-value stores and remote object storage systems.

Merkle trees

Merkle trees [Merkle, 1987] are a cryptographic data structure that facilitate efficient and secure verification of data integrity. They operate by organizing data into a tree-like structure, where each leaf node represents a small portion of data and the intermediate nodes store the hash values of their child nodes. The process of constructing a Merkle tree involves recursively hashing pairs of nodes until a single root hash, called the Merkle root. This root hash serves as a compact representation of the entire data set, allowing for efficient lookup of changes. An example of a Merkle-Tree is shown in figure 2.3 [1].

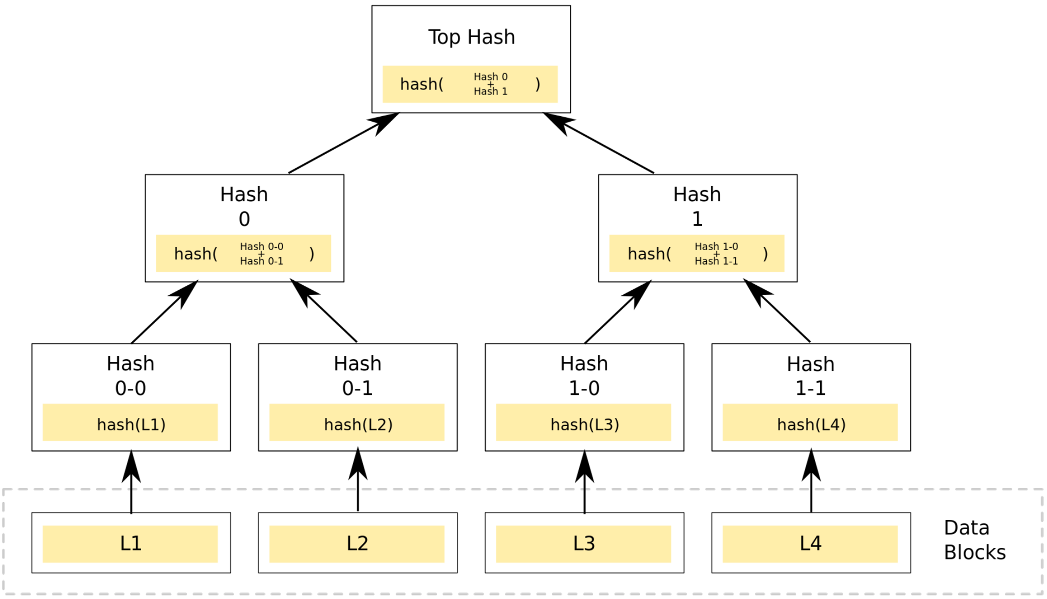


FIGURE 2.3: Example of a simple Merkle Tree.

In the context of synchronizing distributed systems, Merkle trees play a crucial role. They enable multiple parties to compare and synchronize their datasets by efficiently identifying differences in data without transferring the complete dataset. By comparing the Merkle root hashes, participants can determine if their datasets are identical or if specific portions of the data have diverged. This approach minimizes

[1] Credits to Wikipedia

the amount of data that need to be exchanged and reconciled, reducing bandwidth requirements and synchronization time.

Conflict-Free Replicated Data Types (CRDTs)

Related to the field of replica-synchronization, especially in implementing *eventual consistency* in distributed systems is the concept of Conflict-Free Replicated Data Types (or CRDTs) [Shapiro et al., 2011].

CRDTs are a class of data structures designed for distributed systems that aim to provide strong eventual consistency without the need for coordination or centralized authority. They are specifically designed to handle concurrent updates in a distributed environment where there may be latency, network partitions, or conflicting operations.

The key idea behind CRDTs is that they ensure convergence by allowing updates to be commutative and/or associative, meaning that the order of concurrent operations does not affect the final state of the data.

CRDTs have been implemented for a variety of lower-level data structures like counters, sets, maps, registers, even JSONs [Kleppmann and Beresford, 2017].

Chapter 3

Implementation

This chapter is structured as follows: we begin by discussing some common high-level design decisions that apply to all of our implementations. Secondly, we delve into the specifics of each key-value (KV) store, including their internals and implementation details. Lastly, we demonstrate how we leveraged log-structuring to achieve the desired incremental snapshotting functionality of our key-value store.

3.1 Common design decisions

First of all, we go through some design decisions that are common throughout all of the implementations, namely the programming interface and encoding of values.

3.1.1 Application Programming Interface

We designed our implementations to expose a common interface (API) to the programmer. By doing this we allow for easy benchmarking, testing, and ultimately a fair comparison between the engines. The API is programmatically defined within a parent class that is inherited and extended by the classes corresponding to each engine, of which the exact method signatures can be found in appendix A. The methods supported are:

- `get`: For retrieving the value of a given key. This operation is called a *read*.
- `set`: For setting the value of a given key. If the key does not exist, it is inserted in the database with the given value, and if it already exists it is updated to the given value. If the value is empty, this is considered a delete. We refer to all these operations as *writes*.
- `close`: Closes the database by flushing all buffers and closing all files.
- `snapshot`: Takes a snapshot of the current state, by flushing all buffers and pushing the latest created files to a remote directory (more on that in section 3.5). This method takes as argument an integer which is the snapshot identifier. The values of those integers should be unique and ascending but not necessarily consecutive, for example $1, 3, 6, 8$ is a valid sequence of snapshot identifiers for taking four consecutive snapshots, but $2, 5, 4, 8$ is not.
- `restore`: Using the remote directory, it pulls all files associated with a given version, restoring the state of a specific point in time when a snapshot was taken. This method takes an integer as an optional argument, representing the snapshot version to restore. If the argument is not given, the latest version is restored by default.

The decision to treat deletes as mere writes to empty values offers significant advantages in terms of both usage and implementation. By adopting this approach, we eliminate the need to invoke special methods or follow complex deletion procedures. Instead, a straightforward write operation can be used to signify the deletion of a record. On the implementation side, treating deletes as writes to empty values allows us to avoid dealing with intricate concepts such as *tombstones*. In certain database systems, tombstones are special markers used to indicate record deletions [Matsunobu, Dong, and Lee, 2020]. However, by adopting our chosen approach, we eliminate the need for tombstones altogether.

Also, all keys and values are in the form of raw bytes. This the design decision followed in the APIs of major commercial key-value stores too, like *RocksDB* and *Redis*, because besides offering simplicity, it also allows for maximum flexibility, as any other data type can be serialized in bytes (and *has* to be if it is to be written on disk), and makes the encoding of the key-value pairs on disk easy implementation-wise.

3.1.2 Encoding

Regarding the encoding of the key-value pairs on disk, we encode each key-value pair as shown in figure 3.1: we first encode the length of the key in bytes, then we write the key itself, and then we repeat the same for the value. This enables us to avoid any kind of character-escaping mechanisms, special characters, or padding, which would all add complexity, restrictions, and would waste disk space.

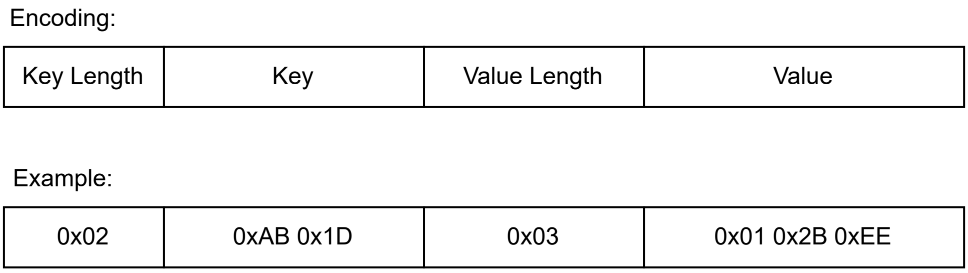


FIGURE 3.1: Encoding & Example. Keys and values are prepended by their respective length values.

An important benefit from using such encoding is that it allows us to encode keys and values of arbitrary size without limit. The trade-off is that we use slightly extra disk space for the encoding bytes if the keys and/or values are large. With encoding bytes of length one we can have keys/values up to $2^{1*8} - 1 = 255$ bytes long, with length two we can have up to $2^{2*8} - 1 = 65535$ et cetera. Each key-value store accepts as argument in the constructor the maximum key length and the maximum value length, which we use to determine the amount of bytes we will use for the encoding.

3.1.3 Filesystem and Persistence

Another design decision is to store all data in files under one directory on disk, which enables easy backups and management in general.

Regarding the behavior of the key-value store upon initialization, if the key-value store is not initialized with a connection to a remote directory, and finds data in its local data directory from a previous run, it will rebuild its indices using this local data.

If on the other hand a key-value store is initiated with a remote connection (either a path in the same machine, which is expected to have been mounted remotely elsewhere, or *minio* - more on that in section 3.5), it will attempt to fetch the latest snapshot. If such a snapshot is available, it will overwrite any preexisting data in its local directory, giving preference to the snapshot.

3.2 Log-Structured Merge-Tree

The Log-Structured Merge-Tree (LSM-Tree) is a disk-based data structure [O'Neil et al., 1996], and one of the most prominent, battle-tested, and well-researched database engines. It was invented by Patrick O'Neil in 1996 and has since been used in multiple databases, such as Google's *LevelDB*, Meta's *RocksDB* and Apache's *Cassandra*.

The LSM-Tree makes extensive use of the *log-structuring* technique, which first appeared in the LFS file system [Rosenblum and Ousterhout, 1992] and has since been used not only in LSM-Tree-based database management systems, but also in other types of storage engines, even B-Tree-based ones [Levandoski, Lomet, and Sengupta, 2013].

Log-structuring offers significant speedups by significantly reducing the number of writes per page and transforming them into a "sequential" format. In other words, it consolidates numerous random writes into a single large multi-page write [Levandoski, Lomet, and Sengupta, 2013].

In this work, we use log-structuring extensively, because, besides its advantages in I/O operations, it also provides a straightforward way to create incremental snapshots of the database's state. We analyze the way we leveraged log-structuring for incremental snapshotting later, in section 3.5.

Given the close relationship between log-structuring and the LSM-Tree (which makes extensive use of it), we will introduce the concept in tandem with the LSM-Tree.

3.2.1 Design

The power of the LSM-Tree can be partially attributed to the fact that it uses lightweight indices, when compared to B-trees which effectively double the cost of every I/O operation to maintain their indices [O'Neil et al., 1996]. This enables the LSM-Tree to scale to very high write and read rates.

However, one other important factor for the LSM-Tree's fast I/O is the use of an in-memory buffer, also called *memtable*, which aggregates the updates and when it's full, it flushes them to disk sequentially. As it is well known, disks perform much faster sequential operations that operations than require random-access, especially in the cloud, where inexpensive disks have limited I/O rates [Levandoski, Lomet, and Sengupta, 2013].

This buffer flushes the aggregated data into *sorted* chunks of data that are commonly referred to as SSTs for "Sorted String Tables", but we will just call them "runs". Sorting is essential for indexing, as it enables us to lookup keys in logarithmic time instead of linear.

Writing data

Initially, as we are writing data, we keep them in our buffer, and when this buffer is full, we flush it into a run-file. This can be seen in figure 3.2, where the file `L0.0.run` is created, corresponding to the first file of the first run (everything is zero-indexed).

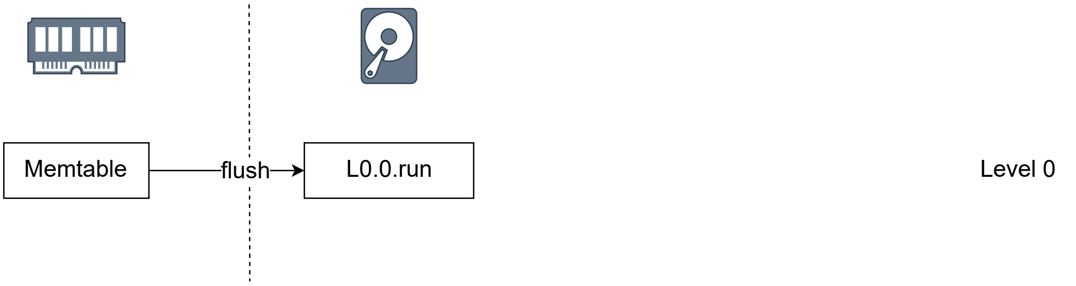


FIGURE 3.2: Example of LSM-Tree flushing.

