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

May 14, 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

TODO

Acknowledgements

TODO

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 [López et al., 2021]. 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].

In order to be highly scalable, FaaS offerings are stateless. However, as most applications require some form of state-keeping, developers are often forced to manage their applications' state using external databases. Recently, there have been multiple works that aim to relieve the burden of state-management from the shoulders of application developers [Bykov et al., 2011; Burckhardt et al., 2021; Zhang et al., 2020], by handing application state to external databases and making their management transparent to the developers, providing them with *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 have to be ACID - Atomic, Consistent, Isolated, and Durable [Gray and Reuter, 1992].

The result is often the developers mixing business logic with consistency checks, rollbacks, snapshots and timeouts, leading to systems that are exceptionally hard to maintain and prone to failures. The need for an intermediary layer that abstracts the distributed fault-tolerance logic and provides the application developer with certain guarantees, at the state level or even at the transactional level if possible, becomes evident.

SFaaS systems build 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 [TODO cite], 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.

1.1 Problem Statement

In a (transactional) dataflow SFaaS system, the key-value stores need to have specific properties to be considered suitable. These properties are [Chandramouli et al., 2018]:

1. *Incremental snapshots* [TODO cite?]. 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 will present a way to have the best of both worlds: *both fast incremental snapshotting and low recovery times*.

2. *State recovery to a previous version from previous snapshots (rollback)* [TODO cite?]. 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, especially for workflows that perform aggregations on data or analytics [TODO cite?]. Therefore, the state backend should be suitable 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 [TODO cite].

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 point we can form our research questions:

1. Which types of key-value stores are more fitting as embedded state stores in the worker processes of transactional dataflow SFaaS systems?
2. How do changes in the parameters of each selected type of key-value store affect its performance?
3. 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?
4. How does the performance of a key-value store that offers incremental snapshotting functionality compare to that of a "naive" in-memory key-value store that snapshots its entire state at each step, in terms snapshot creation time?
5. Is there a key-value store that is absolutely superior for state management?

1.3 Contributions

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

1. We have implemented three different key-value stores, as it is crucial to ensure that comparisons are made on a level playing field. This means that all key-value stores have been implemented using the same language and with similar design choices for mutual functionality, such as data encoding and data structures. This approach ensures that only the key-value store logic differs, allowing for fair comparisons.
2. We conduct a series of experiments to answer our research questions we posed in section 1.2. Specifically, we analyze the parameters of each implemented key-value store and examine the trade-offs in their designs with respect to resource utilization.
3. We perform a comprehensive comparison among each key-value store, evaluate the effectiveness of incremental snapshotting, and ultimately determine whether a key-value store stands out as the best option for our use-case.

1.4 Outline

The thesis is structured as follows: Chapter 2 presents the related work. [...TODO] Chapter 3 introduces comprehensive descriptions of the internals of each type of key-value store and the implementation details and design decisions that belong to each of them. [...TODO] In Chapter 4, we evaluate our implementations, performing benchmarks and comparisons between the key-value stores. Additionally, we

discuss the results obtained and answer our research questions. Finally, in Chapter 5 we summarize our research, we present our conclusions and we propose potential directions for future research.

Chapter 2

Related Work

2.1 TODO

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 capability of our key-value store.

3.1 Common design decisions

3.1.1 Application Programming Interface

Firstly, 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).
- `restore`: Using the remote directory, it pull all files associated with a given version, restoring the state of a specific point in time when a snapshot was taken.

The decision for deletes to be just writes to empty values was taken because it greatly simplifies both usage and implementation. The user does not have to call special methods, and on the side of the implementation, we avoid dealing with intricacies like "tombstones" - special markers popular in many databases (like [Matsunobu, Dong, and Lee, 2020]).

Also, all keys and values are in the form of raw bytes. This is also the design decision followed in the APIs of major commercial key-value stores 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 makes the encoding of the key-value pairs on disk easy.

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 escaping and special characters. Additionally, each key-value store accepts as arguments 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.

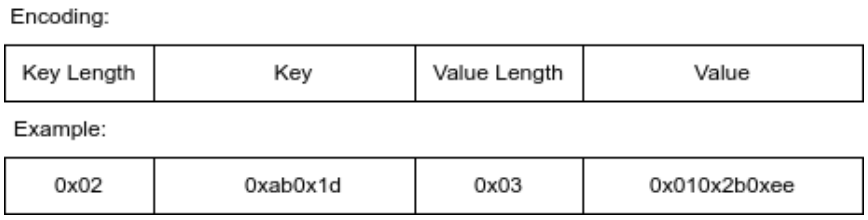


FIGURE 3.1: Encoding & Example.

Another design decision is to store all data in files under one directory on disk, which enables easy backups and management in general. Upon startup, a key-value store will attempt to fetch the latest snapshot, if it has been initiated with a connection to a remote source (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). If a key-value store is not connected to a remote directory, and finds data in its local data directory from a previous run, it will rebuild its indices from this data.

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 key-value store backend engines. It was invented by Patrick O'Neil in 1996 and has since been used in multiple databases, such as Google's LevelDB [*LevelDB*], Meta's RocksDB [*RocksDB*] and Apache's Cassandra [*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.

So, 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.

