**Replacing Apache Impala’s statestore with etcd**

Ethan Xue

3B Work Term Report

September 2019

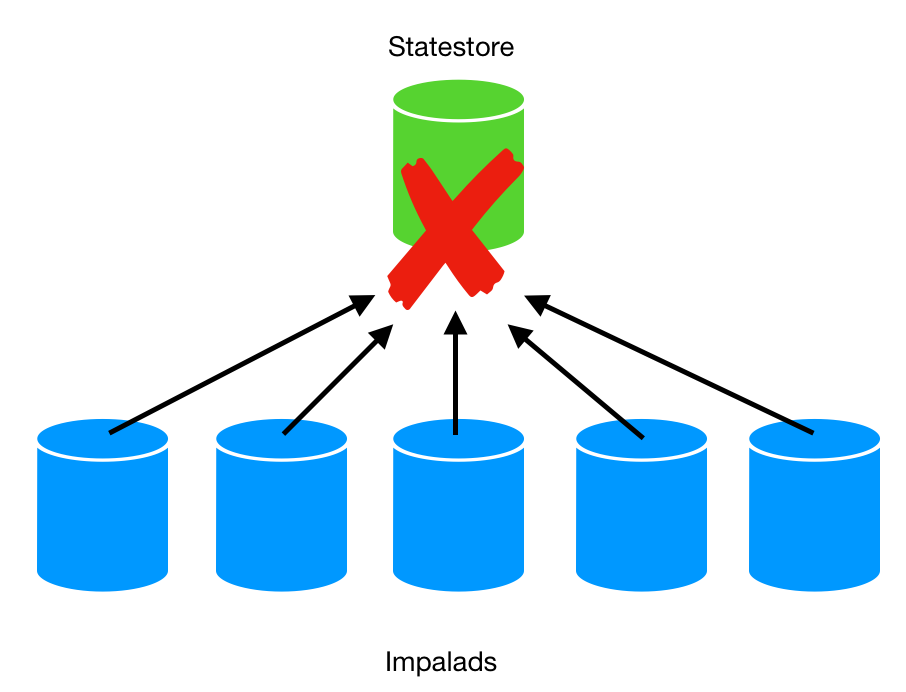
Relevant courses: CS343, SE464, CS451

**Context**

Apache Impala is a massively-parallel processing SQL query engine for Apache Hadoop written in C++ and Java. It also supports other forms of storage such as Apache Kudu and Amazon Simple Storage Service (S3). Impala was developed at Cloudera in 2013 as an open-source project inspired by Google F1. The link to the project can be found here: <https://github.com/apache/impala>. As an intern on the Impala team at Cloudera this summer (2019), I had the privilege of choosing an intern project to work on for four months from a list of options. After much thought, I decided to take on the challenge of prototyping replacing Impala’s statestore with an alternative system. This was a fairly research-oriented project, as I had to understand the existing flaws with Impala’s statestore, conduct research on a number of viable alternatives, and create a prototype of Impala with the statestore replaced with the solution. I chose this project because I hoped that it would give me an opportunity to learn about distributed systems and I am pleased to say that my learning expectations were fully met.

The main design decision I had to make was choosing an appropriate replacement for Impala’s statestore that 1. Satisfies outlined design constraints and 2. Is a more fitting solution than other alternative systems. My research led me to choosing etcd, an open-source project that I will spend much of this report dissecting.

First and foremost, what is Impala’s statestore and why are we trying to swap it with something else? Impala is a distributed system, meaning that it consists of a cluster of computers (nodes) working together to accomplish a task: SQL querying. Each node runs an Impala process, called an Impalad. Because of this architecture, it is important that Impalads have access to up-to-date snapshots of metadata at the cluster-level. For example, cluster membership metadata should be available to all Impalads, so that each Impalad can identify all the other Impalads in its cluster. The statestore is a critical component of Impala because it manages the coordination and synchronization of cluster-wide metadata for Impalads. From a high level, the role of the statestore is to receive metadata from Impalads, store them internally, and then broadcast metadata to other Impalads so that they can stay have the latest information.