As we continue writing key-value pairs, we create new runs in the same level by flushing our memtable (figure 3.3), until their number reaches the maximum allowed runs per level, which is defined by the parameter `max_runs_per_level` when instantiating the LSM-Tree. When that happens, a merge is triggered; the merge will merge these files into one file in the next level, and will check if the number of runs in that level is equal to the maximum runs per level. If it is, it will cascade the merging recursively to the next level, and this process will keep happening until no merges need to be done.

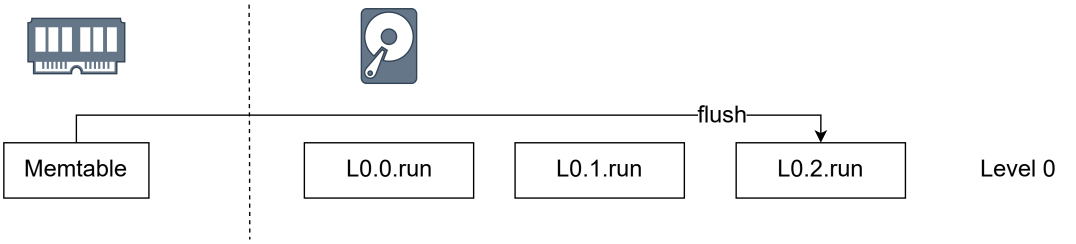


FIGURE 3.3: Example of LSM-Tree flushing (cont.).

The merging process is shown in figure 3.4, where the runs in the first level are merged into `L1.0.run`. After being merged, the files in the first level are deleted. The merging process resembles the greedy merging step in the mergesort algorithm, because every run is sorted. We keep a number of file descriptors equal to the number of runs we are merging, and go through all of them at the same time. We take care to write the smallest key first, to make sure that the resulting merged file is also sorted. In case of two or more conflicting keys during the process, we write the latest one (the one with the largest run index) and skip the rest, as those have been overwritten by a more recent write and are not valid anymore. This is also how the LSM-Tree performs garbage-collection - during the merging process, invalid values are dropped.

Reading data

To retrieve values using the `get` operation, it is necessary to search through the files in reverse order to locate the latest write. This involves performing a binary search on each file, starting from the first level, and then searching within each level from the runfile with the highest index to the lowest.

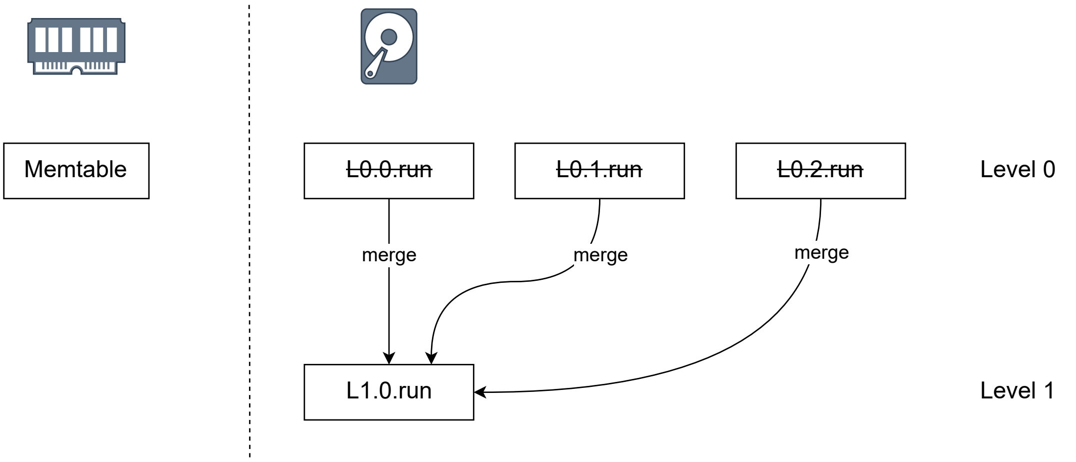


FIGURE 3.4: Example of LSM-Tree merging.

This search can be time-consuming if done on the files themselves, because it would involve a large number of I/O operations, so we use a data structure called *fence pointers* [Li et al., 2009] to speed up the process. The fence pointers are essentially sparse in-memory indices implemented with arrays that allow us to do binary-search in memory, and they associate a key with its offset in the runfile. Of course, they don't store all the keys, as that would be like keeping all the keys in memory and thus we would miss one of the main points of using an LSM-Tree. Instead, we use a subset of them, and since the runfile is itself sorted, if the key we are looking for does not have a fence pointer itself, we still know the offsets among which it should be (hence the name "fence pointers") and we can go ahead and search for it linearly on the file. The gap in numbers of key-value pairs between the offsets of the pointers is controlled via a parameter called `density_factor` - the higher its value, the greater the gaps and the more key-value pairs we have to search sequentially on disk. An example illustrating how fence pointers function can be found in figure 3.5.

The fence pointers offer a significant speedup, but we can skip entire runfiles if we know for sure that they don't contain the key we are looking for by using Bloom filters [Tarkoma, Rothenberg, and Lagerspetz, 2011]. The Bloom filter is a probabilistic data structure that when queried if a key exists in a set (a runfile in our case) it will answer negatively with 100% certainty if it does not. The positive answer is not always accurate, but having a few false positives is no problem for files that we were going to search anyway if we didn't have the Bloom filter.

The Bloom filter achieves this probabilistic lookup by employing a bitarray of m bits and k hash functions. To illustrate how it works, let us consider the example in figure 3.6 with a bitarray of $m = 20$ bits and $k = 3$ hash functions.

To insert value x, we hash it using all three hash functions obtaining the values $h_i(x), i = \{1, 2, 3\}$, then we calculate the values $p_i = h_i(x) \mod m$, and set the bits of the bitarray with positions equal to p_i to 1. When we want to query the Bloom filter, to check whether the value x exists, we use the hash functions and modulo operation again the same way and check whether the bits at positions p_i are set to 1. If there is at least one bit that is set to zero then x definitely does not belong in the set of values inserted in the Bloom filter and if all values are set to one, then x *probably* belongs in the set. The positive answer is probabilistic simply because the same bits may have been set to 1 from insertions of values other than x.

The probability of getting a false positive answer from the Bloom filter is a function of the bitarray length m, the number of hash functions k, and the number of

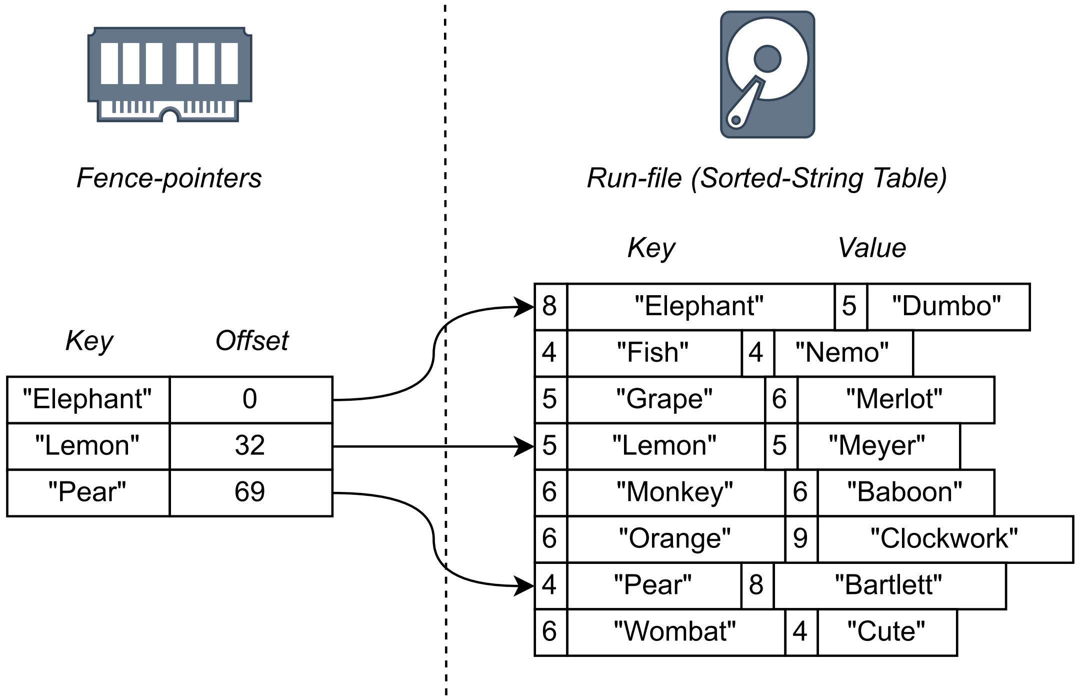


FIGURE 3.5: Fence pointers example with density factor equal to 3, i.e. a pointer that maps a key to a file offset is created for every 3 entries of the SST.

inserted elements n. Assuming that the hash functions are *perfect* i.e. the probability distribution of the hash function values is uniform, we can calculate this probability. With some algebraic manipulation, we can also calculate *the optimal number of hash functions k and bitarray length m, given the number of elements to be inserted n and the false-positive probability ϵ*. These values are shown below, with m being rounded up and k rounded down for performance reasons, implementation-wise:

$$m = \left\lceil -\frac{n \ln \epsilon}{(\ln 2)^2} \right\rceil$$

$$k = \left\lfloor \frac{m}{n} \ln 2 \right\rfloor$$

The creation of Bloom filters and fence pointers occurs in-memory concurrently with the writing of a runfile, when flushing the memtable or when merging other runs. After the Bloom filters are created, they stay in-memory so that they can be queried for faster lookups, but we also persist them on disk by embedding them into the runfiles. If a system failure happens, the key-value store can quickly load all the fence pointers and Bloom filters from the disk without having to go through all the discovered files to rebuild them from scratch. This design decision significantly reduces the recovery time of the store at the expense of using (a bit) more disk space.

To embed the fence pointers and the Bloom filter into a runfile, we append each of them at its end along with two 64-bit values that correspond to their offsets in the runfile. These two offsets therefore define three segments in the run-file: the first is the SST, the second contains the fence pointers and the third is the Bloom filter.

After these additions, value retrieval looks as follows (see figure 3.7): starting from the first level and from the rightmost latest run, we query the Bloom filters for the key we are looking for. When a Bloom filter answers positively, we query

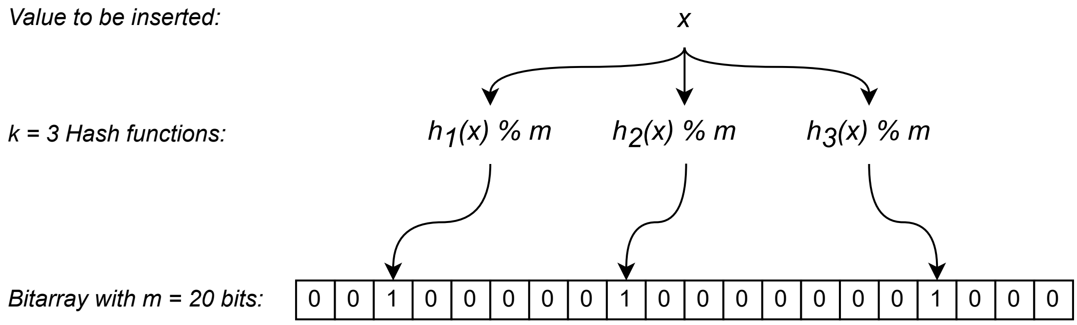


FIGURE 3.6: Bloom filter example with bitarray of 20 bits and 3 hash functions.

the fence pointers, and get an offset. We look up at most d key-values in that file following this offset, where d equals the density factor. If the key is not found, we repeat this process with the next runfile. If we exhaust the lookups and haven't found the key, we return the empty value (0 bytes).

