Marie Petitjean

A Simple Distributed Key-Value Store in Go

Introduction

A Key-Value store or KV store is storage system which stores data by associating a unique identifier, known as the key, with a value. This is different from the structure of relational databases, where data is stored in tables which are made up of rows and columns. Without the overhead of a tradition relational database, KV stores are optimized for fast read and write operations. This makes them ideal for web applications that often store data specific to a user, such as session details, recommendations, or preferences. For example, Amazon uses their KV Store, Dynamo, to provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog.

However, at Amazon's scale, these systems become more data-intensive than compute-intensive so considerations about reliability, scalability, and maintainability become critical. One approach that a KV store could increase the amount of load that it can handle is by increasing the amount of compute power, by increasing CPU and RAM of the machine. But at the scale of Amazon, where the amount data is quickly increasing every day, the applications become more data-intensive than computation-intensive, so this kind of scaling will only work to a certain point. The other approach is horizontal scaling where we add multiple machines and divide the work among them. In a distributed KV store, there are many machines which each are responsible for certain portions of the key space. This allows us to horizontally scale by adding or removing machines, and repartitioning keys, as needed to meet the load.

This paper will describe a very simple distributed KV store implemented in Go. It will describe the design considerations including sharding and replication and how they relate to the designs of more complex KV stores such as Dynamo, Cassandra, and Voldemort. Next it will describe the implementation details, usage, and go over some benchmarks to analyze its performance. Finally, we will conclude with a summary of what we learned during this project, some challenges, and future improvements.

Design Considerations

Storage

The main objective of this project is to focus on building a distributed KV store and to get a better understanding of the considerations and implementation details of creating a KV store that is distributed. As a result, we decided not to implement the details of a KV store and to instead use one that is already established and well implemented in Go. In this decision we considered two options, BoltDB and BadgerDB.

BoltDB is a simple KV store which is architecturally similar to LMDB and uses B+ trees, has ACID semantics, and fully serializable transactions. While LMDB focuses on raw performance, allowing unsafe operations like direct writes as a tradeoff for performance, BoltDB focuses on simplicity and does not allow these operations. Like most KV stores, BoltDB is good for read intensive workloads, however it struggles with high write throughput.

BadgerDB is a more recent KV store implemented in Go that uses LSM trees and has ACID semantics. It is based off of WiscKey which is a KV store that is highly SSD optimized and has both high read and write throughput. Since BadgerDB uses Log-structure merge (LSM) trees

it means that it can achieve higher write throughput because the foreground writes occur in memory and all the background writes have a sequential access pattern compared to BoltDB which use B+ trees, which can have multiple random disk writes for small updates.

From an implementation perspective, BoltDB and BadgerDB have very similar usage. It was also an important factor to consider that BoltDB is currently stable, but an archived repository, whereas BadgerDB is recent and is still being supported. A nice feature provided by BoltDB is that it allows the use of Buckets, which are collections of KV pairs in the database, which provides an easy extension to the implementation of replication, as described later in the paper. With these considerations, we decided to implement the KV store such that it could use either BadgerDB or BoltDB, depending on user input. This allows the user to make the decision about which underlying KV store best suits their needs and allows us to learn from benchmark tests of an KV store implemented with LSM trees compared to one with B+ trees.

Sharding

In a distributed KV store, sharding refers to breaking up the data into multiple partitions. This means that each partition could be on its own disk and the query load can be distributed across many processors, allowing us to add or remove partitions as needed depending on our load.

A simple method to partition is by the hash of the key. By taking the hash of the key, which is represented as a string, we are able to acquire some number between 0 and 2^{32}-1, which we can then mod by N, where N is the number of partitions, to get which partition a specific key should go to. The benefit of hash functions is that it can prevent skew and hotspots because it uniformly distributes the keys over the range. This implementation of sharding is also extremely simple to understand and implement. However, one downside is that it is much more difficult to do efficient range queries because even if the key themselves are similar to each other, the hashes can be very different, so they are likely to no longer be in the same partition.

A more significant challenge with partitioning by hash of key in the method described above is that the introduction or removal of a partition is more expensive. Resharding is an important part of a distributed KV store because it is fundamentally affected by increasing the number of CPUs or Disks as the load increases or as the amount of data increases. When we introduce an additional partition, our new method of assigning which key gets assigned which partition becomes hash(key) *mod* (N+1). As a result, a significant amount of keys will need to be moved from one node to another node, which is expensive. For example, if we have 10 partitions and a key hash of 123456 then it will be assigned to partition 123456 *mod* 10 = 6. However, if we introduce another partition, then after resharding the new partition will be 123456 *mod* 11 = 3. Therefore, this key will have to be removed from partition 6 and added to partition 3.