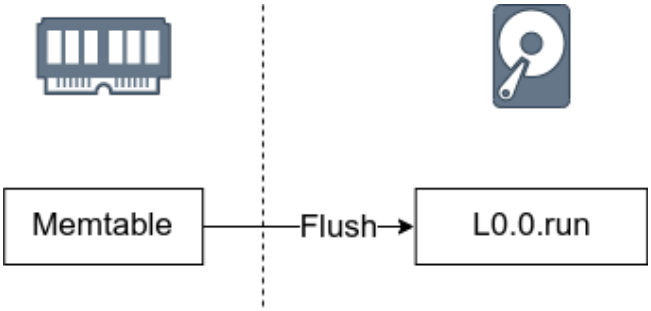


FIGURE 3.2: LSM-Tree flushing.

As we continue writing key-value pairs, we create new runs in the same level by flushing our memtable, 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. The merging process is shown in figure 3.3, 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.

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

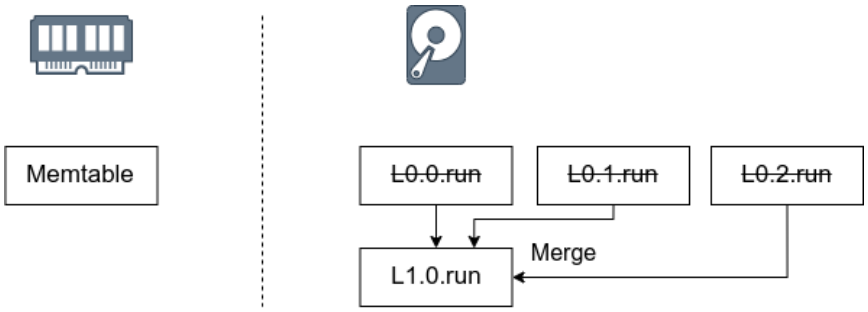


FIGURE 3.3: LSM-Tree merge.

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.

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 arrays that allow us to do binary-search in memory, and 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.

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.

After these additions, value retrieval looks as follows (see figure 3.4): 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 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).

As a final design choice, we add a write-ahead log (WAL) to make the database more resilient. 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 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. For our use-case, the trade-off is worth it.

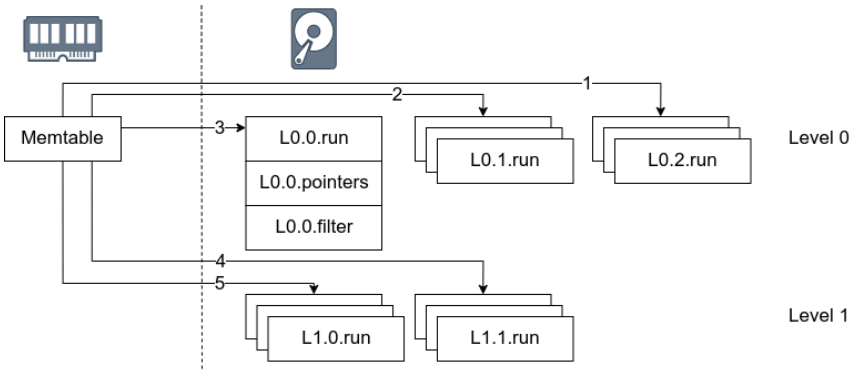


FIGURE 3.4: LSM-Tree value retrieval.

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 [Sarkar et al., 2021].

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. We use JSON to serialize them and store them to disk, and when we want to load them in memory, we read the JSON file, rebuild the sorted container and keep it in memory to serve queries.

For the bloom filters, we could not use the most popular publicly available implementation due to a versioning incompatibility so we implemented it. We serialize and deserialize it using JSON as well, with base64 encoding for the Bloom filter's bitarray.

3.3 AppendLog

AppendLog is primarily based on *Bitcask* [Sheehy and Smith, 2010], a log-structured hash-table key-value store. Bitcask constitutes now one of the backend choices for Riak's, a distributed key-value store. It is an operationaly simple store, 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.

Ignoring 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].

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 3.5. 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 analyse this further in the following section, 3.3.2.

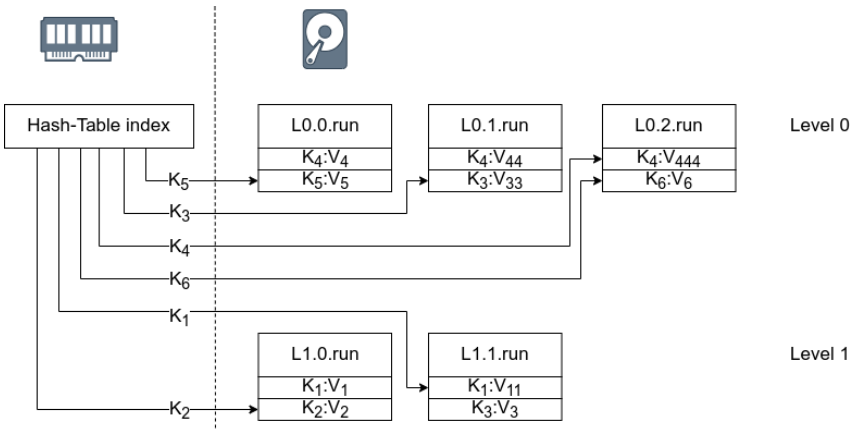


FIGURE 3.5: Example of operation of the AppendLog.

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.

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].

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 3.6, 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.

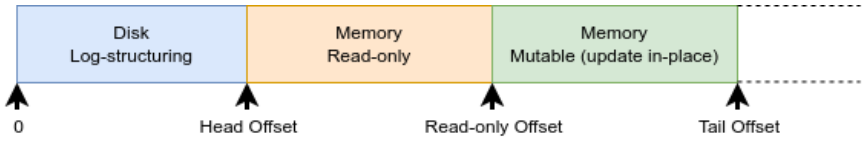


FIGURE 3.6: Logical Address Space used in HybridLog.