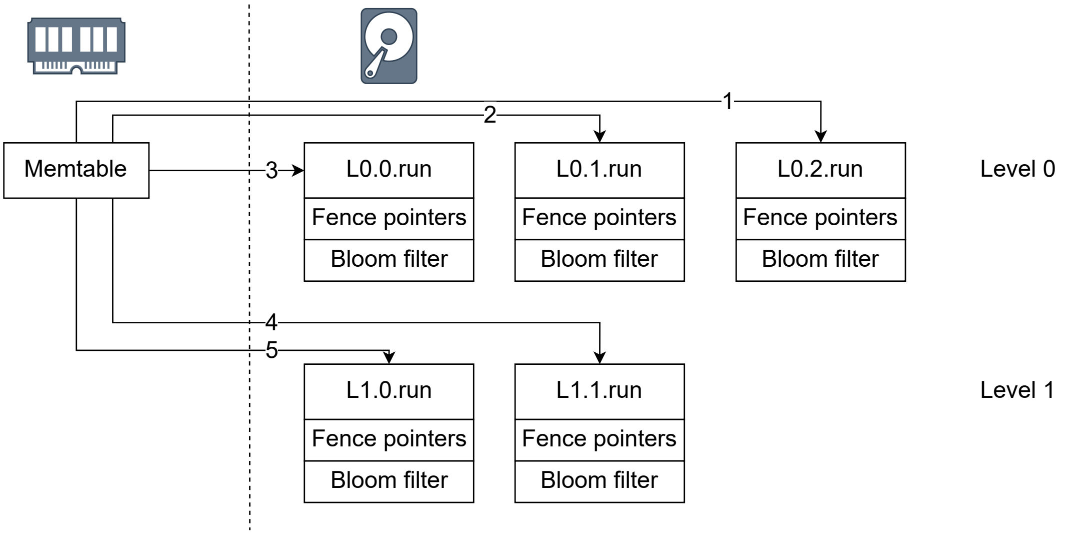


FIGURE 3.7: Example of value retrieval in an LSM-Tree. The numbers in the arrows signify the search order.

Fault-tolerance

As a final design choice, we add a write-ahead log (WAL) to make the database more resilient by preventing loss of unflushed records in the memtable in the event of a system failure. More specifically, when we write a value to the store, we also write it to an append-only log. Since the log is append-only, it is still relatively fast despite the I/O, and at the same time it allows us to *rebuild the memtable* by re-inserting the values after a system crash, making the database more fault-tolerant.

Tiering vs Leveling

LSM-Trees come in two flavors, depending on the merging strategy: there are the LSM-Trees that use *tiering* and those that use *leveling* [Sarkar et al., 2022]. In tiering, we use up to R runs per level, while in leveling we only use one. As we increase R, the first level essentially transforms into an append-only log, which has the highest write speed. However, the reads become slower, as the LSM-Tree has to search a higher number of files to retrieve a value. On the other hand, in leveling when $R = 1$, the LSM-Tree merges each file directly to the runfile of the next level, using the file sizes as thresholds that trigger merges. This optimizes the read performance but impedes the writes due to frequent merges [Sarkar et al., 2021]. Figure 3.8 illustrates the difference between the tiering and leveling.

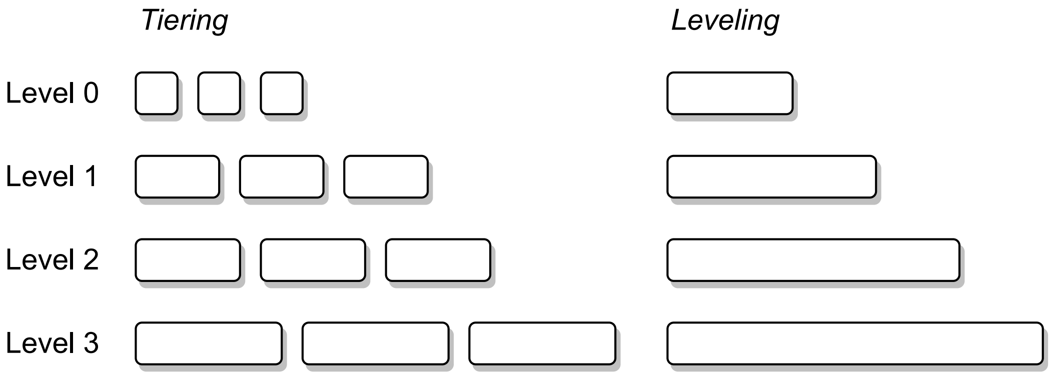


FIGURE 3.8: Tiering vs Leveling in LSM-Trees. Tiering sort-merges R runs per level ($R = 3$ in this example) into a run in the next level, while in leveling each run is greedily sort-merged with the run from the next level.

Our implementation uses tiering because we are optimizing for writes. Nonetheless, the R value described above is still configurable, and we will analyze the performance of the LSM-Tree for various values of it in Chapter 4.

3.2.2 Implementation

As we stressed in the previous subsection, the properties of the LSM-Tree are derived primarily from having *sorted* runfiles. To remove the values from the memtable when flushing it in order, we need a data structure that does this operation efficiently. At the same time, we want this data structure to support efficient lookup and insertion/update of values. These requirements are satisfied by Skip lists, or self-balancing binary-tree structures, like AVL trees and Red-Black trees. The skip list is used in some commercial LSM-Tree-based key-value stores, like *LevelDB* but operations on them are not guaranteed to be efficient due to their probabilistic nature. On the other hand, AVL trees and Red-Black trees have guaranteed access, lookup, insertion, and delete complexity of $\mathcal{O}(\log(n))$.

In our implementation, we used the `sortedcontainers` package, a Python implementation of an associative array which offers the same complexity for the above operations. For the fence pointers, we used the same package because the fence pointers' keys have to be sorted *to allow for efficient in-memory lookup*, with the binary-search algorithm.

For the Bloom filters, we could not use the most popular publicly available implementation due to a versioning incompatibility so we implemented it from scratch using a bitarray which is persisted using base64 encoding. As hash function we used MurmurHash3, a fast non-cryptographic hash function, which is a common choice for Bloom filters and other probabilistic data structures that require general hash-based lookups.

3.3 AppendLog

AppendLog is primarily based on *Bitcask* [Sheehy and Smith, 2010], a log-structured hash-table key-value store. Bitcask constitutes one of the backend choices for Riak, a popular commerical distributed key-value store. It is an operationally and conceptually a simple database, but it is precisely its simplicity that makes it fast and robust.

3.3.1 Design

The AppendLog has two main components: a (log-structured) append-only log, and an in-memory hash-table.

To understand how it operates and its design, we will start with the writes. Leaving aside log-structuring for now, we assume that we only use an append-only log, and we write key-value pairs to it. For every key-value pair we write, we use the hash-table as an index which keeps track of the key-to-offset mapping in this log. The writes in this log are immutable - if we update a key to a new value, we just append it as a new key-value pair.

Then, to read the value of a key, we query the in-memory hash-table for key, get the offset, and seek to this offset and read the key-value pair.

This simple design is very fast because it writes data to the disk sequentially, and sequential I/O is faster in both mechanical and solid-state disks. In mechanical HDDs it is faster because the rotational parts of the disk do not have to seek to other positions so they do not add overhead, and in SSDs sequential writes mitigate the phenomenon of *write-amplification* [Hu et al., 2009] which we introduced in Chapter 2.

However, the design so far has a major drawback; it lacks garbage-collection. As updates to values are appended, the old values are useless and only take up disk space. To solve this issue, we introduce log-structuring to the design, which we have already used in the LSM-Tree implementation to solve a similar problem. With log-structuring, we leverage the merging step to drop the old values.

Concretely, as we write values, we use a size-threshold value for the logfile size that when exceeded, we close the log file and start a new one. These logfiles are equivalent to the runfiles in the LSM-Tree's log-structuring scheme. Then, we use a second parameter as the upper limit of the number of logfiles. When this limit is reached, we merge the files in this run into a new file in the next level and at the same time we update the hash-table index to point to the new location.

This new design decision has the following implication: the index can no longer just point to an offset, as we have multiple files in our log-structured scheme. The solution is to simply store the file information in the hash-table alongside the offset, so the index points to the offset of a specific file.

The entire design so far is visualized in figure 3.9. In this figure, we see an example of a potential snapshot during the operation of an AppendLog instantiated with the parameter of maximum runs per level set to three and maximum key-value capacity per file set to two, right before the merging phase. The first level is full and thus the files `L0.0.run`, `L0.1.run` and `L0.2.run` are about to be merged in `L1.2.run`. We notice how the index always points to the latest record. In the next section (3.3.2) we explain how the merging is implemented.

Compared to the LSM-Tree, the AppendLog has the following advantages:

1. It offers significantly faster reads, since a value retrieval is essentially a query to an in-memory hash-table, a seek to a file offset and a file read operation. There is no need to search multiple files or lookup multiple data structures.

2. It is unencumbered by the overhead that the creation of the fence pointers and the Bloom filters add to the LSM-Tree.

3. The hash-table index itself is faster than the LSM-Tree's insertions and deletions. The hash-table has an complexity for these operations of (amortized) $\mathcal{O}(1)$ while the memtable is $\mathcal{O}(\log(n))$, where n is the number of entries to the memtable.

The advantages however come at the following costs:

1. The keys have to all fit in memory, since they have to be hosted to the hash-table. This hampers the scalability of the AppendLog.

2. The AppendLog does not perform any buffering before flushing the entries to disk. In some cases this fact may degrade performance. We will analyze this further in the following section, 3.3.2.

To recover from a failure, the AppendLog needs to rebuild its main index. To do this, it has to scan all the files in reverse order (i.e. reverse to the order they are written) and for each key-value pair it needs to update the index. After this procedure, the index will point to the latest records.

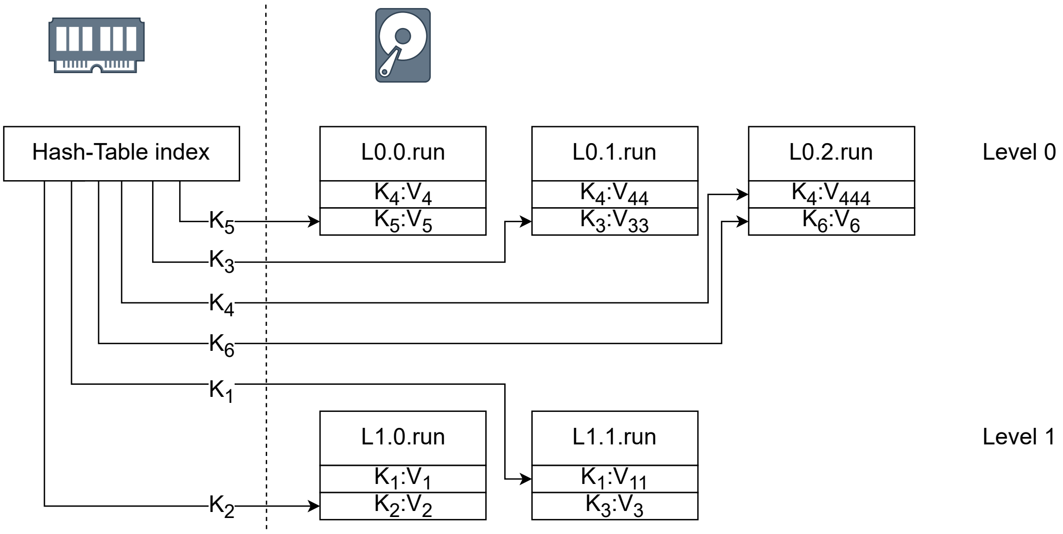


FIGURE 3.9: Example of operation of the AppendLog. K_i:V_j are key-value pairs.

3.3.2 Implementation

Although the implementation of the AppendLog is straightforward, it does feature certain intricacies that require attention, like the merging strategy and the record flushing.

Regarding merging, the resulting files need to be devoid of invalid records, i.e. key-value pairs that have been updated more recently. This step is important as it is the only garbage-collection mechanism. This can be done in multiple ways using extra memory, but there is in fact a way to achieve it using the already present index without extra memory or modifications. Concretely, for a single file, we read through the file sequentially, going over all the key-value pairs. For each key-value pair that we encounter, we query the index - if the offset that the index returns is equal to the current read offset of the file we are scanning, then this means that this record is indeed the latest for the queried key and must be preserved. Thus, we write it to the merged file, otherwise we drop it and continue to the next read. This process is repeated for the rest of the files in a level, resulting in a single merged file. After that, we can delete the merged files.

To improve the efficiency of the merging process in the AppendLog, we have implemented a garbage-collection mechanism called *compaction* that is triggered right after flushing. This feature is optional and can be enabled as needed. By performing some of the garbage-collection work on flushed files before merging, we can distribute the total workload more evenly during the operation of the AppendLog. This, in turn, allows for faster and more streamlined merging, as some of the work that would typically be done during merging has already been completed. We evaluate the effectiveness of this mechanism in Chapter 4.

One ramification of this merging algorithm is that it compels us to write the keys along with the values on disk, because we need to know the key associated with a value so that we can query the index appropriately, leading us to using more disk space. However, there is no other way to know which record is the latest (and at the same time make this information persistent) without using extra memory, which is more expensive than the disk and also volatile. This is also the approach that Bitcask follows [Sheehy and Smith, 2010].

