DISCO: A DIstributed Strongly-COnsistent Key-Value Store on LevelDB

Kalyani Unnikrishnan Deepti Rajagopal Basava Kolagani Adil Ahmed

13th May 2022

Contents

1 Overview

We implement DISCO, a cloud-native distributed key-value store built on top of levelDB [4]. We preserve levelDB's consistency guarantees and make use of its inherent properties and features to offer different modes of operations to server administrators.

DISCO aims to be a highly customizable solution for various application workloads. For instance, if users needed to run various analytics dashboards that fetch data from specific applications - this data would need to be fetched in real-time, resulting in read-heavy workload patterns. If the application's data were stored in one location, this would be straightforward - but infeasible in a consistent hashing based system. We allow keys to be hashed based on fixed length prefixes to ensure similar data can be stored together, resulting in improvements of the order of hundreds of times in magnitude.

The system may also be used for write-heavy workloads such as logging, where high write volumes are observed with rare reads. We allow customizations that allow writes to be buffered before being committed, allowing significant boosts in write performance at the cost of some durability - which may be an acceptable trade-off in such a workload.

We use consistent hashing to distribute responsibility for keys and values across our nodes and use shared storage to store levelDB instances. In addition, each server stores a local copy of it's EFS levelDB instance for fast serving of reads. Our system is deployed on AWS EC2 [2] instances. We use Apache Zookeeper [5] for master elections and to store meta-data. AWS Elastic File System (EFS) [1] is used as the storage layer. We instrument the code to run custom failure scenarios and crash tests. We also run performance tests for various workloads to benchmark our system. We present the details of the design, testing and evaluations in this report.

2 Design

2.1 Architecture

DISCO is a strongly-consistent key-value store. Keys are split among nodes using consistent hashing [6]. Consistent hashing ensures that when a node joins or leaves, keys only need to be moved to/from one other server. Each server creates two levelDB instances, one running locally and one running on EFS shared storage. One of the storage nodes also serves as a master - for co-ordination and configuration management. We support three modes of operation - DEFAULT, LOCALITY, and WRITE to optimize for varying application workloads. The system operation is illustrated in Figure 1.

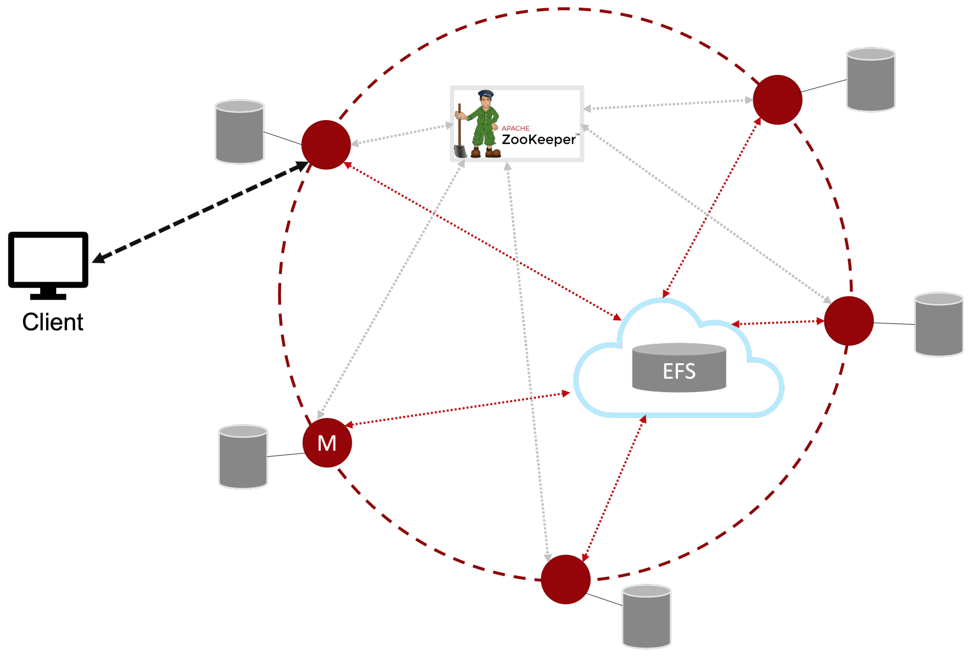


Figure 1: System Architecture.