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.

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.

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 3.7. 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.

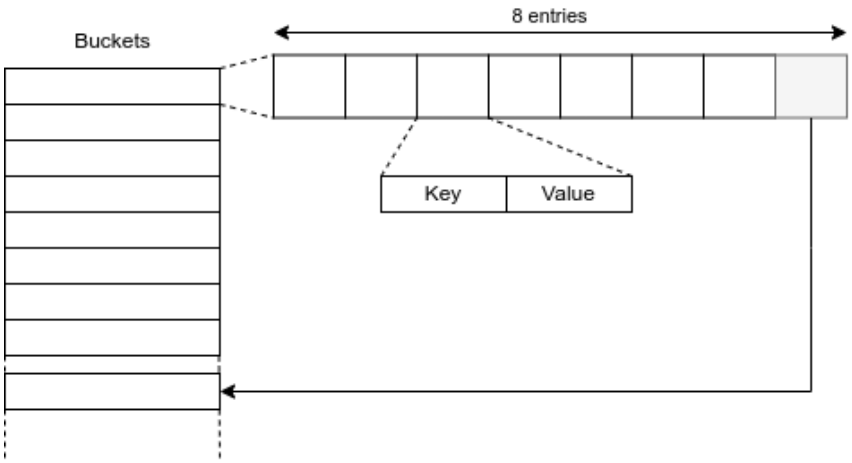


FIGURE 3.7: Hash-index of HybridLog.

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.

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 backend 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.

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.

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.

To improve the efficiency of the merging process in the HybridLog, we have implemented a garbage-collection mechanism called *compaction* that is triggered before merging. 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 HybridLog. 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 its effectiveness in Chapter 4.

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 Replicas

First of all we define an abstraction we call *Replica*. The replica 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 replicas we have two implementations: A directory in the local filesystem (to which a remote directory can be mounted) called `PathReplica`, and a bucket in the S3 compatible object store *minio* for cloud setups.

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

If a replica 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

Every time a key-value store creates a file, it uploads it automatically to the remote storage via the replica. The replica maintains a map that maps filenames to their latest versions and *does not overwrite files*. Instead, if for example a file is uploaded a second time with the same filename, the replica will keep both files under different versions.

We leverage this property in log-structuring as follows: a file in log-structured storage belongs in a level and has a specific "run index" within that level, therefore can be characterized by two integers. In the replica, such file is kept under the same two integers plus a third integer for the version. Now as the key-value store operates, new backups of the files are being created in the remote storage and tagged with their versions automatically. A user can also trigger a snapshot manually, which will flush any in-memory records and upload any new files created.

3.5.3 Rollback

Having the versioned files in the remote storage, we can now revert back to old versions. We demonstrate the method through an example. Assume we want to revert to version 7, in a log-structured store with a maximum of 3 runs per level. First, we find which files exactly we need to retrieve by translating the the version into a list, each element of which denotes which runs we need to fetch from which level. In our example, this array would be $[(0,0),(1,0),(1,1)]$, meaning that we have to fetch one file from the first level (file `L0.0`) and two files from the second level (files `L1.0` and `L1.1`). The algorithm for this translation - or "version-expansion" - can be found in appendix A.

Once this version has been expanded, we fetch the files we need *at their latest versions*. For this purpose we use the map that the replica maintains internally. Finally, the store rebuilds all indices from those files, as if it has started up and has discovered these files in its local data directory.

Chapter 4

Evaluation

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 length 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, and for the parameters of HybridLog we use `mem_segment_len`=10^4, `ro_lag_interval`=10^3, `flush_interval`=10^3, and `compaction_enabled=False`.

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:

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

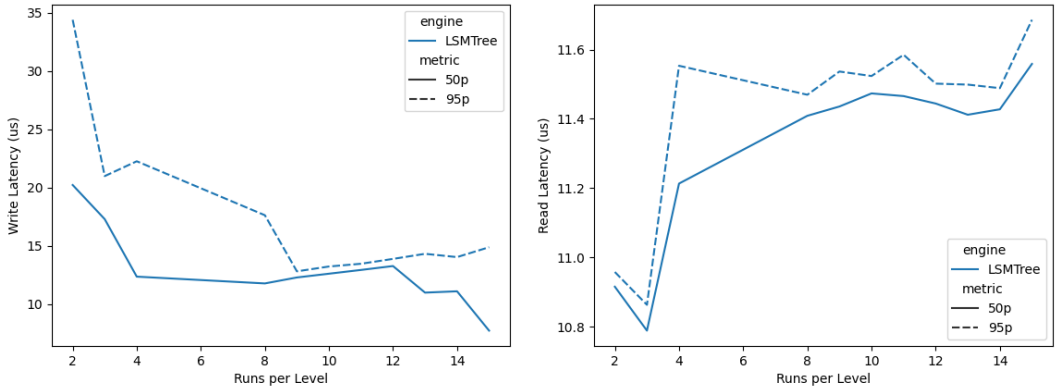


FIGURE 4.1: Latency vs Max Runs per Level.

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.