When implementing the AppendLog, we used a profiler to look for bottlenecks. Evidently, the *open()* system call adds significant overhead to I/O operations. This led us to keep the files open for reading (and for writing where applicable) and keep their file descriptors available in memory at all times. The files are then closed when the store's *close()* method is called.

Another intricate point is the flushing of the records. Because the AppendLog does not use any data structure to buffer the writes (at least at the implementation level), we need to flush immediately, otherwise the index may point to an unflushed record and this can lead to an erroneous read. The use of flushing right after a write is necessary, even if it can potentially lead to reduced performance. On the positive side, the AppendLog does not need any write-ahead logging, precisely because it flushes everything immediately.

In the next section we will introduce HybridLog, which uses buffering to avoid flushing immediately.

3.4 HybridLog

The HybridLog is similar to the AppendLog, albeit with a key distinction: contrary to the AppendLog, it does buffer the writes in memory.

The HybridLog is based on the *hybrid log* introduced in Microsoft's KV store FASTER [Chandramouli et al., 2018]. In the following two sections, we will present the design of HybridLog, its differences from the original in FASTER, and its implementation details.

3.4.1 Design

FASTER in the original work [Chandramouli et al., 2018] consists of two main components: A special hash-index, and the hybrid log, which spreads across memory and disk, hence the name.

The hash-index in FASTER is a concurrent, lock-free, and scalable to the number of threads hash-table. It leverages a framework (introduced in the same work [Chandramouli et al., 2018]) called *Epoch Protection Framework* for lock-free coordination between the threads. It consists of 2^k 64-byte cache-aligned buckets, that each has eight 8-byte entries of which the first seven are for entries and the last one serves as an overflow bucket pointer. Each bucket entry has three parts: a *tentative* bit used for concurrency control, a 15-bit tag and a 48-bit address, which points to a record. Each record has an 8-byte header (16 bits for metadata like *invalid* and *tombstone*, required by some log-structured allocators, and 48 bits for storing the address of the next record, in case of conflicts), then the key that we store and finally its value.

These records can either be allocated in memory (using some memory allocator like *jemalloc*), in an append-only log, or in a hybrid log, which combines memory and disk. The hybrid log is a logical log, which holds records that are addressable in a logical address space. This logical address space is presented in figure 3.10, along with the special offsets of it that denote its three main segments: the segment that resides on disk, starting from offset zero up to the *head offset*, the in-memory read-only segment starting from the head offset all the way to the *read-only offset*, and the mutable segment, also in-memory, from the read-only offset onwards. There is also the *tail offset* which points at the offset of the last record. The logical segments themselves are implemented as follows: the area residing on disk is an abstraction of log-structured files, and the area residing in memory is a ring buffer.

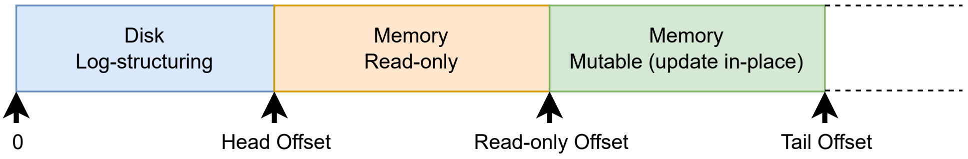


FIGURE 3.10: Logical Address Space used in HybridLog.

As records are written to the HybridLog, we first insert them to the tail of the ring buffer, we update the hash-index, and we move the tail offset further. At every write, we also query the hash-index; if a key exists already in the mutable area, it is updated *in-place*. As we write new key-value pairs and the mutable area grows (because the tail offset moves towards higher logical addresses), we move the read-only offset too if needed, so that it stays behind the tail offset at a constant lag. This lag is configurable as an instantiation parameter of the HybridLog.

The records in the read-only area, as the name implies, are immutable. That is, when a write occurs on a key that is already present in that area, it is copied to the mutable area and updated there, which in the original work [Chandramouli et al., 2018] is called a *read-copy-update*. In our design we simplified a bit this procedure and we just do a new insert of the key-value pair with the new value in the mutable area.

Like with the read-only offset, we also maintain the head offset, which also has to stay at a constant lag behind the read-only offset, and this lag (or interval) is also configurable as an instantiation parameter. When the gap in the logical addresses between the head offset and the read-only offset reaches the defined value of the interval, we flush all the read-only records to disk, i.e. the entire read-only area, and move the head offset to the last logical address that resides on disk, just before the read-only offset.

To retrieve a value, we first query the hash-index; if the key does not exist in the index, we just return the empty value (zero bytes). If the key exists and has a logical offset greater than the head offset, it lies in memory so we retrieve it from the ring buffer. If it resides on disk, we translate the offset appropriately and retrieve it from one of the files by doing a seek and a read operation.

The disk area is log-structured, in the same way that AppendLog is - they both use the same merging strategy for their files, and they both use the same value retrieval method to retrieve values from the files.

It is important to notice how the buffering policy acts like a cache for the writes. The in-memory updates and the read-copy-update from the read-only area exploits the temporal locality of keys. Therefore, this design choice should accelerate workloads with strong temporal locality. Also, the buffering stage does not require continuous flushing of the records by design, turning a succession of frequent small flushes into a large one. This behavior by itself yields faster writes. The downside of this (because no design choice comes without trade-offs) is that we have volatile records. If the system suddenly crushes, we inevitably lose the unflushed records.

To address the issue of potentially lost records, the authors of the original work [Chandramouli et al., 2018] suggest using a write-ahead log as a workaround. Similarly to the approach we took with the LSM-Tree, a write-ahead log can provide a reliable record of updates and help ensure data consistency in the event of system failures.

Regarding recovery after failures, the HybridLog does exactly what the AppendLog does to rebuild its hash-index; it scans all the runfiles in reverse order and points the index to the latest records.

3.4.2 Implementation

The implementation of the HybridLog essentially extends the implementation of the AppendLog by replacing the hash-index, adding the ring buffer, and also adding some logic to support the translation of the logical addresses.

Hash-index

The first step of our implementation is the hash-index. Because Python (the language of the implementation) does not allow low-level memory management, we had to simplify the design. The simplified design can be seen in figure 3.11. The index consists of a Python list that holds "buckets". Each bucket is itself a list of length 8. The first 7 entries are integers, of which the upper bits hold the keys and the lower bits the values (which will be used to hold the logical addresses). The last entry holds the index of the next bucket, in case of overflow. New buckets are allocated at the end of the list holding the buckets.

To lookup a key in this hash-index, we hash it first using the MurmurHash3 hash function which is suitable for hash-based lookups, we calculate the modulo of the hash with the initial number of buckets, and then we follow the buckets, scanning the entries for the key, until we exhaust the buckets.

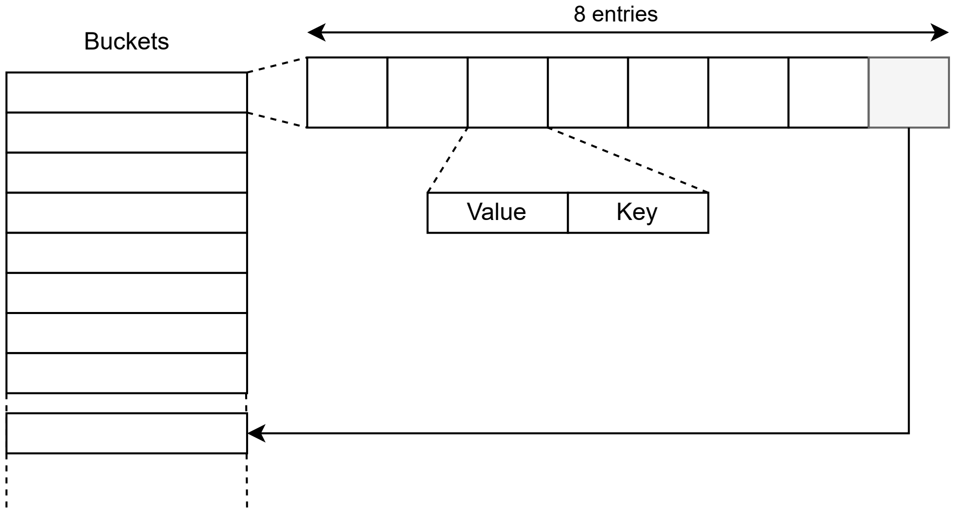


FIGURE 3.11: Hash-index of HybridLog.

To insert or update a new key, we first perform a lookup. If we find the key in some bucket, we update its value. Otherwise, we scan for an empty space an insert the key-value pair. If there is no room, we allocate a new bucket and set the last bucket to point to it. When the inserted key-value pairs reach 75% of the total capacity of the bucket, we resize it by allocating a new one with double the capacity (2^{k+1} buckets if the previous one had 2^k) and copy over the existing records. Deletion is implemented an update of the key's value to the empty value.

Another simplification that is necessary is the removal of the epoch-protection framework. Again, since we are working in Python and we do not have access to low-level threading capabilities, we did not implement the framework.

After implementing the simplified hash-index, we realized that it is actually quite slow, about three times slower than a Python dictionary. Upon reflection, the reduced performance appears to have been a predictable outcome, since it is implemented entirely in Python, while Python's dictionary is implemented in C and bypasses all the overhead that the high-level features of an interpreted language like Python add.

Thus, we continued the implementation using the Python dictionary as the back-end for the hash-index. This choice, in addition to the dictionary being faster, is supported by two more reasons:

1. It allows for fairer comparisons in the evaluations and comparisons in Chapter 4, because the other engines also use the Python dictionary as a HashMap, especially AppendLog which uses the dictionary as its main index as well.

2. We do not have any limitations about the key's length anymore.

Ring buffer

After the hash-index implementation, our attention turned to the ring buffer. This data structure is represented in Python as a list with two pointers, one for reading and one for writing, which wrap around the list in a circular fashion. To achieve this, we calculate the respective buffer offsets using the modulo operation with the buffer's length. This approach allows for efficient and continuous data processing within the buffer, without the need for costly buffer reallocations or data movement.

Flushing and Merging

Then, we implemented the logic for the flushing to disk, along with the log-structuring. Every time the lag between the head offset and the read-only offset reaches the corresponding predefined interval limit (given as a constructor argument), a flush occurs of the read-only area of the ring buffer. Each flush creates a new file. When the number of the files reaches a given threshold, a merge is triggered, which merges the files into one, placed in the next level in our log-structured setup, exactly like we do with the AppendLog.

The next checkpoint of the implementation is the logical address translation. The logical addresses need to be mapped to offsets of the ring buffer or offsets of files. For the ring buffer, the mapping is straightforward: we just use the modulo operator and the size of the buffer. For the disk, we used a Python dictionary which maps a logical offset to a specific offset of a specific file. This decision uses extra memory, but cannot be avoided. In FASTER, the authors use an allocator which also uses extra memory behind the scenes. If we had only one logfile and entries with fixed length, we could have had a one-to-one address translation between the logical offsets and the file offsets by adding or subtracting a constant every time, but *giving up on log-structuring and the freedom to use whatever length for our keys and values is not worth the trade-off*.

3.5 Snapshots

In the context of distributed systems, fault tolerance is central. Replication is one of the most effective methods that systems employ to achieve fault tolerance. By storing copies of data across multiple nodes, replication can help ensure that the system remains available even if some of its nodes fail.

As we design state storage backends, it is important to provide the user with interfaces that allow for remote storage of the state and the ability to access different versions of that state. This includes the ability to roll back to previous versions of it if necessary.

In this section we will look into the method we implemented for creating snapshots efficiently from our log-structured key-value stores, as well storing them in remote storage, and restoring previous versions of it.

3.5.1 Remotes

First of all we define an abstraction we call *Remote*. The remote is an abstraction for remote storage. The endpoints it exposes to the user are the following:

1. `put`: Uploads a file to the remote storage.
2. `get`: Fetches a file from the remote storage. By default it fetches the latest version but a previous version of it can be retrieved as well.
3. `gc`: Keeps only the files associated with the latest version and deletes the rest to free up storage space.
4. `restore`: Retrieves all the files associated with a given version.
5. `destroy`: Deletes the remote storage with all the files in it.

The exact method signatures can be found in appendix A.

For the backend of the remotes we have two implementations: A directory in the local filesystem (to which a remote directory can be mounted) called `PathRemote`, and a bucket in the S3 compatible object store *minio* for the cloud called `MinioRemote`.