2.2 Regular Operation

The API supports four operations:

1. `put(key, value)`
2. `get(key)`
3. `delete(key)`
4. `getrange(prefix)` (supported only in LOCALITY mode)

All keys and values are strings of variable length. The GETRANGE operation allows all keys beginning with a fixed prefix to be fetched in one operation, when the system is running in LOCALITY mode.

DISCO supports the three modes of operation described below:

1. DEFAULT: This is the baseline configuration. All `Put` operations are are asynchronously sent to the local and levelDB instances. `Get` operations are done from the local instance.

2. LOCALITY: In locality mode, we optimize for read-heavy workloads that exploit locality. Keys are hashed based on a fixed-length prefix with a configurable length. This allows keys beginning with a common prefix to be assigned to the same server, for faster sequential / batch reads. All writes are persisted to the local and EFS levelDB instances immediately. GETs and GETRANGEs are performed on the local instance. We envision that an organization can allow different application owners to configure fixed length prefixes for their applications. These prefixes can be injected and stripped by the application backend. This allows data of the order of hundreds of MBs to be fetched in a few seconds.

3. WRITE: This mode is optimized for write-heavy workloads in which reads are rare, such as logging. Writes are buffered at the node and are committed to EFS and to the local DB when a configurable write threshold is reached. All buffered writes are committed before serving a `get` call, to maintain strong consistency. Enabling this mode with a high threshold results in reduced durability, as the node may fail while the writes are buffered resulting in data loss.

The above configuration (mode, batch size and prefix length) is saved in a Zookeeper node and is read by each server on initialization.

2.3 Co-ordination between nodes

DISCO elects a master for co-ordination. The master is elected using an ephemeral `znode` on Zookeeper (cite) that contains it's IP address. The first server to successfully create this node is elected as the master. The master maintains a mapping of hash ranges to the nodes responsible for keys in those ranges. The master is responsible for broadcasting the updated map to all servers when a new node joins or an existing node exits the ring. The master is also responsible for detecting failures via heartbeats.

2.4 Node Joins

When a node joins the ring, it contacts the master and is assigned a server ID based on which it computes it's hash range. It then contacts it's successor, initiating a `Split` operation. The successor creates a levelDB instance on EFS for the new node, reads the required keys from its local DB copy and performs a single batched write to add the keys to the newly created EFS instance. The keys are then deleted from the local and EFS copies of the successor.

The new node then creates a local copy of its newly created EFS DB instance and begins normal operation.

2.5 Node Exits

A node failure is detected by the master when it misses five consecutive heartbeats. The master then calls a `Merge` operation, asking the failed node's successor to merge it's leveldb instance with that of the failed node. The successor reads all keys and values from the EFS copy of the failed node's DB and applies it in one batched write operation to it's own DB and to it's local cache. This is an expensive operation as it requires an unbounded number of network calls from DISCO to EFS to read in all the data. Operations to the key range of the failed node are unavailable during ongoing merges, to avoid consistency issues when data is updated just after being added to the batched write.

2.6 Piggybacking

The mapping of servers to key ranges is present on all servers. A client can send an operation to any server - the operation will be re-routed in one hop. The server map is also piggybacked in the response to the client, allowing it to eliminate the extra hop for future operations.

All writes to levelDB (both to EFS and locally) are performed asynchronously, which provides a 1000x speed improvement over synchronous writes [3]. Any buffered writes are synced to disk and EFS when a GET call is received on that particular node.

2.7 Consistency and Fault Tolerance

The system is designed to be robust and crash consistent under a majority of failure scenarios. We develop a consistency matrix to show fault injection at different points of the workflow, and the end result - as shown in Figure 2.

Crash during regular operation A crash is detected when a server misses five consecutive heartbeats. We choose to wait for five missed heartbeats as `Merge` is an expensive operation and we do not want to invoke it during a transient network partition. The client will retry the operation with the master, which will redirect the request to the successor upon successful completion of the merge.