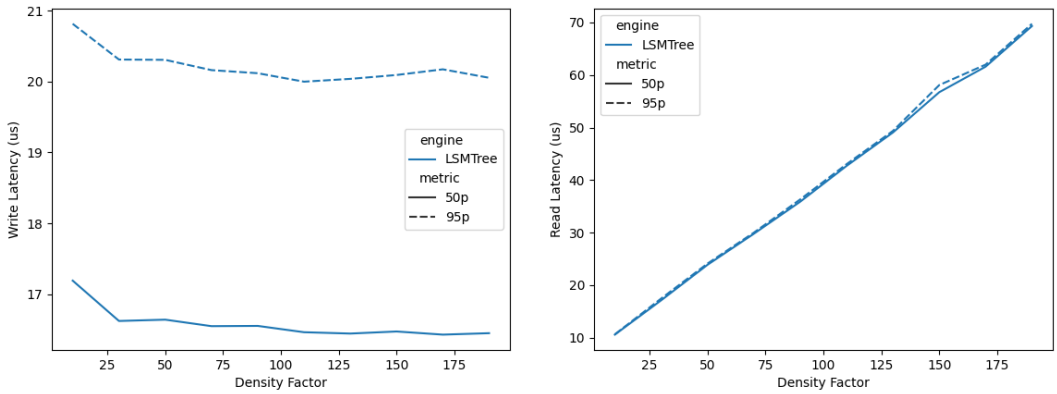


FIGURE 4.2: Latency vs Density Factor.

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 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.

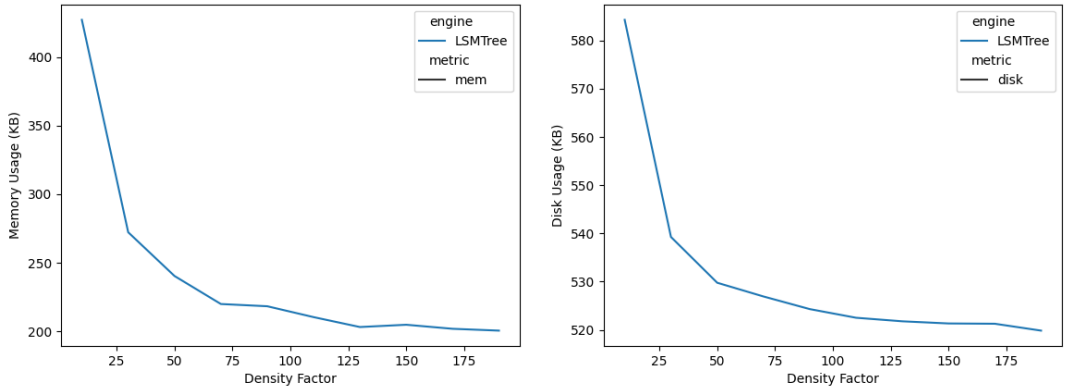


FIGURE 4.3: Memory and Disk Usage vs Density Factor.

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.

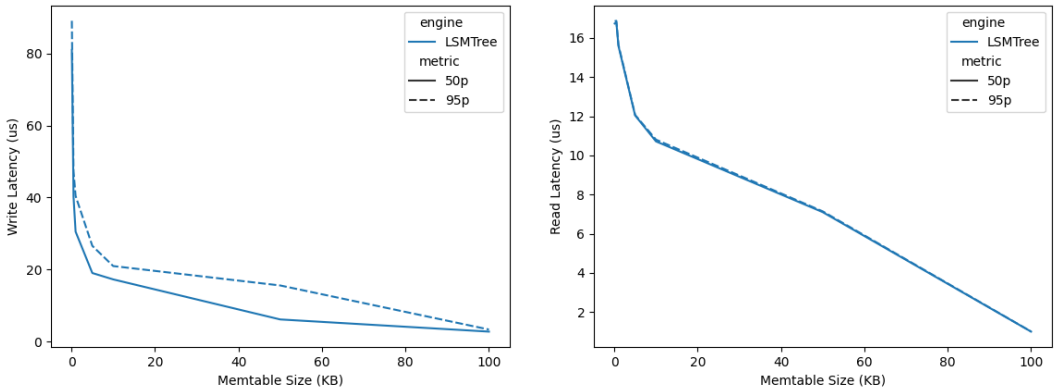


FIGURE 4.4: Latency vs Memtable Size.

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.

4.1.2 HybridLog

Memory Segment Size

Besides the indices, HybridLog also keeps a memory segment in memory, which is essentially a ring buffer. The parameter `mem_segment_len` controls the size of this segment. In figure 4.6 we see its influence in the latencies of the writes and the reads, and in figure 4.7 we see the memory usage.

As expected, the size of the in-memory segment is irrelevant to the speed of both writes and reads, while it directly affects the memory used by the engine. It