A simple extension of this design is described in Voldemort's approach to partitioning. Instead of assigning each node a single partition, they create many more partitions than nodes and assign each node several partitions. As a result, when an additional node is introduced, it can take partitions from each existing node and the partitions become redistributed. When a node is removed, it redistributes its partitions to the existing nodes. A difficulty with this approach is that the number of partitions is fixed and it is difficult to choose what the "correct" number of partitions is to find the best balance while the size of the database may be dynamic.

A common approach that avoids these problems is to use consistent hashing and can be found in many distributed KV stores including Dynamo, Cassandra, and Voldemort. With consistent hashing there exists a ring which has the output range of the hash function where the largest hash value wraps to the smallest hash value. Each node is then assigned a position on the ring at random. Each key is assigned to a node by hashing the key to find its position and then walking the ring clockwise to find the first node with a position larger than the key's position. The main benefit of consistent hashing is that adding and removing nodes only affects the immediate neighbors so only a small number of keys need to be relocated to a new node compared to the methods presented above. Some drawbacks exist with consistent hashing, such as non-uniform data and load distribution, however Dynamo and Cassandra each present solutions to these problems through either creating virtual nodes or analyzing load information of the ring.

Using consistent hashing is clearly the best solution for sharding, however it is much more complicated to implement than the simple modulo approach. Since the goal of this project is to create a simple KV store, we will opt for the simplest approach where each node is assigned a single partition, and each key is assigned a partition by finding hash(key) *mod* (N+1) where N is the number of nodes. Future iterations of this project could implement the two methods described above.

Replication

Replication describes storing copies of the same data on multiple different machines. In the context of our simple KV store, this allows there to be multiple copies of the data in case there was a failure with one machine causing it to lose a copy, and to help serve read queries. The simplest approach for replication is to use a leader and follower approach. In this approach, one of the replicas is designated as the leader and when a put request is sent then they must be sent to this leader which has permission to write the data. The other replicas are followers and are read only. When the leader writes new data then it also sends the data change to the followers, usually as part of a replication log. The followers use the replication log to update its local copy of the data and apply writes in the same order as they were processed by the leader. In this simple KV store we will use this basic approach with some modifications to better suit our implementation, which will be described in the implementation section.

Implementation

This simple distributed KV store is implemented entirely in Go and comprises of 4 main components: the API, the configurations, the database, and replication.

API

The code to spin up a node is found in `/cmd/kvstore/main.go`. Creating a node requires you to pass through the *db-location* which is the file path for the db, the *http-address* which describes the host address for this node, the *config-file* which describes the path of the config file, the *db-type* which specifies whether to use BoltDB or BadgerDB, the *shard* which describes the name of the shard or node, and *replica* which indicates whether or not this node is a replica. For a replica, the shard name should be the same as the master shard that it is a replica for.

Marie Petitjean

The client interacts with each node through 3 http endpoints, where the handlers are implemented in `/pkg/api/api.go`. The first endpoint `/put?key=some-key&value=some-value` is used to put a key and value pair into the KV store. The second endpoint `/get?key=some-key` is used to get a key from the KV store. If the key is found then it will return the value, otherwise it will return an empty string to indicate that the key was not found. If the key is not located on the shard that the request was made to, then it redirects the request to the master of the correct shard.

Resharding

The third http endpoint is `/clean` which is used for resharding. In order for the client to add an addition shard, or node, they must shut down all the currently running shards. From our design considerations above, we found that introducing just one shard with the hash(key) *mod* N method can lead to a large number of keys having to be moved to another shard. A very small modification that we can make is considering only having powers of 2 of shards. Doing this will result in a predictable pattern of keys being relocated from one shard to another. For example, 1234567 *mod* 2 is 1, so this key will belong to shard 1 if we only had two shards (shard 0 and shard 1). However, if we were to introduce 2 more shards, to get the next power of 2, which is 4, then we get that 1234567 *mod* 4 is 3, so this key will have to be relocated to shard 3. We are able to find that keys in shard 1 will only be relocated to shard 3, if they need to be relocated, and keys in shard 0 will only be relocated to shard 1, if they need to be relocated.

Since we know where the keys will be relocated if they need to be relocated then we can perform the following process when we need to add additional shards. Take down the http servers for each of the current nodes. Modify the *config.yaml* file to contain the new shards, where the number of shards is the next power of 2. Determine which shards are associated to each other with the additional shards, and copy the database so that they are the same. For example, copying the database of shard 0 to shard 2 and the database of shard 1 to shard 3. Spin up all the shards and run `/clean` on the all the shards, this will remove all of the keys that do not belong to this shard. While this approach is extremely simple, there are quite a few limitations which we will discuss in the future work section of this paper.

