# Hive: A Globally-Distributed Key/Value Store

Michaël Bonfils
*mbonfils@scaleway.com*

Patrik Cyvoct
*pcyvoct@scaleway.com*

Florian Florensa
*fflorensa@scaleway.com*

Eric Gouyer
*egouyer@scaleway.com*

Mickaël Grégori
*mgregori@scaleway.com*

Stanislas Lange
*slange@scaleway.com*

Nicolas Sampré
*nsampre@scaleway.com*

Quentin Selle
*qselle@scaleway.com*

Louis Solofrizzo
*lsolofrizzo@scaleway.com*

*Scaleway Storage Team* - 2021

## Abstract

This paper reports our experience creating, developing, and deploying a globally distributed key-value store intended as a database backend for our S3 API, Hive. Hive is a system to distribute data on a global scale, with various desired consistency, replication, and database sharding for linear read and write latency.

## 1 INTRODUCTION

Scaleway is a European cloud provider that runs a large number of cloud-oriented products. One of these products is Scaleway Object Storage, a storage API based on Amazon's S3. There are strict operational requirements on Scaleway's production in terms of reliability, performance, and efficiency, and to support continuous growth, any platform needs to be highly scalable. Reliability is one of the essential requirements because even the slightest outage has significant financial and trust consequences.

This project was born to meet the reliability and scaling needs of the Scaleway S3 product. We needed a platform to scale up to millions of different databases with billions of entries while maintaining storage client separation, good latency, and performance. We also needed to build a reliable, consistent, and flexible platform. Indeed, in some regions, we can have multi-data centers replication, while in other regions, we do not have the housing required for such operations.

Hive is a scalable, globally-distributed database designed, built, and deployed at Scaleway. At the highest level of abstraction, it is a database that stores key/value pairs and shards data across many RAFT [1] clusters, which are also called RAFT groups. Replication is used for global availability and geographic locality; clients automatically failover between replicas of a database. Hive is designed to scale up to thousands of machines across multiple data centers worldwide and billions of database entries.

As any cloud provider, dealing with failures in an infrastructure comprised of millions of components is a standard mode of operation; there are always a significant number of failures at any given time. As such, Hive was designed to treat failures as the typical case without impacting availability or performance.

Hive is not meant to become a general-purpose database, like Redis or PostgresSQL. It is a highly optimized key-value store, which works well for a few specific operations. That being said, it is still a key-value store to be used as-is for various applications but without performance improvements.

This paper is structured as follows. Section 2 presents a global overview of the design of Hive, Section 3 presents the integration within the existing S3 product, and Section 4 presents benchmarks and evaluation of the database. In section 5, we go into details about the future work we want to achieve with Hive.

## 2 HIVE

Hive is a key-value store currently deployed at Scaleway as the database engine for the Scaleway S3 product. This section provides background on Hive's design, consistency and data repartition.

Clients can use Hive to store data safely, with specific optimizations for specific access patterns. Depending on the use case, a client can choose a consistency per read or write request; for a DNS database engine, eventual consistency might be preferable to have low write latencies, but for an S3 engine, strong consistency is paramount. Rather than creating a generic database engine with a query language, we created specific storage engines optimized for the use-cases they are dedicated to.

The main client of Hive is the Scaleway Object Storage product, which stores a reference to objects in the database and uses it for specific S3 operations, like versioning, delimiter, or prefix listing. It also uses consistency features from Hive to ensure bucket unicity worldwide and strong consistency multi-datacenter replication to ensure safety. A significant point of the trouble of our previous S3 database architecture was database sharding, as the databases were growing larger, impacting latency and even replication in some cases. Hive solves this issue by splitting S3 objects lexicographically among many shards, automatically splitting and merging shards when needed.

Hive uses the RAFT [1] quorum protocol to ensure consistency and replication for a shard, though with some modifications. To not have one big RAFT cluster, we have split all the shards into RAFT groups, which are RAFT state machines dedicated to specific data sets. This also enables us to avoid catastrophic failures in a quorum fail or some other internal error.