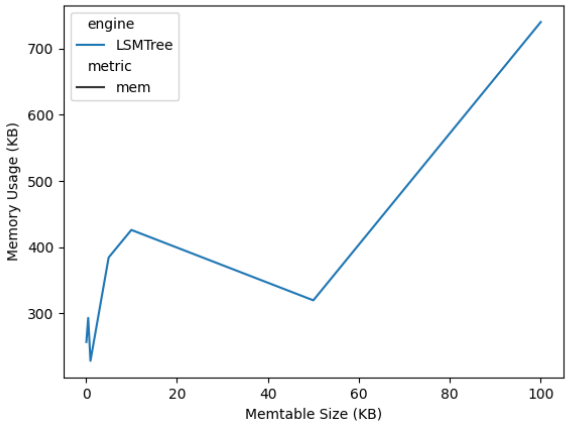


FIGURE 4.5: Memory Usage vs Memtable Size

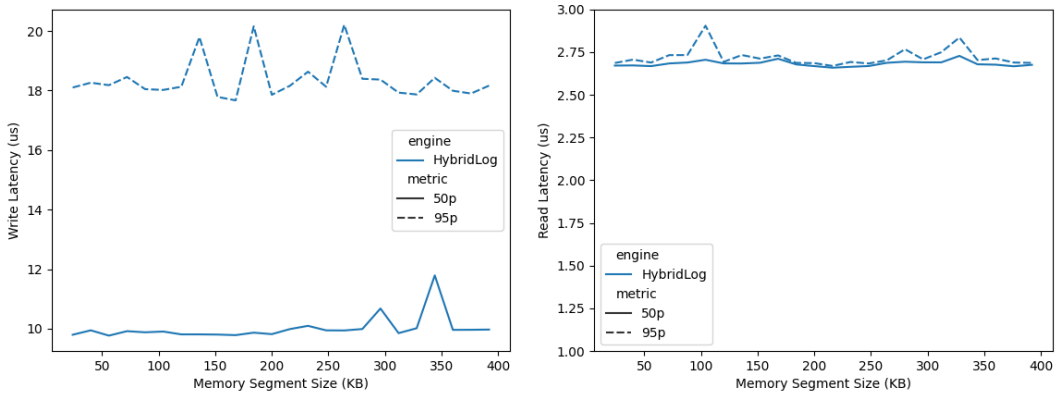


FIGURE 4.6: Latency vs Memory Segment Length.

is irrelevant to the latencies because, as we will see later, it is the `ro_lag_interval` which actually matters.

Hence, it is important that we keep this parameter as low as possible. Since it must always hold that the size of the memory segment is larger than the sum of the sizes of the sub-segments defined by `ro_lag_interval` and `flush_interval`, this parameter should ideally be set a value slightly larger than the sum of these two intervals.

Read-only Segment Size

The read-only segment size is controlled via the value of `ro_lag_interval`. Contrary to the memory segment size, this is the parameter which actually 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 big, we expect many in-memory hits, therefore better performance for both writes and reads. This is exactly what we observe in figure 4.8.

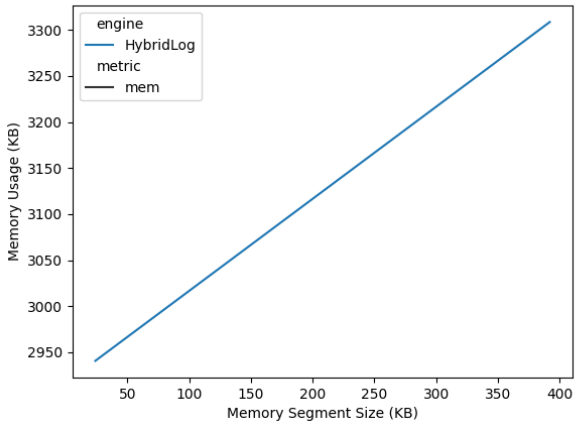


FIGURE 4.7: Memory Usage vs Memory Segment Size

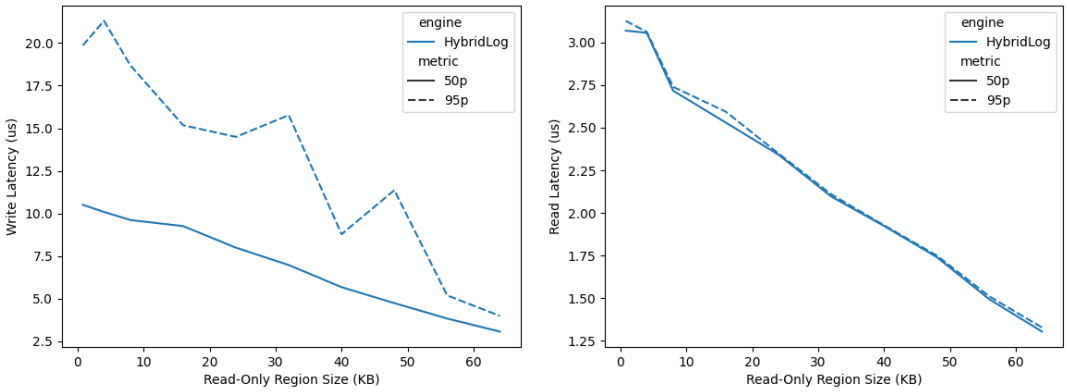


FIGURE 4.8: Latency vs Read-only Segment Size.

Flush Segment Size

The flush segment, whose size is adjusted via the `flush_interval` parameter, contains read-only entries that are ready to be flushed to disk. The bigger 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 4.9. 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 4.9.

Compaction

Regarding compaction, one may wonder if it could offer some speedup in practice, since it could be the case that its potential benefit is implicitly provided during merging already, and the system is just wasting time doing extra unnecessary work.

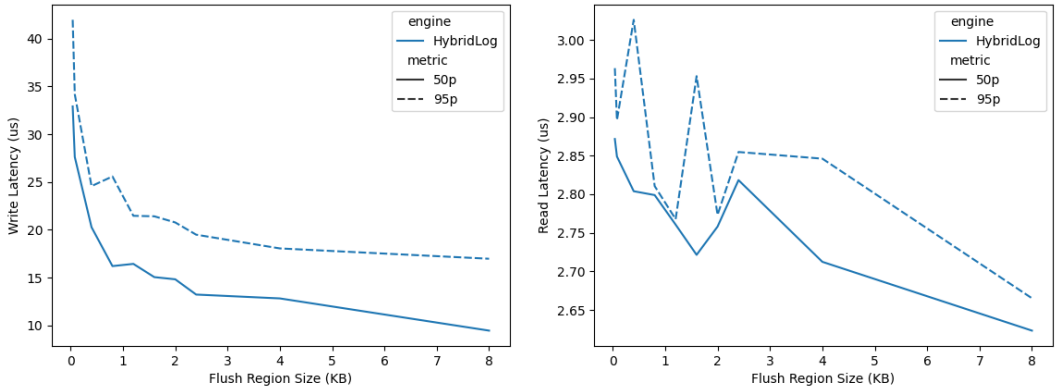


FIGURE 4.9: Latency vs Flush Segment Size.