Configurations

To make this KV easier to use for the client, we provide a config file to describe each of the shard. The config file, `/config.yaml` is a simple yaml file where each Shard object is described in a separate yaml file, they are put together in a single file with the `---` separator. Each Shard object describes the Name, which should match the *shard* flag when spinning up the HTTP server. It also describes the index, which we will use to uniquely identify each shard, the address which is the same as the *http-address* flag of the http server, and a list of replicas which have the http address of all this master node's replica followers. For example, the following would be a system that has two shards, each that has one replica. It is important that indexes are monotonically increasing and that no index is skipped, they do not necessarily need to be in the correct order in the yaml file though.

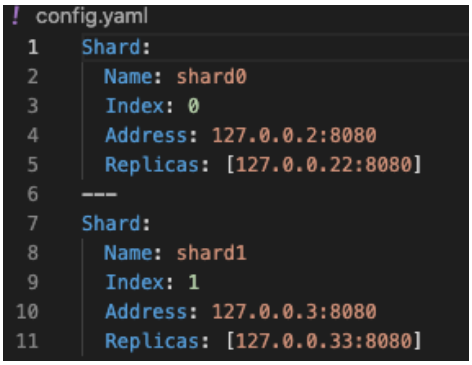

Figure 1: Valid config.yaml File

The Go file `/cmd/config/config.go` implements the logic of the parsing of this yaml file which is parsed every time a node is started, changes in the config file between nodes will cause problems. There are also unit tests implemented in `/cmd/config/config_test.go` to verify that the parsing of the file works as intended.

Database

The directory `/cmd/db/` contains several Go files which implements the logic that interacts with BoltDB and BadgerDB. In order to abstract away the implementation of the KV store from the http server itself, we created a *Database* interface in `/cmd/db/db.go` which specifies functions on the database, which are separately implemented by `/cmd/db/boltdb.go` for BoltDB and by `/cmd/db/badgerdb.go` for BadgerDB. This design is also allows us to easily implement the use of other KV stores in the future if we decide to do so.

The two most critical functions that are implemented are *PutKey(key string, value []byte) error* and *GetKey(key string) ([]byte, error)*. As their function describes, these functions are used to put a key/value pair into the KV store and to get a value for a given key.

To implement the resharding endpoint, the database also needs to implement *DeleteBulkKeys(deleteKey func(string) bool) error* which take a function *deleteKey* which takes a key and returns whether or not that key should be delete. The function goes through the keys the database and deletes those that should be deleted according to the function.

Replication

To implement the master, follower approach to replication we use the Buckets feature of BoltDB. Each bucket is a collection of key/value pairs within the database. However, since BadgerDB does not have any such feature, we decided not to implement replicas when using BadgerDB.

For each node we have two buckets, the first is the default bucket and the second is the replica bucket. For each master node, when it receives a write, it writes the key/value pair to both buckets. In each replica node, when you spin up the node it starts a go routine which is constantly running in the background, which is implemented in `/cmd/replication/replication.go`. The routine repeatedly makes a request to the `/get-next-replication-key` of the master node. The master node handles this request by getting the first key from the replica bucket and return its key and value. From the replica node, if the response of the request to the master node has a key/value pair then it puts that key/value pair into its, the replica's, default bucket. It then sends a request to the master node's endpoint `/delete-next-replication-key` with the key/value pair that it just put into its local

database. When the master receives this request, it verifies that the key/value pair matches a key/value pair in the replica bucket, and if it does then it deletes that key/value pair and sends a response back that it was ok or if there was an error. The replica node verifies that it was ok or if there was an error. If at any point there is an error then we delay the next iteration of this loop as to not overwhelm the master node, and then we try again.

Demo

For more information about how to use this simple distributed KV store, there is a *Readme.md* file which describes how to use it as well as a few simple demos. I have also attached a video to this assignment where I go through a quick demo of how it works.

Benchmarks

For our benchmarks, we implemented a small program in `/pkg/cmd/perf.go` which we can build with `make perf` and can run with flags which specify which node we want to benchmark, how many random read and write we want to do, and the number of go routines which we want to run in parallel.