Master Failure. The master creates an ephemeral znode in Zookeeper that contains it's IP address. Upon failure, this znode is deleted - thereby allowing another server to create the znode and take over as the master. A server detects the absence of the master when it fails to receive heartbeats for a 3 seconds, following which it attempts to create the znode and become the new master. If the node already exists, it reads in the IP of the master from the newly created file.

Incoming requests during Joins and Exits Incoming requests during `Split` and `Merge` operations could result in inconsistent results. For example, during a `Split` - a key might be updated at a node when it has already added that value to the batch that is sent to the new server. This is avoided by using a semaphore to prevent any incoming operations when a server is occupied with a `Split` operation.

If a request comes in during a `Split` procedure, the request is blocked and the client retries the request with the master. The master can now direct it to the correct node in one hop.

During a `merge` operation, the client is unable to contact the recently exited node, leading it to retry from the master. The master can then route the request to the appropriate server. Given that the master does not update the server map / hash table until the merge is complete, the client is prevented from reading stale data.

Possible Inconsistency In some rare conditions, a crash at the precisely the wrong time may leave the system in an inconsistent state as detailed below.

1. If a node (a successor) crashes while merging the database of another failed node, the master asks the next node in the ring to merge the first successor's database. At this time, some keys of the first failed node may be lost - leading to some data loss.

2. In WRITE mode, buffered writes at a server may be lost when a storage server crashes.

	INIT	GET	PUT	GETRANGE	SPLIT	MERGE
MASTER	✓	✓	✓*	✓	✓	✓
FOLLOWER	✓	✓	✓	✓	✓	✗

Figure 2: Fault Injection points vs Consistency Guarantees. The columns indicate the injection point of the fault. A tick indicates that the system is guaranteed to be consistent while a cross indicates that some data loss is possible.

3 Implementation

3.1 Frameworks, Libraries and Languages

We implement both our client and server in C++. We use the gRPC library for RPC calls between clients and server as well as between the servers themselves. We also implement a python wrapper of the C++ functions for ease of testing. Zookeeper is used for master election, configuration management and rendezvous (i.e. finding the master IP). We also allow clients to access Zookeeper to learn the identity of the master.

We use the `ctypes` library to build a python wrapper to invoke the client API. We used python to enable flexibility with advanced testing and evaluation scenarios.

3.2 Deployment

Our system was deployed on four `t2.micro` AWS EC2 instances, each with 1GB memory and 4GB storage space. A shared EFS volume was mounted across each instance.

3.3 Challenges

We encountered some interesting challenges over the course of this project effort:

- At each stage of the design, we sought to reduce every operation to a bounded number of RPC calls to enable fast performance. We were able to achieve this for all operations except `Merge` , which may result in an unbounded number of GET calls to EFS depending on the size of the database. Streamlining this approach required precise understanding of how levelDB stores details of its SSTables in MANIFEST files, which proved to be too bugggy and require several changes to the levelDB codebase. In the end, we chose to take the simple approach and rely on EFS's performance.
- We faced several issues with reachability for our Zookeeper instance and the servers themselves when deploying the applications in containers using AWS Elastic Container Service (ECS). We faced the same issue when deploying on EC2, resulting in us using a Cloudlab node to expose the Zookeeper instance.

4 Testing

We developed a robust stress-testing mechanism to check for correctness. In each correctness test, a sequence of PUT operations were performed. Then, GET requests for the corresponding PUTs were triggered in an infinite loop while a sequence of complicated operations such as node joins, node exits and master failures were triggered. Checksums were used to validate if an inconsistency occurred at any point during this sequence.

For evaluations, we use workloads with configurable Put and Get ratios, number of operations and sizes of keys and values. We run several different experiments and present our results in 5

5 Evaluation

We ran several benchmarks and conducted measurements to evaluate our final implementation.

5.1 Experimental Setup

Our experiments are conducted on AWS. We run the servers on four nodes, and the clients are spawned using threads on two nodes.