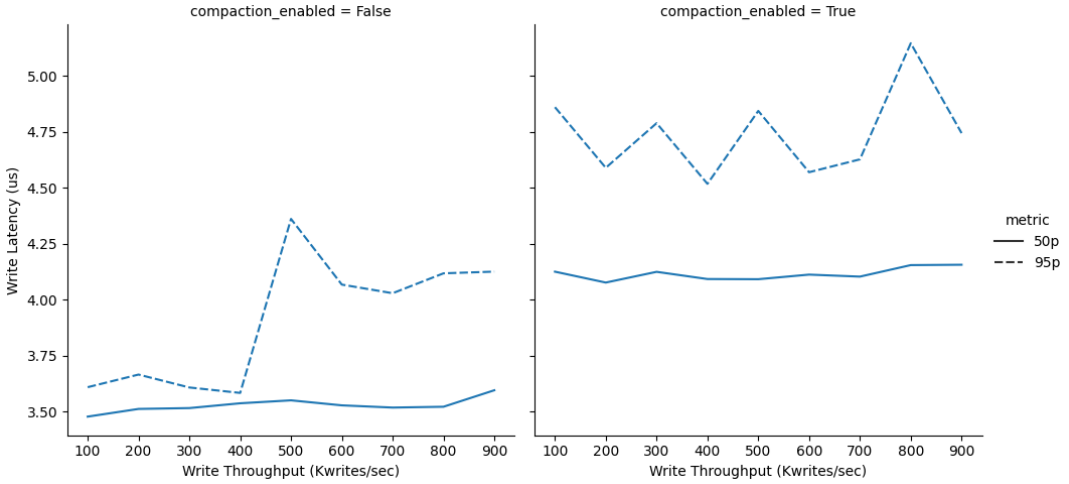


FIGURE 4.10: Write Latency vs Throughput, with Compaction disabled (left) and enabled (right).

From the experiment results in figures 4.10 and 4.11 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 neither for writes, which are in fact impaired, as compaction introduces a significant overhead. Therefore, compaction should be avoided in log-structuring.

4.1.3 AppendLog

In AppendLog we only have one tunable parameter, the threshold value, which is the maximum amount of bytes we can write to a runfile 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 4.12. As for the reads, they are not significantly affected, as expected.

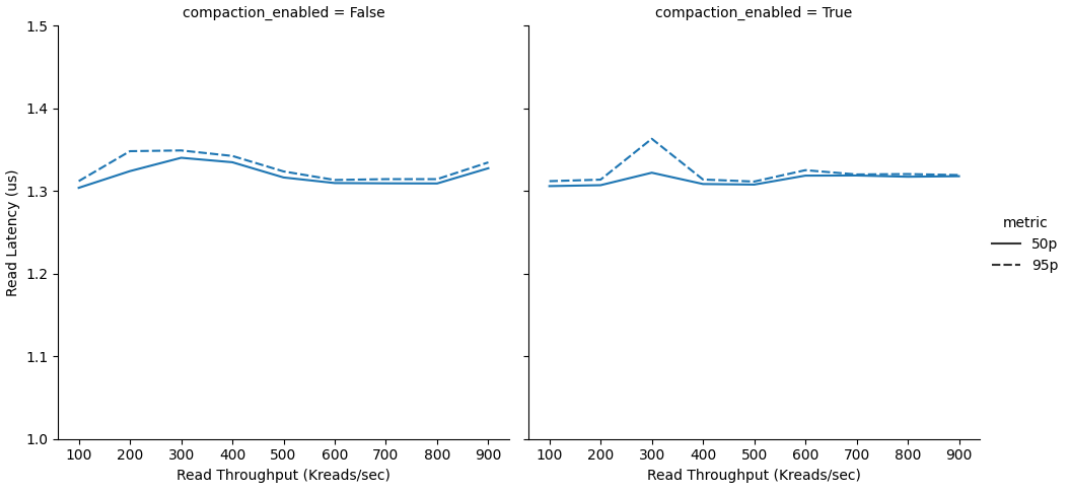


FIGURE 4.11: Read Latency vs Throughput, with Compaction disabled (left) and enabled (right).

Threshold

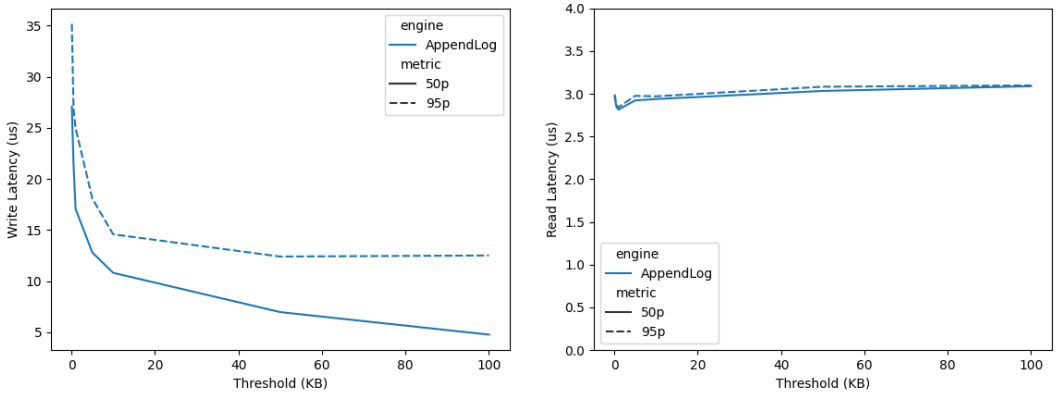


FIGURE 4.12: Latency vs Threshold.