Writes

When running 4 nodes, with BoltDB under the hood, we find that with 1000 write iterations with no concurrency gives us an average write of about 21.5ms and a QPS of 46.4. If we increase the number running in parallel such that we have 4 go routines each with 250 write iterations we find that the average write time increases to about 80ms and the QPS for each routine is about 12, so the total QPS across all of them is 49. We see that increasing the number of concurrent threads making queries does not increase the QPS. We are also able to see that when we are running the benchmark, the program `kvstore` is not very CPU intensive and instead the latency comes mostly from BoltDB doing disk writes. Since there are many disk writes, and the QPS does not increase with increased concurrency, we believe that BoltDB is not doing concurrent disk writes. We can greatly improve write throughput by setting `DB.NoSync` to true. In this case, when running 1000 iterations with no concurrency, the average right time is 259.829µs and the write QPS is 3848.7. When running 250 iterations with concurrency of 4, we get an average write time of 396.761µs and a total QPS of 10043.2 which is a significant improvement over a concurrency of 1. While this provides a significant improvement to write throughput, turning on `DB.NoSync` can be dangerous because it allows the OS/disk to rearrange writes to improve performance, which could lead to a corrupted database if there is some sort of failure.

When running 4 nodes with BadgerDB, we find that the average write time is much better than BoltDB without `DB.NoSync`. With 1000 iterations and no concurrency, the average write time is 159.358µs and has a QPS of 6275.1. With 250 iteration and a concurrency of 4, we find that the average write time is 279.18µs and has a QPS of 14304.8. However, we find that as we increase the concurrency, the QPS tops out at around 16000. As expected, BadgerDB performs better with write throughput than BoltDB.

Reads

If we first consider running 4 nodes with BoltDB with `DB.NoSync` turned off, then we find that with 1000 read iterations and no concurrency, the average read time is 155.675µs and

the total read QPS is 6423.6. If we have 250 iterations with a concurrency of 4 then we find that the average read time is 249.396µs and the total read QPS is 15802.5. So, the read throughput increases as we increase concurrency. Even with a high load of 1000 iterations and a concurrency of 4, the average read time is 240.664µs with a read QPS of 16493.6. In the same benchmark with `DB.NoSync` turned on, we find an average read time of 243.92µs and a QPS of 16507.6, so this does not affect read throughput as significantly as it affect write throughput.

When running 4 nodes with BadgerDB we find that with 1000 iterations and no concurrency, that the average read time is 156.716µs with a QPS of 6381.0. If we have 250 iterations with a concurrency of 4 then we find that the average read time is 225.096µs and the total read QPS is 17425.9. So, the read throughput increases as we increase concurrency. Even with a high load of 1000 iterations and a concurrency of 4, the average read time is 211.729µs with a read QPS of 18754.6. The read throughput and latencies with BadgerDB is very similar to BoltDB.

Conclusion

Future Improvements

In our approach to sharding, when we add additional shards then there is a lot of movement of keys to and from different shards and it requires that we iterate through all the keys to determine if they are in the right shard. This is very computationally expensive and not very efficient. It also requires that we take down all of the shards, which means we cannot serve any read or write requests during this time. It would be better to implement a strategy like the fixed number of partitions in Voldemort or even consistent hashing, which would reduce the number of keys which need to be relocated. Another possible improvement is to automate the process a little bit more for ease of use. Adding more shards requires a lot of work from the user, including modifying the config file, copying dating, and running an endpoint. It would be better to have some automated script in order to do this work where the user can simplify the number of shards that it wants to change to.

Another future improvement would be to implement handling failovers of master nodes. In this current implementation, while we have replicas, we do not use the replicas to handle the case where a master node fails. A possible approach to implementing this would be to determine when a master node has failed, and if we find that one has failed, to choose a new leader. There are several possible options to choose a new leader, one possible way is through an election process where we choose the replica with the most up to date data.

A final simple future improvement would be to allow read queries to be redirected to replica nodes as well. Currently, if you send a get request to a replica node and the key is on that node, then that node will respond to the request. However, if you send a get request to another master node or replica that does not have the key, then it redirects the request to the master node that contains the key. While this provides the user with the most up to date data, it may be the case that the user would rather have more read throughput with the tradeoff of getting some stale data. We could accommodate this by allowing redirects to also go to replicas, so the read-throughput should theoretically be higher.

Marie Petitjean

Summary

While this simple distributed KV store does not implement the features and designs of more complex distributed KV stores such as Dynamo and Cassandra, it served as a tool for thinking about the design considerations that go into building a distributed KV stores. Through the process of this project, I had to consider how to build a distributed KV store from scratch in Go in such a way that it would be simple to use and understand. It served as a great learning experience and has the ability to be the foundation of more complex designs.

References

Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev*. 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

"BadgerDB" github.com/dgraph-io/badger

"Bolt" github.com/boltdb/bolt

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev*. 41, 6 (December 2007), 205–220. https://doi.org/10.1145/1323293.1294281

Kleppmann, M. (2021). *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly.

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1, Article 5 (February 2017), 28 pages. https://doi.org/10.1145/3033273

"Project Voldemort Documentation" *project-voldemort.com*