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 5, 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 four 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. We perform a comprehensive comparison among them, 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 rest of this thesis is structured as follows. In Chapter 2 we go through the related work. [...TODO] Chapter 3 contains extensive explanations of the internals of each type of key-value store and the implementation details and design decisions of each of them. [...TODO] In Chapter 4, we evaluate our implementations, performing benchmarks and comparisons between the key-value stores. We discuss the results and answer our reseach questions. Finally, in Chapter 5 we conclude by recapitulating the current work and proposing some directions for future reseach.

Chapter 2

Related Work

2.1 TODO

Chapter 3

Implementation

In this chapter, we will present some high-level design decisions that are common in all our implementations, then we will go through the internals and the implementation details of each of the key-value (KV) stores and finally present we leveraged log-structuring to fullfil the incremental snapshotting property we want our key-value store to have.

3.1 Common design decisions

Firstly, we designed our implementations to expose a common interface (API) to the user. This allows for easy benchmarking, testing, and ultimately a fair comparison between the engines. The API programmatically is defined in a parent class that the classes corresponding to each engines inherit and extend. Firstly, all the engines have a common API. This can be found in appendix A.

1. empty value == delet 2. binary keys and values because allows us to encode the length first for the encoding 3. disk binary encoding 4. data under a single directory 5. engine will rebuild from local files if found. if given remote replica, will restore the latest version by default. should be able to rollback

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, 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 file that we call a run.

3.3 AppendLog

3.4 HybridLog

3.5 Incremental Snapshots

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 behaviour of the respective engine, it is important to explore them visually.

4.1.1 Log-structured Merge-Tree

The first parameter of the LSM-Tree is `max_runs_per_level`. This controls the maximum amount of runs allowed in a level when log-structuring and will be more carefully examined in the subsection 4.1.4 as it is relevant to the other engines as well.

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.

4.1.2 HybridLog

4.1.3 AppendLog

4.1.4 Number of runs per level in Log-structuring

4.2 Comparison

4.3 Incremental Snapshotting

Chapter 5

Conclusion

5.1 Summary

5.2 Future Work

Appendix A

Code

A.1 Key-value store API

```
def get(key: bytes) -> bytes:
    # retrieves a value associated with the given key.
    # if empty

def set(key: bytes, value: bytes):

def __sizeof__(self):

def close(self):

def snapshot(self):

def restore(version: Optional[int] = None):
```

LISTING A.1: API function signatures.

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.

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].

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

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.

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.

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

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.