To connect a remote storage to one of the key-value stores, the user creates an instance of a type of `Remote` of choice and passes it as a constructor argument when instantiating the key-value store.

If a remote is given to a store, the store will prioritize it over local files for recovery. Instead of performing file discovery at the local data directory to rebuild the indices and the in-memory data-structures from the local pre-existing files, the store will query the remote for the latest version saved. If no version exists, the store starts anew, otherwise it fetches the files of the latest version and uses those to recover the state of that version.

3.5.2 Incremental Snapshots

The implementation of the incremental snapshotting functionality in a log-structured store is done with the use of *delta maps*.

The general idea boils down to this: the store performs file discovery on its local directory and enumerates its local files in a set. It then queries the remote about the files it has stored and the remote sends those files in a different set. The store then calculates the difference between these two sets and then proceeds to "push" or "pull" the missing files, according to which operation is performed, snapshot or rollback respectively. This way, when the store takes a snapshot it will upload only new or changed files containing the recent writes, and if it is restoring a previous version from the remote, it will again only download the differences (the *deltas*), avoiding repetition of work that is already done and hence making the snapshots and recoveries faster. Importantly, the store will flush any records reside in memory before taking a snapshot.

To implement this conceptually trivial procedure we need three things: a way to identify the changed files, a way to keep all the versions of all files in the remote without overwrites, and a way to associate each snapshot with a set of files and their versions. We can cover all three requirements with the following:

1. We add a version counter to each instance of a key-value store called "global version". The value of this counter is appended to each runfile's filename and is incremented every time a merge takes place. To illustrate with an example, the file with name `L0.0.run` after this addition will be `L0.0.0.run`, and it will be different than the file with name `L0.0.1.run` - the latter is created after a merge occurred in level 0. Deletions of files after merging are done as previously, without changes needed.

2. Save all the filenames of a snapshot in a version-file in the remote and associate that file with the snapshot version by adding the snapshot version in the filename.

When the store requests to perform a rollback to a specific version, it will read the version-file associated with the version to be restored and fetch all the relevant files. This can be done again with deltas trivially to be maximally efficient, like it is done in snapshots, but in our implementation we just delete the old files and fetch the files needed, for simplicity. Importantly, when recovering a previous snapshot, the global version of the store needs to be set to a value such that newer snapshots will not interfere with older ones. We choose this value to be the maximum global version detected in the remote files' names, which preserves this invariant and maintains the properties of snapshot and recovery we require in our design.

Incremental snapshotting is visualized in figure 3.12. In this example, the remote has saved a version comprised of the file `L1.0.0.run`, which contains the merged changes from files `L0.0.0.run`, `L0.1.0.run` and `L0.2.0.run`. When the store writes two new files, `L0.0.1.run` and `L0.1.1.run` and takes a snapshot at this point, it will only push these two to the remote, since the remote already has the file `L1.0.0.run`.

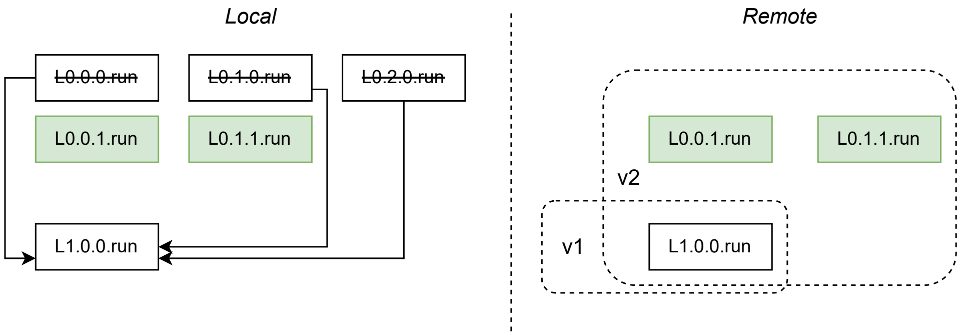


FIGURE 3.12: Incremental snapshotting example.

The global version addition we implemented also has a very convenient side effect: it allows us to execute merges in the background. Since this feature has not yet been implemented, we discuss it in the following section, 3.6.

3.6 Possible extensions and optimizations

In this section, we will explore a range of potential extensions and optimizations for the implemented key-value store engines. We will begin by focusing on LSM-Tree-specific additions and modifications, followed by an examination of optimizations that can be applied to all engine types.

3.6.1 LSM-Tree

The implemented LSM-Tree uses a write-ahead-log (WAL) to ensure that buffered records are never lost. However, this reduces the performance of the writes, as each write must be written to disk instead of solely being added to memtable.

The trade-off is worth it in the general use-case, but probably not for our use-case. This fault-tolerance property is at the level of local storage. In a distributed system, where snapshots are stored in different stores remotely, the WAL may be redundant. In a future version of the key-value store implementation it will be made optional.

At the core of the LSM-Tree, there are a number of optimizations that can be implemented to increase performance:

1. The false-positive probability of the Bloom filters can be tweaked by dynamically allocating memory in order to optimally balance the costs between updates and lookups [Dayan, Athanassoulis, and Idreos, 2017].

2. The merge frequency can be adapted to remove superfluous merges based on the workload and hardware [Dayan and Idreos, 2018]

3. By setting increasing capacity ratios (which are functions of the number of runs at each level) between smaller levels, newer data can be merged more easily, leading to faster writes [Dayan and Idreos, 2019].
4. The segments defined by the fence pointers can be compressed using an online lossless compression algorithm like the LZW algorithm [Welch, 1984] to increase disk-space efficiency.
5. Bloom filters can be added per-level instead of per-file, or added alongside them, to potentially skip entire levels when looking up values in tiered LSM-Trees and substantially boost read performance.

The aforementioned improvements naturally come with a resource trade-off, as their implementation necessitates increased memory usage, larger disk space requirements, or both.

3.6.2 Improvements for all engines

There are several key optimizations that can be universally applied to all three engines:

1. The process of merging files can be trivially executed in the background, particularly leveraging the mechanism of versioned files we implemented for the incremental snapshotting functionality.
2. It is imperative to enable concurrent reads while writing to the engines for enhanced performance.
3. When it comes to writing to files, especially those that involve Bloom filters, the implementation of memory mapped I/O could yield significant benefits. This technique involves mapping files directly into the system's memory, enabling faster and more efficient data transfer between the application and the storage.
4. To ensure data integrity and prevent any potential data corruption, we could augment each record with a small metadata field that includes a checksum. This can be done for example by appending one byte at the end of every record. This additional information allows for the verification of data integrity.
5. In order to enhance "analytics readiness", i.e. the ability to perform aggregations on the data stored, there is an improvement that can be implemented. To perform analytics on the data in a scalable manner, the files need to be independently scannable. To scan each file independently, we should avoid querying any indices. To achieve that, an easy way is through incorporating a "tombstone" (see 3.1.1) within the metadata of each record. This tombstone serves as a marker, indicating the presence of invalid or irrelevant records, thereby allowing for the exclusion of such records without the need to query any indexing mechanism.
6. Compression can be applied to the remotes for more efficient disk-space utilization.

Chapter 4

Evaluation

In this chapter, we perform a series of experiments to evaluate our implementations and pave the path towards answering our research questions. We start off by examining each key-value store implementation independently, by studying how their parameters influence their performance and what trade-offs exist among them. Then, we perform a comprehensive comparison between all the key-value stores, to assess each engine's strengths and weaknesses. Finally, we evaluate the incremental snapshotting functionality in a test environment and in a real-world transactional dataflow system.

4.1 Parameters

Each of our implemented key-value stores is instantiated with a set of parameters. In Chapter 3 we explained what each parameter represents, but to be able to understand the trade-offs among them, and how various settings of them influence the behavior of the respective engine, it is important to explore them visually.

In this section, the experiments performed aim to highlight qualitatively the effect of each parameter and do not constitute stress tests.

For the following demonstrations, we use by default - unless explicitly stated otherwise - the following settings: The randomly generated keys and values have lengths of 4 bytes, the sets of available keys and values have cardinality 10^3 each, the distribution of picking keys and values from the sets is uniform, the input write and read throughput are 10^3 writes and 10^3 reads per second respectively, and for latency measurements that are sampled (to calculate the 50th and the 95th percentile), the number of samples is 10. Also, for the LSM-Tree we use `max_runs_per_level`=3, `memtable_bytes_limit`=10^3, `density_factor`=10, for the parameters of HybridLog we use `ro_lag_interval`=10^3, `flush_interval`=10^3, and for the AppendLog we use `threshold`=10^3 and `compaction=False`.

All the tests in this section are performed in a machine with 4 Intel Xeon CPU cores, 8 GB of RAM and an NVMe SSD drive, the write bandwidth of which has been benchmarked (with the `fio` tool) at 16.6 MB/s.

4.1.1 LSM-Tree

Max Runs per Level

The first parameter of the LSM-Tree is `max_runs_per_level`. This controls the maximum amount of runs allowed in a level. As explained in Chapter 3, as the number of runs per level increases, a log-structured database becomes write-optimized, and when it is kept close to 1, the database is optimized for reads. In figure 4.1 we demonstrate this behavior:

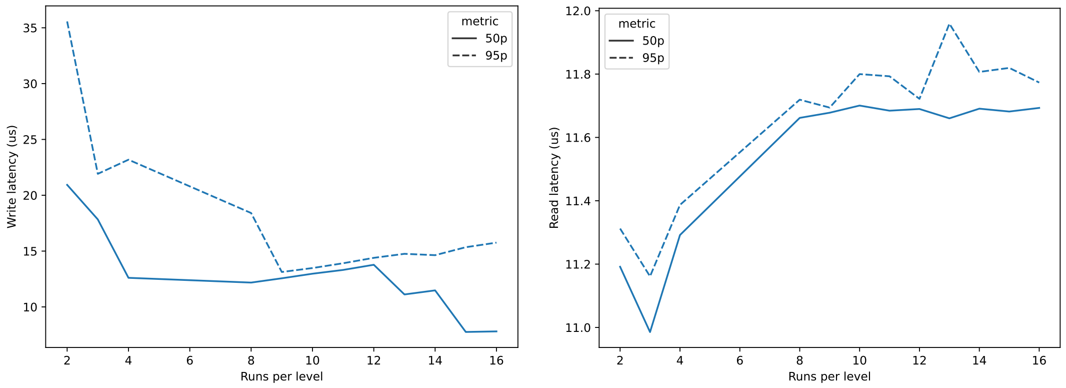


FIGURE 4.1: Latency vs Max Runs per Level.

Clearly, the write latency drops, when `max_runs_per_level` increases, and the read latency is low when the parameter is relatively small.

The LSM-Tree behaves as expected due to the following reasons: when the number of runs per level increases, the log-structuring scheme degrades into a large fragmented log spread over several smaller logs with infrequent merges. This essentially becomes a large log, enabling the maximum writing speed. However, at the same time, accessing a key requires searching through multiple runs per level, leading to slower reads.

This parameter is central, and relevant not only to the LSM-Tree but to the other two log-structured engines, HybridLog and AppendLog. More specifically, the effect on the write latency on these two is the same, but not quite so for the read latency. Because of the fundamental difference in indexing (the latter two use in-memory hash-based indices that point directly to files and offsets), the read latencies are not affected. One needs to just keep the parameter "balanced" enough so that then merges are not very large and infrequent, which would impact the overall performance of the stores.

Density Factor

The `density_factor`, as explained in section 3.2.1, controls the width of gaps between the fence pointers of the LSM-Tree.

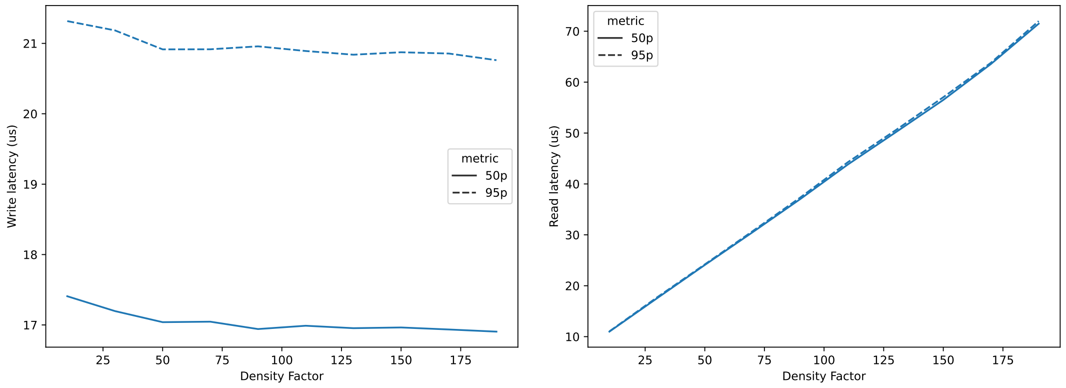


FIGURE 4.2: Latency vs Density Factor.