4.2 Comparison

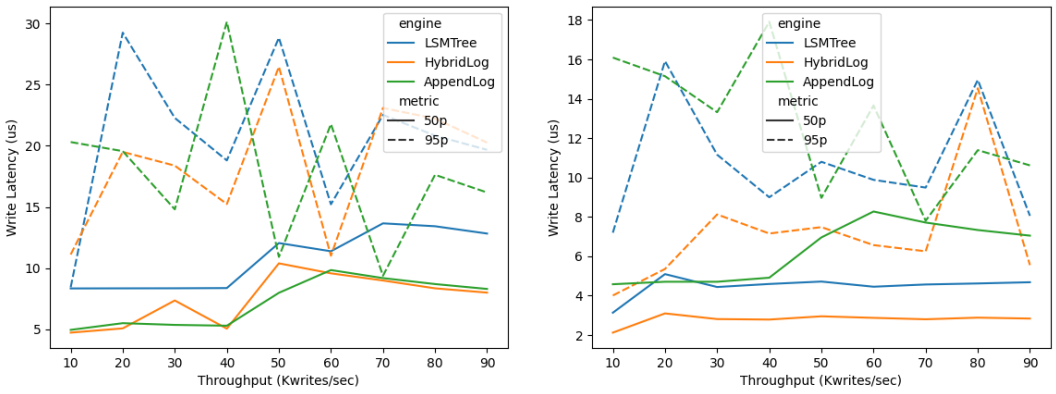
In this section we proceed to compare the engines on their performances when executing the same task with similar parameters. For the following experiments, we use the following parameters: Key and value lengths of 5 bytes each (so 10-byte key-value pairs), 10^5 unique keys and values, and 10 samples 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 `mem_segment_len`=210K, `ro_lag_interval`=10K, `flush_interval`=10K, and for the AppendLog `threshold`=100K.

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

4.2.1 Write Latencies

In figure 4.13 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 4.14.

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.



(A) Uniform distribution (B) Zipfian distribution

FIGURE 4.13: Latency vs Max Runs per Level.

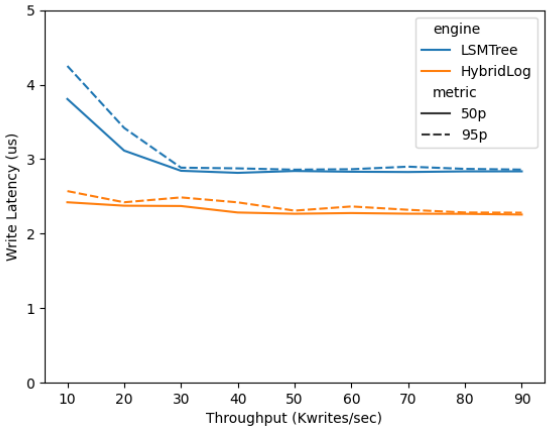


FIGURE 4.14: Write Throughtput when data fits the memory

4.2.2 Read Latencies

Upon examining the latencies for the reads in figure 4.15, it becomes clear that the HybridLog and AppendLog outperform the LSM-Tree by a large margin. This is because of their fast hash-based in-memory indices and minimal I/O.

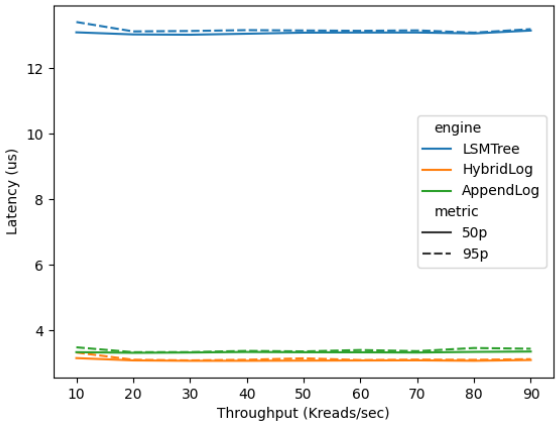


FIGURE 4.15: Read Latencies

4.2.3 Memory

HybridLog's superiority as the fastest key-value store comes at the cost of high memory usage, as can be seen in figure 4.16. 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 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.