5.2 Impact of Local Copy

We measure the impact of local copy on various mixed workloads (x% writes, (100-x)% reads). As can be seen in Figure 3, when a server has to serve reads directly from EFS in the absence of a local copy, the latency of the request is considerably higher than when it responds to reads from its local levelDB instance due to the added cost of a network round-trip to EFS for each GET request. Additionally, it can be noted that as the ratio of `Put` operation increases, the latency per operation decreases. This is due to the fact that writes in levelDB are asynchronous, and are only externalized by the reads that are called. As the `Put` percentage increases and the `Get` percentage decreases, we observe lower latency due to the smaller number of reads that need to be externalized.

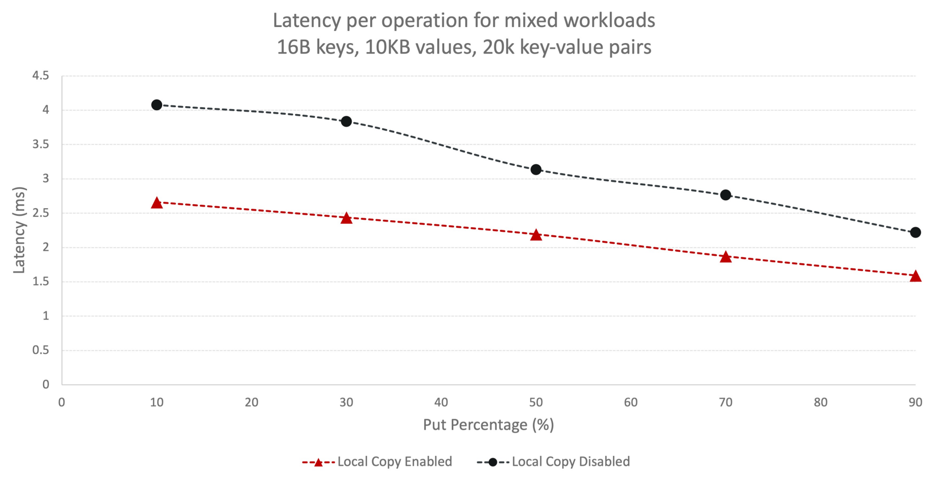
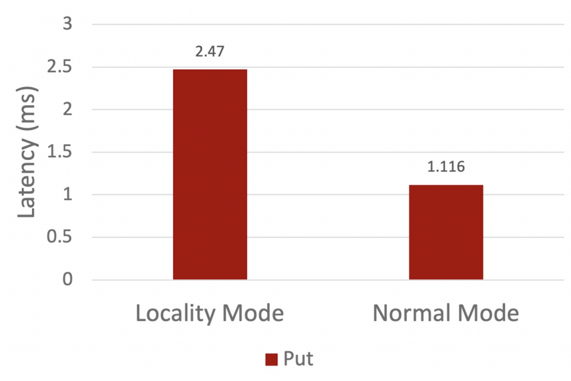


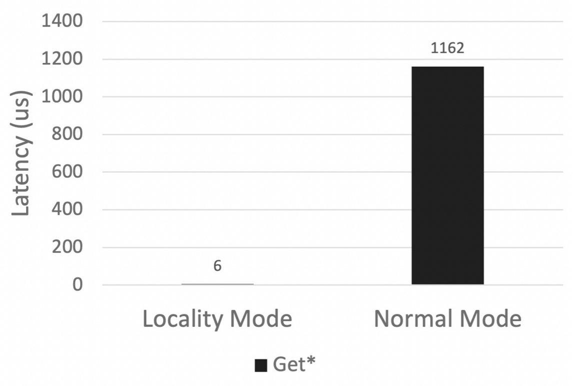
Figure 3: Impact of Local Copy

5.3 Performance in LOCALITY Mode

We measure the performance of `Puts` and `Gets` in a read-heavy workload with 10,000 key-value pairs, where each key is a 16 byte string, and each value is 100 bytes. As shown in Figures 4(a), and 4(b), the PUT latency for workloads with same key prefixes is considerably higher than that of the normal mode workload. This is because all the writes for the same prefix go to the same server, resulting in a bottleneck at this server. In contrast, the latency for a GETRANGE operation in such a read-heavy workload is significantly lower than using individual GET calls to fetch, resulting in a performance improvement of almost 200 times when compared to that in the normal mode.