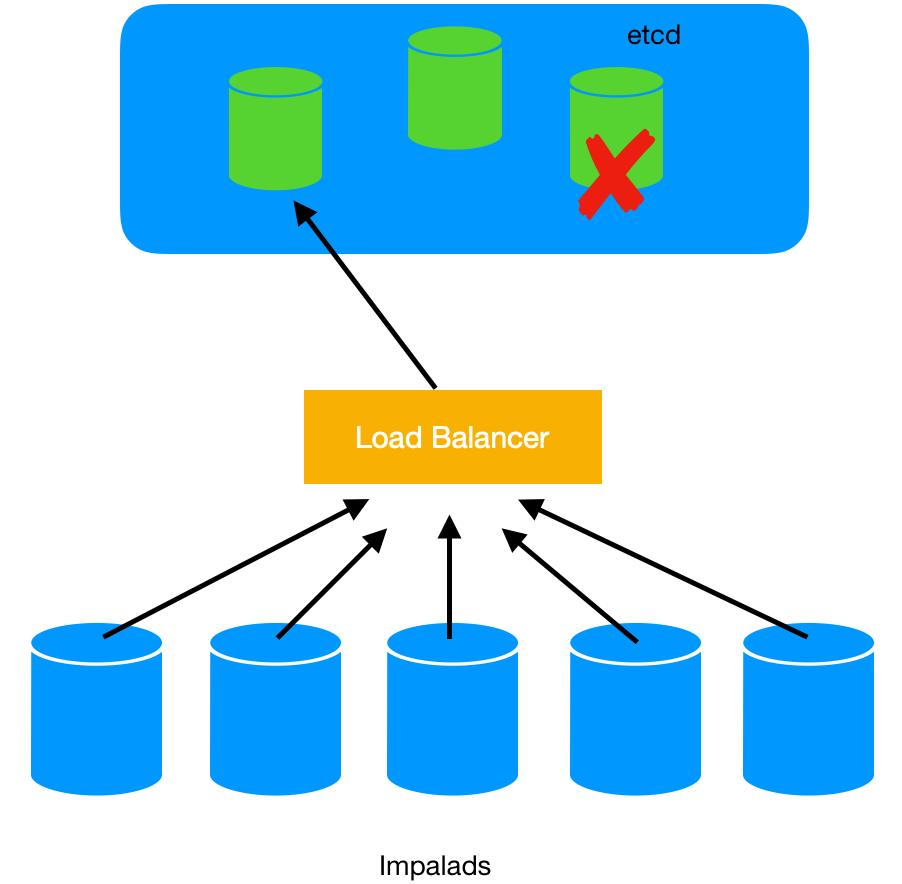


The fundamental problem with the statestore is its lack of fault-tolerance. Simply put, fault-tolerance is the ability of a system to *do its job* even when *things go wrong*. There is a single statestore daemon process running for each Impala cluster –this creates a single point of failure. If the node running the statestore goes down due to a software or hardware malfunction, cluster-wide metadata could become inconsistent as there is no longer a way for metadata to move around between Impalads. If this happens, users might observe undefined behaviour, which is clearly not ideal. Another limitation of the statestore is that it is entirely managed by Impala contributors, who have to sacrifice engineering effort to test and maintain it. It would be nice if we could leverage an existing open-source project that meets all of the required design constraints for being a solid replacement for the statestore: high availability and fault-tolerance, high consistency, high reusability, and great reputation.

**Engineering Decision**

The best solution to the problem that satisfies all design constraints is etcd. Etcd is an open-source, distributed key-value store written in Go offering strong consistency and high availability. The link to the repository can be found here: <https://github.com/etcd-io/etcd>. The choice for using etcd was first suggested by a colleague, who read about it online. My goal was to understand the benefits of using etcd to see if it was a good fit, and to compare it to potential alternative datastores. Below are the findings of my research in how etcd satisfies the design constraints.

Etcd has high fault-tolerance. Being a distributed system, etcd is deployed as a cluster in production. The recommended cluster size according to the official docs is three or five. Each node in the cluster has a full replica of all the key-value pairs, meaning that even if one of the nodes goes down, there will be no data loss and etcd can continue to work properly through the alive nodes. This attribute holds true unless the etcd cluster loses quorum. Losing quorum means that a cluster of n members has fewer than (n / 2) + 1 members alive, and this caveat is due to the Raft consensus algorithm. For this reason, a separate health-check service can be deployed to restart an etcd node if it goes down, further making etcd less susceptible to complete failure, as compared to the statestore.



Etcd supports strong consistency through the Raft consensus algorithm. Strong consistency means that if you write data to etcd, and the operation is successful, then you are guaranteed to read the changes just made from any node in the cluster. This attribute is crucial, as cluster-wide metadata stored in etcd must always be in a valid state to avoid undefined behaviour. If there was no strong consistency, an Impalad might read the old value of a key-value pair that was recently updated. The concept of consistency applies to distributed systems, so it not applicable to the statestore. However, etcd’s strong consistency is what gives it an edge over other candidate datastores, such as Redis.

Etcd is highly reusable. It has a simple application programming interface, making it flexible for clients to decide on how to use the key-value store in a way that best suits their needs. Reusability is important because we can use etcd to store other types of data, not just those related to cluster coordination such as cluster membership metadata.

Etcd has proven reliability. Not only is etcd backed by a large community of developers on Github, but it is also used by well-known open-source projects such as Kubernetes, which uses etcd to store all of its cluster configuration data, and companies such as Huawei Technologies. On a side note, one major advantage of etcd is that we get it *for free* since Cloudera uses Kubernetes with Impala in production.