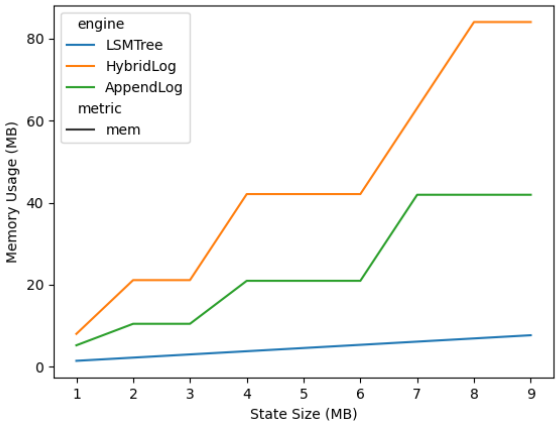


FIGURE 4.16: Memory Usage

4.3 Incremental Snapshotting

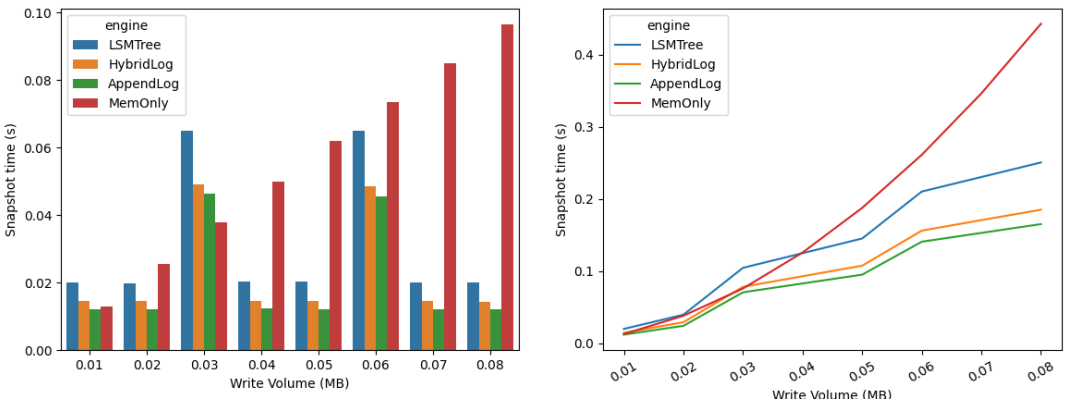
This section focuses on evaluating the incremental snapshotting capabilities of the three log-structured engines. Towards this goal, to demonstrate the advantage of having incremental snapshots, we compare the LSM-Tree, HybridLog and AppendLog to "MemOnly", which is a naive implementation of a key-value 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 4.17. 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.

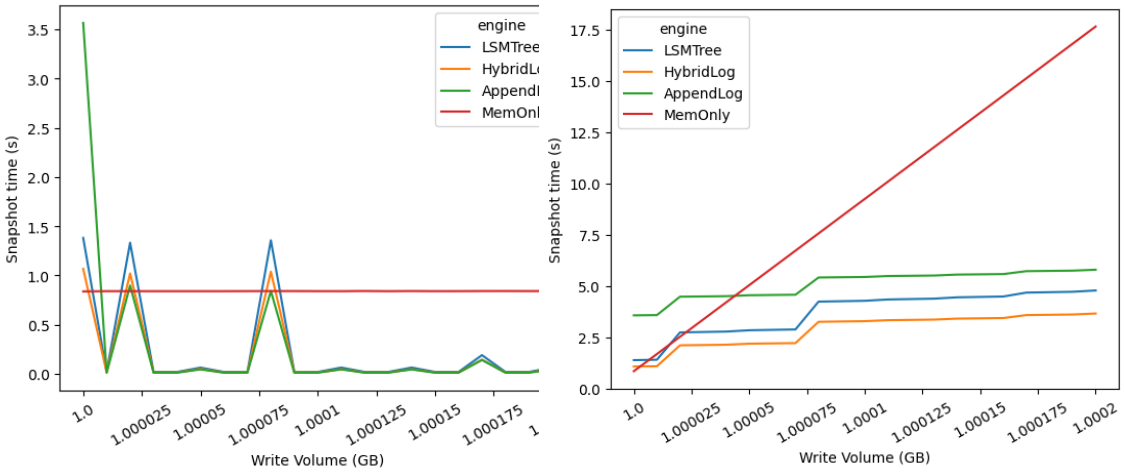
For the second experiment, where only updates take place, the results can be seen in figure 4.18. Again, as expected, the LSM-Tree, HybridLog and AppendLog only push the updates, while the MemOnly store pushes the whole state every time. By observing the cumulative graph, it is evident that the log-structured stores take snapshots more efficiently than the naive method.



(A) Discrete (B) Cumulative

FIGURE 4.17: Snapshotting Time vs Write Volume, when we increase the state by inserting new records.

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.



(A) Discrete (B) Cumulative

FIGURE 4.18: Snapshotting Time vs Write Volume, when state stays the same and we only update it.

Chapter 5

Conclusion

5.1 Summary

5.2 Future Work

Appendix A

Code

A.1 Key-value store API

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

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

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

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

    def __sizeof__(self) -> int:
        pass

    def close(self) -> None:
        pass

    def snapshot(self) -> None:
        pass

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

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

A.2 Replica API

```
class Replica:
    def put(self, filename: str) -> None:
        pass

    def get(self, filename: str, version: int = None):
        pass

    def gc(self):
        pass

    def restore(self, max_per_level: int, version: int = None):
        pass

    def destroy(self):
        pass
```

LISTING A.2: Replica API - method signatures.

```python
def expand_version(version: int, max_per_level: int) -> list[tuple[int, int]]:
    acc = []
    while version != 0:
        acc.append(version % max_per_level)
        version //= max_per_level

    levels_runs = []
    for i, e in enumerate(acc):
        j = e - 1
        while j >= 0:
            levels_runs.append((i, j))
            j -= 1

    return levels_runs
```

LISTING A.3: Algorithm used in snapshot rollback.

Bibliography

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.

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.

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.

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.

López, Pedro García et al. (2021). "Serverless Predictions: 2021-2030". In: *arXiv preprint arXiv:2104.03075*. URL: https://arxiv.org/pdf/2104.03075.pdf.

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.

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)*.

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.

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.