In figure 4.2 we observe the following: as the density factor increases, the writes remain virtually unaffected, and reads become drastically slower. This is because the

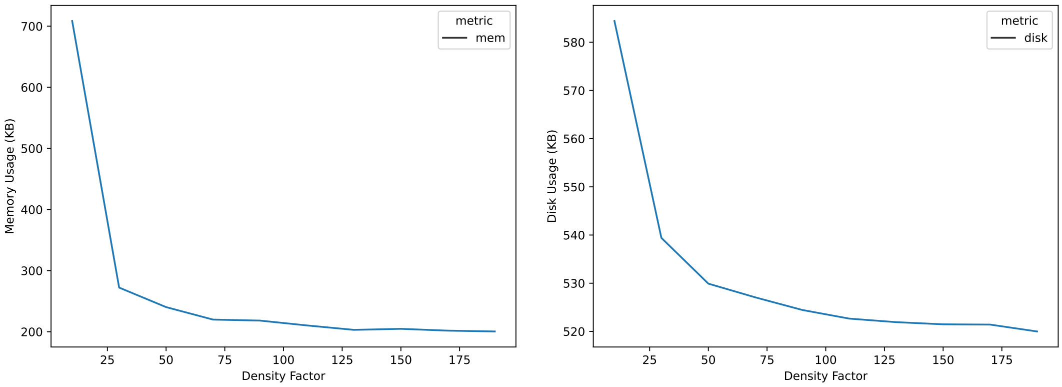


FIGURE 4.3: Memory and Disk Usage vs Density Factor.

LSM-Tree, when the density factor is high and therefore the gaps within the offsets are large, has to go through more bytes in the file to find the requested key, which slows down the reads.

However, there is an obvious tension here: we cannot keep the density factor too small, because that would result in higher memory and disk usage, as demonstrated in figure 4.3.

Memtable Size

The size of the LSM-Tree's memtable, controlled by the `memtable_bytes_limit`, is the amount of bytes the in-memory structure can hold before it flushes to disk.

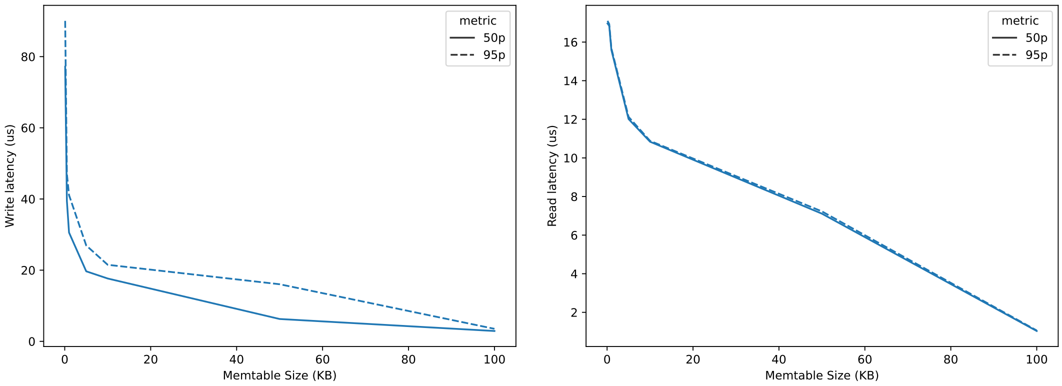


FIGURE 4.4: Latency vs Memtable Size.

In figure 4.4 we notice that as the size of the memtable increases, the latency of both the writes and reads drops. This is expected, as with bigger memtables, the probability of accessing a key without the need to reach to the disk is higher. However, the memory usage obviously goes up, as seen in figure 4.5, and thus we cannot keep this parameter too large.

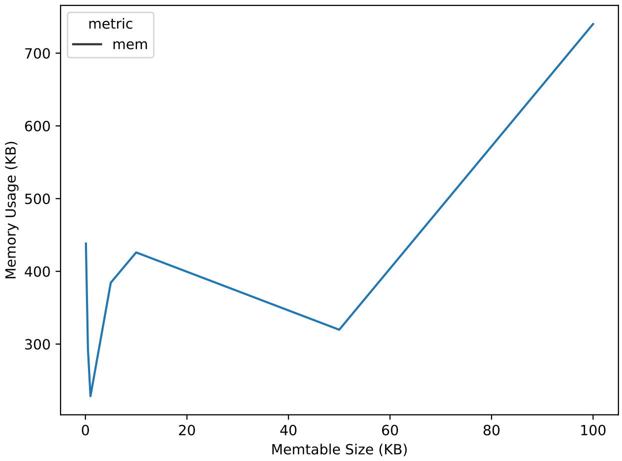


FIGURE 4.5: Memory Usage vs Memtable Size

4.1.2 HybridLog

Mutable Segment Size

The mutable segment size of the HybridLog memory segment is controlled by the value of the RO (read-only) Lag Interval - `ro_lag_interval`. This parameter influences directly the probability of an in-memory hit of a key lookup, and thus the cache-like behavior of the whole memory segment.

If this value is large, we expect many in-memory hits and therefore better performance for both writes and reads. This is exactly what we observe in figure 4.6. However, we obviously cannot increase this segment indefinitely because

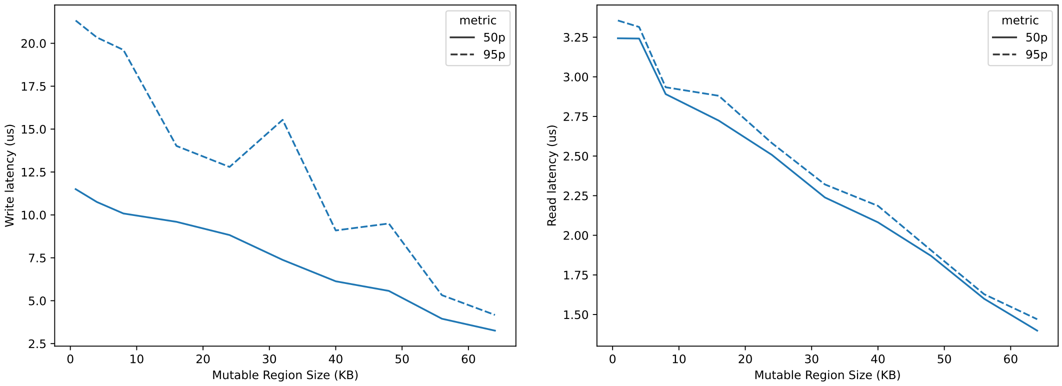


FIGURE 4.6: Latency vs Read-only Segment Size.

Read-only Segment Size

The read-only segment, whose size is adjusted via the `flush_interval` parameter, contains read-only entries that are ready to be flushed to disk. The larger the segment, the less the probability for disk access and therefore the higher the performance of the key-value store. This is evident in figure 4.7. The obvious trade-off present here, is that if this value is set to be large, we require a larger memory segment size, which will use more memory.

Additionally, it is crucial to ensure that the value is not set too low. If it is set too low, it may impede the speedup of performance from large flushes to disk, which occur sequentially and are therefore fast. Furthermore, setting the value too low may result in numerous small logs that require frequent merges, thus adversely impacting performance. This phenomenon is also illustrated in the same figure 4.7.

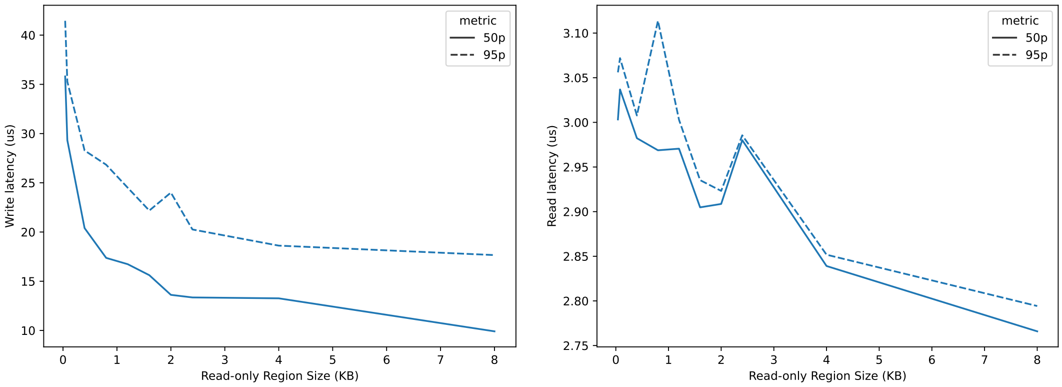


FIGURE 4.7: Latency vs Flush Segment Size.

4.1.3 AppendLog

Threshold

The threshold value is the maximum amount of bytes we can write to a runfile in AppendLog, before closing it and starting the next one.

This parameter is similar to the `flush_interval` parameter of the HybridLog. When it is too low, frequent merges hinder the write performance, and as it increases, writes on average become faster (because the runfile becomes essentially a large append-only log). However, if the threshold is too high, the files become large and the merges infrequent and cumbersome, which explains the widening of the gap between the 50p and 95p lines in the write latencies in figure 4.8. As for the reads, they are not significantly affected, as expected.

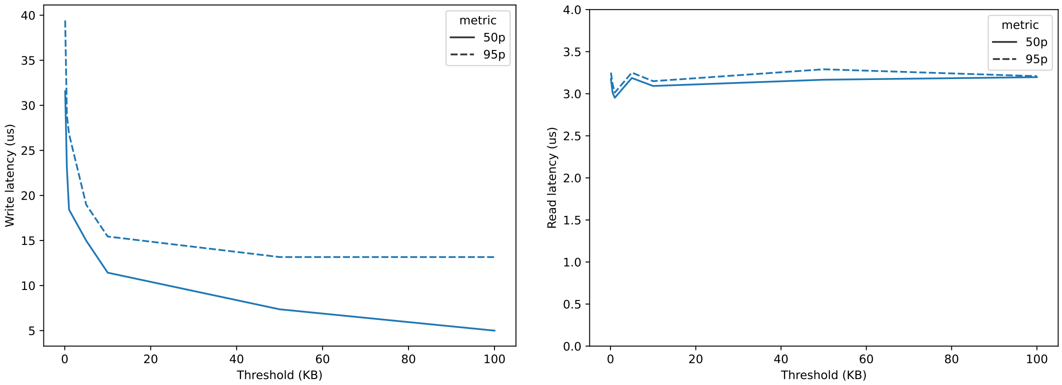


FIGURE 4.8: Latency vs Threshold.

Compaction

Compaction is an experimental optional feature that we will evaluate empirically. It could offer some speedup in practice, or it could be the case that its potential benefit is already implicitly provided during merging and the system is just wasting time doing extra unnecessary work.

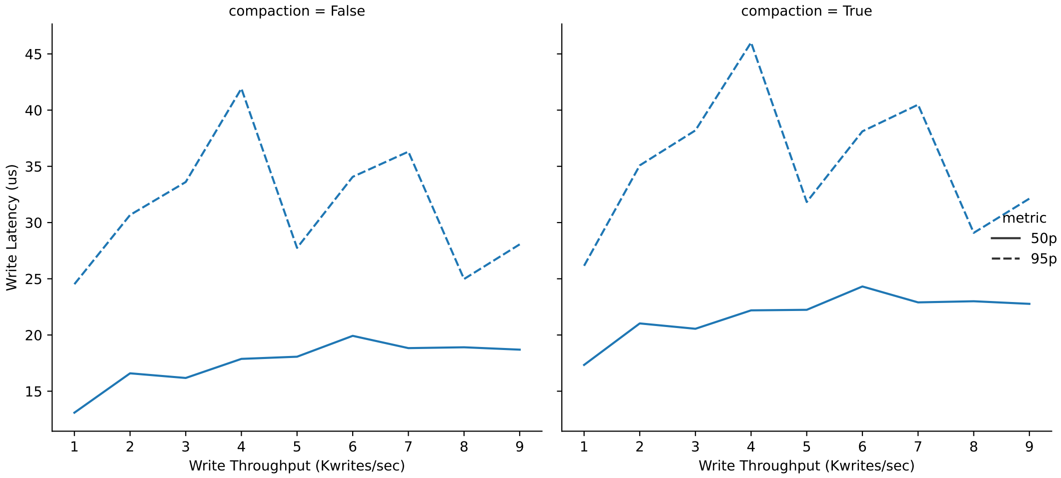


FIGURE 4.9: Write Latency vs Throughput, with Compaction disabled (left) and enabled (right).