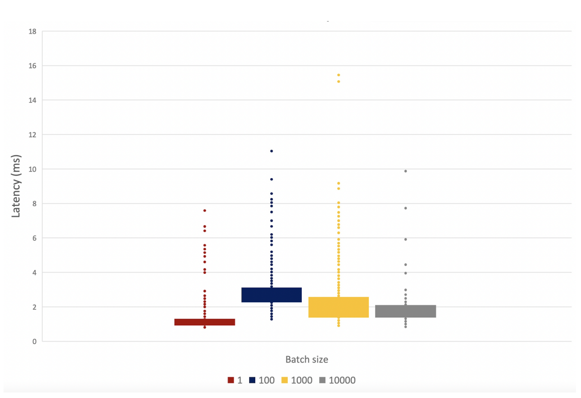
((a)) Put performance in LOCALITY mode vs Normal mode



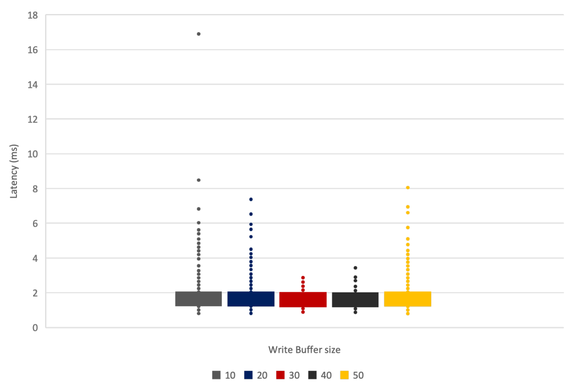
((b)) GETRANGE in LOCALITY mode vs GETs in Normal Mode

5.4 Performance in WRITE mode

In WRITE mode, because we write data in batches, we see an increased performance in the `Put` latency. As shown in Figure 5(b), we measure the performance of `Get` latency, which is supposed to be worsen with increased batch size. We find that, for small changes in the buffer size - there is no steady increase in latency for a `Get` call, indicating that the overhead of committing the buffer to disk is not too high and the real trade-off made here by using this mode is decreased durability and fault-tolerance.



((a)) Latencies measured with high variation in buffer size



((b)) GET latencies measured with steady increases in buffer size

6 Conclusion

We develop DISCO - a distributed key-value store based on levelDB which aims to exploit levelDB's native features as well as features offered by the cloud to offer a robust and performant storage option that can be configured to match multiple workload patterns. DISCO remains consistent during crashes and failures. We trade-off durability for faster performance and run comprehensive evaluations to measure performance and validate consistency. This project was a valuable exercise in implementing various concepts in distributed systems (e.g. elections, replication, consistency, cache consistency, co-ordination etc.) and coalesce it into a useful and performant system. We incorporated techniques picked up across past projects and have arrived at what we hope is a more mature and well-defined storage system compared to our prior attempts.

7 Demo Follow-up

During the demo, we received feedback on expressing all latency measurements as per-operation values rather than aggregate values, to convey more useful information to the user. Our analysis of the `Get` latency for write buffer operations was unclear on the exact design and takeaways from the experiment. We have adjusted our graphs accordingly, and re-designed our experiments to account for the feedback. We also added a conclusion section to the report, to reflect on the process and outcome of the project.

During the demo, we indicated that the system might be left in an inconsistent state if the master crashes during a `Merge` operation. On further analysis, we observe that this is not true. A new master is elected that replays the `Merge` operation as is - resulting in the system remaining consistent with no data loss.

References

[1] Aws efs. `https://aws.amazon.com/efs/`, .

[2] Aws ec2. `https://aws.amazon.com/ec2/`, .

[3] Leveldb documentation. `https://github.com/google/leveldb/blob/main/doc/index.md`.

[4] Leveldb. `https://github.com/google/leveldb`.

[5] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[6] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing

Machinery. isbn:0897918886. doi:10.1145/258533.258660. URL https://doi.org/10.1145/258533.258660.