Besides etcd, there were other popular databases I considered for being potential candidates for replacing the statestore. These databases include: Redis, Apache Zookeeper, NewSQL databases like Google Spanner and CockroachDB, and Apache Kafka. Below are my concrete justifications for why etcd has the edge over each of them with regards to storing Impala cluster metadata.

Redis: Both etcd and Redis can store key-value data, but they have very different architectures and use-cases. Etcd persists all data to disk, whereas Redis uses an in-memory data structure to allow for fast querying. Although perhaps not as fast, the reason why etcd is more suitable is because of its high availability. In our case, the metadata we are storing is critical to the proper functioning of the Impala cluster, so we would rather sacrifice some performance than availability so that we can avoid headaches. Also, unlike etcd, Redis is not strongly consistent out of the box.

Apache Zookeeper: Although both Zookeeper and etcd try to solve the same problem of distributed system coordination, etcd is by definition better than Zookeeper. This is because etcd’s design was inspired by Zookeeper as it was built after it and used the engineering experience and feedback from Zookeeper’s design to build its own features that were better. Etcd made improvements over Zookeeper, such as dynamic cluster membership reconfiguration (allowing nodes to be added/removed dynamically through an API), reliable key monitoring, stable read/write under high load, and others.

NewSQL Databases: Both etcd and NewSQL databases have strong consistency guarantees, but they have different performance characteristics. NewSQL databases are designed to store data on the order of terabytes that can be queried with rich SQL syntax. They are optimized for horizontal scaling across data centers, support sharding, etc. which brings about unnecessary complexity and overhead for our use case, because we simply want to store cluster metadata, which is only a few gigabytes. Etcd is optimized for storing gigabytes of data with simple structure, so we can achieve lower latency by using etcd.

Apache Kafka: One of my colleagues asked me if Kafka could be used instead. This might seem like a possibility to some, considering both the statestore and Kafka follow a publish-subscribe model. However, Kafka would definitely not be appropriate to use, because it was made for a completely different purpose: streaming data. Furthermore, like NewSQL, it is also designed to be massively scalable, which incurs a lot of overhead for our use case.

Finally, if we do a complete comparison between etcd and the statestore, we can see all the reasons why etcd is a better fit for Impala. Etcd is distributed so even if one etcd process fails, Impalads can continue to store and retrieve cluster metadata. Again, this would not happen with the statestore, because it is only a single process. Secondly, since etcd is open-source, adding it will take the onus off of Impala contributors to test and maintain the statestore. This is a huge bonus, because the statestore code is quite complex and difficult to test. Also, etcd has an additional benefit over the statestore, which is an inherit property of its distributed nature: scalability. Although this was not of the major design constraints, etcd is more scalable than the statestore because it supports a higher read throughput. This makes sense intuitively: A Impalad can read from *any* etcd node in the cluster because of the replication and consistency guarantees etcd support, whereas the statestore experiences a higher load on average for reads, since all of them are directed to the single statestore process. Another bonus feature of etcd is that it comes with built-in metric support and debugging. For the statestore, developers had to implement these manually. For metrics, each etcd node exposes an http endpoint with a plethora of various statistics that can be used to monitor its health and performance. For debugging, each etcd has a debug webpage that displays the history of RPC calls made by clients.

Reflecting on my work, I had to apply a lot of theoretical knowledge I acquired on the fly about distributed systems in order to complete my project. I started off the internship with no background in this area, so I had to learn as much as possible through books, such as “Designing Data-Intensive Applications” by Martin Kleppmann, and papers such as “Impala: A Modern, Open-Source SQL Engine for Hadoop” by Marcel Kornacker et al., and “In Search of an Understandable Consensus Algorithm” by Diego Ongaro and John Ousterhout. These resources helped me better maneuver through the etcd documentation and consider different failure modes when creating my prototype. I am also very thankful for having taken SE350 (operating systems) at Waterloo, as it gave me a solid understanding of operating systems that I was able to apply, because my project was quite systems-heavy, and CS247 (software engineering principles), because it taught me best practices for C++ development.

**Conclusion**

I am quite satisfied with the decision I made in choosing to replace the statestore with etcd. After completing a basic prototype, I always already able to observe some of the expected improvements in Impala, such as better availability and resiliency. This was measured through end-to-end test cases I wrote in Python, that simulated various failure modes (such as an etcd node going down and then restarting) and other situations. Although I created a prototype, I am hopeful that my project will eventually be deployed to production and impact the many enterprises using Impala on a large-scale, by giving them greater confidence that the tool they are using to store, query, and process their data has a higher ability to tolerate faults. With regards to my learnings from this project and specifically, from researching etcd and how it satisfies design constraints, I definitely had an extraordinary and fulfilling experience. I am looking forward to having an easier time with future courses at Waterloo, such as CS348 (databases) and CS454 (distributed systems) as I have already gained exposure to related concepts.