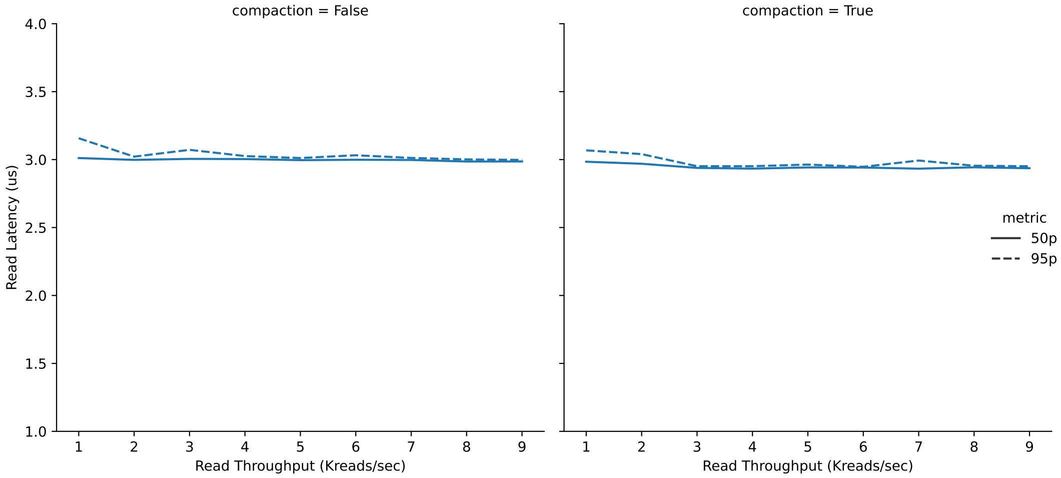


FIGURE 4.10: Read Latency vs Throughput, with Compaction disabled (left) and enabled (right).

From the experiment results in figures 4.9 and 4.10 it seems that this is exactly the case. Compaction offers no advantage for reads (which was expected, since file access is still the same), but also no advantage for writes, which are in fact impaired, as compaction introduces a significant overhead. Therefore, compaction should be avoided in our use-case.

4.2 Comparison

In this section we proceed to compare the engines on their performances when executing the same task with similar parameters. For the following experiments, we use the following parameters: Key and value lengths of 5 bytes each (so 10-byte key-value pairs), 10^5 unique keys and values, and 10 samples per average latency measurement for the percentiles. Also, for all engines we use `max_runs_per_level`=10, for the LSM-Tree `density_factor`=10 and `memtable_bytes_limit`=100K, for the HybridLog `ro_lag_interval`=10K and `flush_interval`=10K, and for the AppendLog `threshold`=100K and `compaction=False`.

The above settings lead to almost equally sized files on disk, and use the same configurable memory, so the comparison is as fair as possible.

4.2.1 Write Latencies

In figure 4.11 we observe the write latencies of each engine as we increase the input throughput. When choosing keys uniformly, HybridLog and AppendLog are significantly faster than the LSM-Tree. This can be attributed to the fast (amortized $\mathcal{O}(1)$) hash-based indexing of those engines, versus the LSM-Tree's memtable's data structure, which has an insert complexity of $\mathcal{O}(\log(n))$. This is also the reason that when we use a state with a size that fits the in-memory structures and therefore does not need to "spill" to disk, the HybridLog still performs faster, as can be seen in figure 4.12.

When we choose keys using a Zipfian distribution instead, some keys are accessed compared to the Uniform distribution, the LSM-Tree and the HybridLog become faster than earlier, because the Zipfian distribution allows them to better leverage their in-memory buffering structures before flushing, thus reducing I/O operations, and the AppendLog becomes slower, because it lacks any similar buffering method to take advantage of the Zipfian distribution. Among them, the HybridLog is clearly the fastest, precisely because its memory segment with its fast in-place updates of recently written records exploits the Zipfian distribution best.

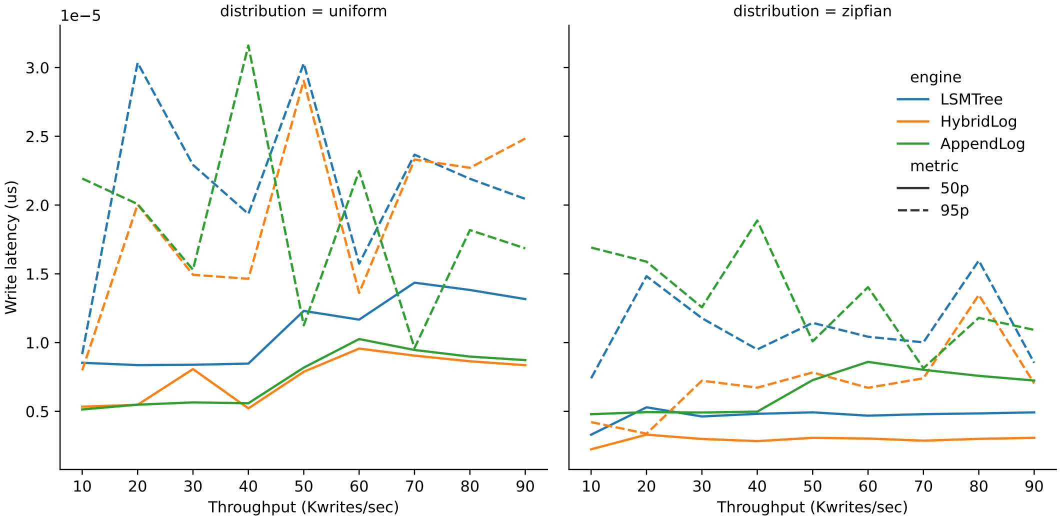


FIGURE 4.11: Write Latency vs Throughput, for Uniform and Zipfian data distributions.

4.2.2 Read Latencies

Upon examining the latencies for the reads in figure 4.13, it becomes clear that the HybridLog and AppendLog outperform the LSM-Tree by a large margin. This is

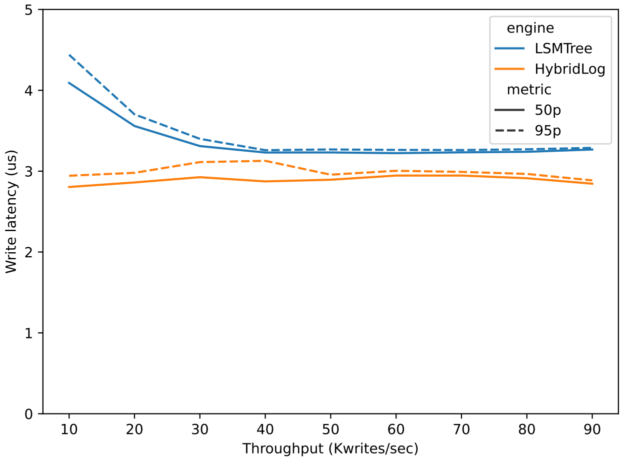


FIGURE 4.12: Write Throughtput when data fits the memory

because of their fast hash-based in-memory indices and minimal I/O.

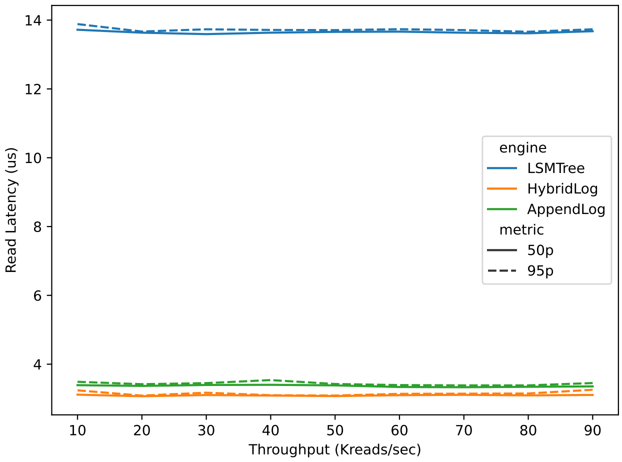


FIGURE 4.13: Read Latencies

4.2.3 Recovery Time

For this experiment, we perform a sequence of writes to each key-value store, and then we close it, restart it, and measure the time that each of them takes to perform file discovery and rebuild all the in-memory data structures (indices etc.).

The results can be found in figure 4.14. The LSM-Tree has by far the fastest recovery because it only needs to deserialize and load into memory the Bloom filters and the fence pointers. The other two stores need to fully scan every file and insert the keys and their file offsets to their in-memory indices.

4.2.4 Memory

HybridLog's superiority as the fastest key-value store comes at the cost of high memory usage, as can be seen in figure 4.15. Indeed, it is the store with the most in-memory structures, including its main index. After that comes the AppendLog, which also keeps its index in memory. Finally, the LSM-Tree uses the least memory

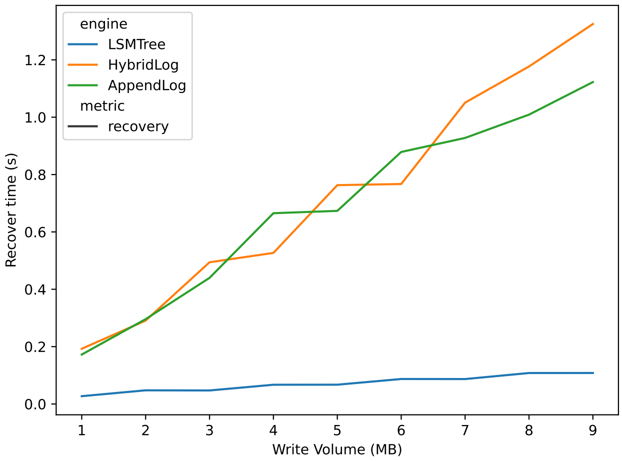


FIGURE 4.14: Recovery Times

of all, making it ideal for low-memory environments (and also the cheaper option). The components requiring memory in the LSM-Tree are the Bloom filters and the fence pointers, which we keep in memory for fast access.

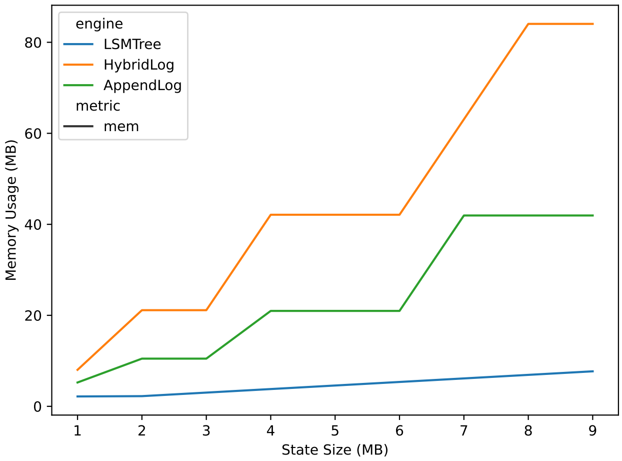


FIGURE 4.15: Memory Usage

4.3 Incremental Snapshotting

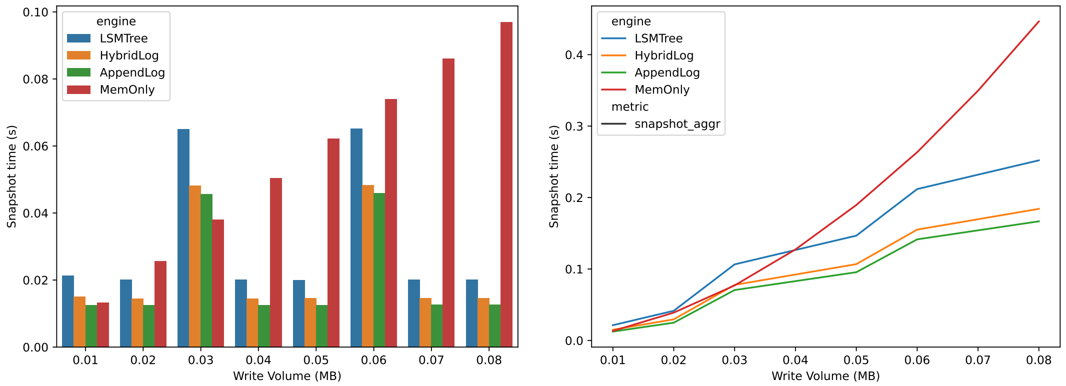
This section focuses on evaluating the incremental snapshotting capabilities of the three log-structured engines. Towards this goal, to demonstrate the advantage of having incremental snapshots, we compare the LSM-Tree, HybridLog and AppendLog to "MemOnly", which is a naive implementation of a key-value store based on an entirely in-memory hosted HashMap that dumps its whole state to disk every time we want to take a snapshot of it.

We do two experiments. In the first, we iterate and write new key-value pairs, taking also a snapshot at the end of each iteration. In the second experiment, we first perform a large write-volume of 1GB, and then we write data in small increments on 1KB, taking a snapshot after each increment.

For both experiments, we use keys and values of 2 and 8 bytes respectively so that the available keys are no more than 2^{16} and therefore we will not need too much memory for the indices of HybridLog, AppendLog and MemOnly. Also, to simulate a snapshot over the network, we add an overhead of 1μs per byte (as if we had a network channel of 1MB/s). The settings for all engines are similar so that the comparison is as fair as possible.

The results of the first experiment are shown in figure 4.16. As expected, the naive MemOnly database dumps the whole state at every step, leading to a quadratic increase of the total time taken to take n snapshots, while the other log-structured stores increase linearly. During each snapshotting step, they only dump the new inserts, except from a few cases when some merging takes place and have to push some larger files as well, but still, they perform better than MemOnly.

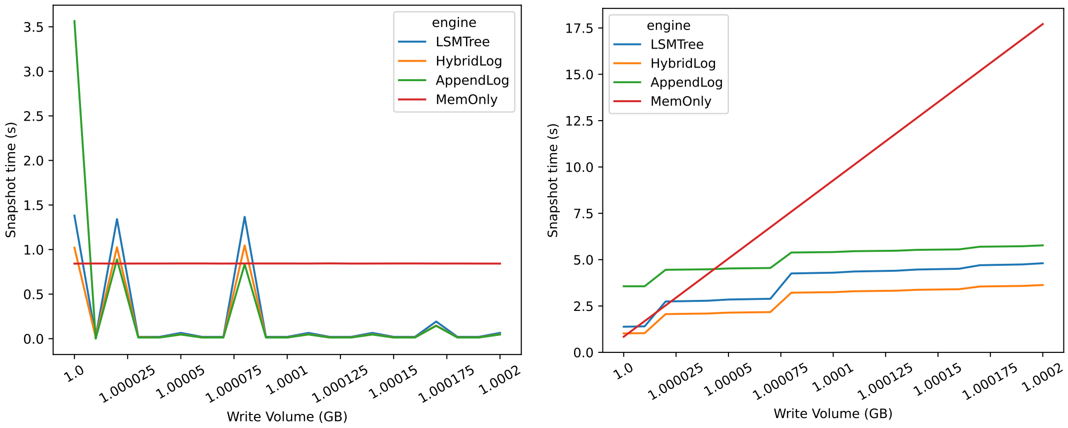
For the second experiment, where only updates take place, the results can be seen in figure 4.17. Again, as expected, the LSM-Tree, HybridLog and AppendLog only push the updates, while the MemOnly store pushes the whole state every time. By observing the cumulative graph, it is evident that the log-structured stores take snapshots more efficiently than the naive method.



(A) Discrete

(B) Cumulative

FIGURE 4.16: Snapshotting Time vs Write Volume, when we increase the state by inserting new records.



(A) Discrete

(B) Cumulative

FIGURE 4.17: Snapshotting Time vs Write Volume, when state stays the same and we only update it.

	MemOnly	**LSM-Tree**	**HybridLog**	**AppendLog**
Spill-to-disk	No	Yes	Yes	Yes
Strongest point	Fastest performance	Fastest recovery, lowest memory	Fastest performance (with spill-to-disk)	Fastest snapshot
Memory Requirements	Keys and values must fit in mem.	None	Keys must fit in mem.	Keys must fit in mem.
Data Loss (w/o snapshot)	Will lose all records	None	Will lose unflushed records	None
Incremental Snapshots	No	Yes	Yes	Yes

TABLE 4.1: Summary of the properties of the key-value stores.

The important takeaway from these two experiments is that while the cumulative time of the naive snapshotting method increases quadratically at the worst case, the log-structured incremental methods increase linearly. This distinction can have significant ramifications in the performance of systems that keep large states.

4.4 Discussion

We summarize our observations in table 4.1.

Chapter 5

Conclusion

5.1 Summary

In this work, we have implemented three different key-value stores, as state backends that support incremental snapshotting in transactional dataflow SFaaS systems. We analyzed their behavior and the trade-offs governing their operation under different settings of their parameters, gaining insight on how they should be tweaked to deliver the best performance according to the use case. Then, we performed fair comparisons between them, indicating the strengths and weaknesses of each and the domains on which each of them excels. Finally, we implemented logic to support incremental snapshotting capabilities and rollback to previous versions, and evaluated this as well.

5.2 Future Work

TODO

Appendix A

Code

A.1 Key-value store API

```python
class KVStore:
    def __getitem__(self, key: bytes) -> bytes:
        ...

    def __setitem__(self, key: bytes, value: bytes) -> None:
        ...

    def get(self, key: bytes) -> bytes:
        ...

    def set(self, key: bytes, value: bytes) -> None:
        ...

    def __sizeof__(self) -> int:
        ...

    def close(self) -> None:
        ...

    def snapshot(self, id: int) -> None:
        ...

    def restore(self, version: Optional[int] = None) -> None:
        ...
```

LISTING A.1: Key-value store API - method signatures.

A.2 Remote API

```python
class Remote:
    def put(self, filename: str) -> None:
        ...

    def get(self, filename: str) -> None:
        ...

    def gc(self) -> None:
        ...

    def restore(self, version: Optional[int] = None) -> None:
        ...

    def destroy(self) -> None:
        ...
```

LISTING A.2: Remote API - method signatures.

Bibliography

Apache Kafka. https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management. [Online].

Armbrust, Michael et al. (2018). "Structured streaming: A declarative api for real-time applications in apache spark". In: *Proceedings of the 2018 International Conference on Management of Data*, pp. 601–613.

Burckhardt, Sebastian et al. (2021). "Durable functions: semantics for stateful serverless." In: *Proc. ACM Program. Lang.* 5.OOPSLA, pp. 1–27.

Bykov, Sergey et al. (2011). "Orleans: cloud computing for everyone". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pp. 1–14.

Carbone, Paris et al. (2015). "Apache flink: Stream and batch processing in a single engine". In: *The Bulletin of the Technical Committee on Data Engineering* 38.4.

Carbone, Paris et al. (2017). "State management in Apache Flink®: consistent stateful distributed stream processing". In: *Proceedings of the VLDB Endowment* 10.12, pp. 1718–1729.

Cassandra. https://cassandra.apache.org/. [Online].

Castro, Paul et al. (2019). "The rise of serverless computing". In: *Communications of the ACM* 62.12, pp. 44–54. URL: https://dl.acm.org/doi/pdf/10.1145/3368454.

Chandramouli, Badrish et al. (2018). "Faster: A concurrent key-value store with in-place updates". In: *Proceedings of the 2018 International Conference on Management of Data*, pp. 275–290.

Chandy, K Mani and Leslie Lamport (1985). "Distributed snapshots: Determining global states of distributed systems". In: *ACM Transactions on Computer Systems (TOCS)* 3.1, pp. 63–75.

Comer, Douglas (1979). "Ubiquitous B-tree". In: *ACM Computing Surveys (CSUR)* 11.2, pp. 121–137.

Dayan, Niv, Manos Athanassoulis, and Stratos Idreos (2017). "Monkey: Optimal navigable key-value store". In: *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 79–94.

Dayan, Niv and Stratos Idreos (2018). "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*, pp. 505–520.

— (2019). "The log-structured merge-bush & the wacky continuum". In: *Proceedings of the 2019 International Conference on Management of Data*, pp. 449–466.

Dong, Siying et al. (2017). "Optimizing Space Amplification in RocksDB." In: *CIDR*. Vol. 3, p. 3.

Fraser, Neil (2009). "Differential synchronization". In: *Proceedings of the 9th ACM symposium on Document engineering*, pp. 13–20.

Gens, F et al. (2019). *IDC FutureScape: Worldwide IT Industry 2020 Predictions. IDC.*

GNU dbm. https://www.gnu.org.ua/software/gdbm/. [Online].

Gray, Jim and Andreas Reuter (1992). *Transaction processing: concepts and techniques*. Elsevier.

Heus, Martijn de et al. (2022). “Transactions across serverless functions leveraging stateful dataflows”. In: *Information Systems* 108, p. 102015.

Hu, Xiao-Yu et al. (2009). “Write amplification analysis in flash-based solid state drives”. In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pp. 1–9.

Kalavri, Vasiliki and John Liagouris (2020). “In support of workload-aware streaming state management”. In: *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*, pp. 19–19.

Kleppmann, Martin and Alastair R Beresford (2017). “A conflict-free replicated JSON datatype”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.10, pp. 2733–2746.

Kuszmaul, Bradley C (2014). “A comparison of fractal trees to log-structured merge (LSM) trees”. In: *Tokutek White Paper*.

Lamport, Leslie, Robert Shostak, and Marshall Pease (2019). “The Byzantine generals problem”. In: *Concurrency: the works of leslie lamport*, pp. 203–226.

Levandoski, Justin, David Lomet, and Sudipta Sengupta (2013). “LLAMA: A cache/storage subsystem for modern hardware”. In: *Proceedings of the International Conference on Very Large Databases, VLDB 2013*.

LevelDB. https://github.com/google/leveldb. [Online].

Li, Yinan et al. (2009). “Tree indexing on flash disks”. In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE, pp. 1303–1306.

Matsunobu, Yoshinori, Siying Dong, and Herman Lee (2020). “MyRocks: LSM-tree database storage engine serving facebook’s social graph”. In: *Proceedings of the VLDB Endowment* 13.12, pp. 3217–3230.

Merkle, Ralph C (1987). “A digital signature based on a conventional encryption function”. In: *Conference on the theory and application of cryptographic techniques*. Springer, pp. 369–378.

Noghabi, Shadi A et al. (2017). “Samza: stateful scalable stream processing at LinkedIn”. In: *Proceedings of the VLDB Endowment* 10.12, pp. 1634–1645.

O’Neil, Patrick et al. (1996). “The log-structured merge-tree (LSM-tree)”. In: *Acta Informatica* 33, pp. 351–385.

Rajan, R Arokia Paul (2018). “Serverless architecture-a revolution in cloud computing”. In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. IEEE, pp. 88–93.

Redis. https://redis.com/. [Online].

RocksDB. https://github.com/google/leveldb. [Online].

Rosenblum, Mendel and John K Ousterhout (1992). “The design and implementation of a log-structured file system”. In: *ACM Transactions on Computer Systems (TOCS)* 10.1, pp. 26–52.

Sarkar, Subhadeep et al. (2021). “Constructing and analyzing the LSM compaction design space”. In: *Proceedings of the VLDB Endowment* 14.11.

Sarkar, Subhadeep et al. (2022). “Compactionary: A Dictionary for LSM Compactions”. In: *Proceedings of the 2022 International Conference on Management of Data*, pp. 2429–2432.

Shafiei, Hossein, Ahmad Khonsari, and Payam Mousavi (2019). “Serverless Computing: A Survey of Opportunities, Challenges, and Applications”. In: *ACM Computing Surveys (CSUR)*.

Shapiro, Marc et al. (2011). “A comprehensive study of convergent and commutative replicated data types”. PhD thesis. Inria–Centre Paris-Rocquencourt; INRIA.

Sheehy, Justin and David Smith (2010). “Bitcask: A log-structured hash table for fast key/value data”. In: *Basho White Paper*.

Tarkoma, Sasu, Christian Esteve Rothenberg, and Eemil Lagerspetz (2011). “Theory and practice of bloom filters for distributed systems”. In: *IEEE Communications Surveys & Tutorials* 14.1, pp. 131–155.

Welch, Terry A. (1984). “A technique for high-performance data compression”. In: *Computer* 17.06, pp. 8–19.

Zhang, Haoran et al. (2020). “Fault-tolerant and transactional stateful serverless workflows”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 1187–1204.