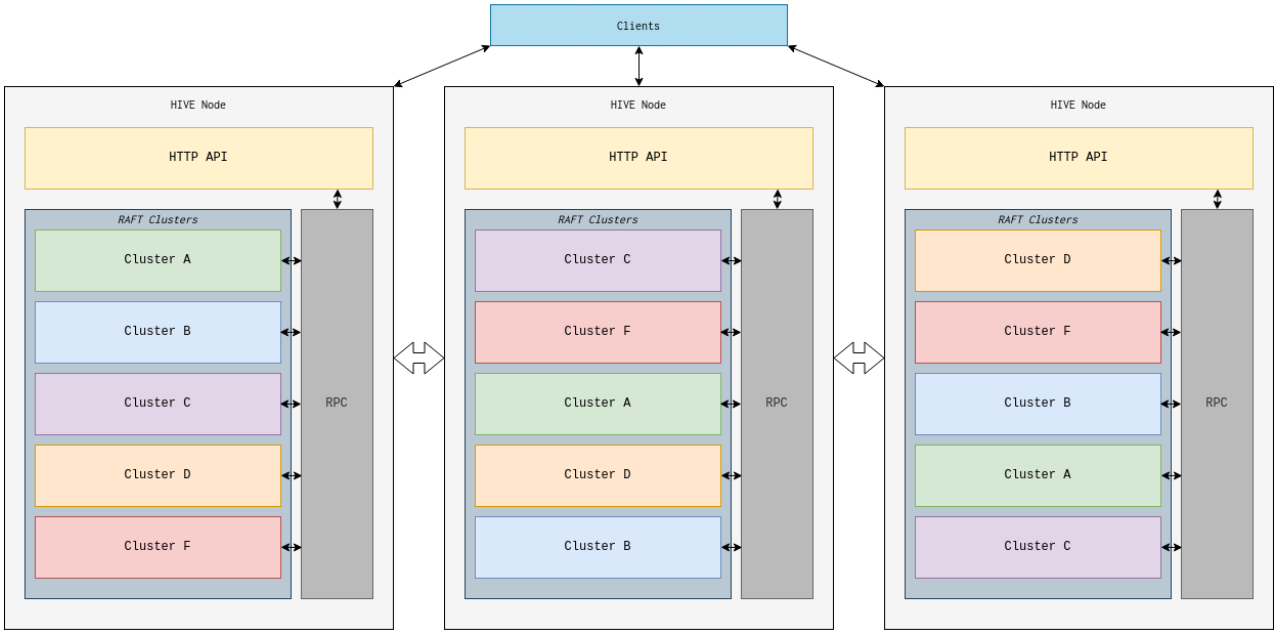


Figure 1: Hive bird's eye view

```
size=5 location=fr-par failure_domain=az consistency=strong read_consistency=strong
```

Figure 2: An allocation rule for a Hive cluster

## 2.1 Design Overview

Hive is composed of many different nodes on different machines, data centers, and regions [1]. A node stores multiple clusters (thousands of clusters per node) and responds to query for reading and writing on those clusters. A node can also take clients' requests and redirect the operations to the specific nodes holding the information. Thus, the cluster is split into two logical parts: Storage & API.

First, the API: Any Hive node can respond to any requests regarding whether the node has the data. The node does a cluster resolve on each request, caching most of the result for the next one, and then knows which node it needs to talk to fulfill said request. It is opposed to the traditional client redirection from RAFT architectures, as the node does the redirection for the client. This approach enables multiple optimizations, but the main ones are that we can cache the path and the nodes of the requests in order not to have to resolve it again, and we can put a simple load balancer, like HAProxy, in front of Hive, without worrying about redirections.

Thus, Hive has multiple APIs: Frontal ones, usually an HTTP API, that clients talk to, and an internal one, using protobuf for internal node communications. This design allows us to bind the client-facing server to a private IP address and use IPv6 internet addresses for node-to-node communication. In order to ensure the safety of communications between nodes, every packet is end-to-end encrypted using a public-private key pair issued to each node, with a central authority issued signature.

Second, the Storage: Each Hive cluster is composed of one or more storage backends, which can have different database engines and maintain an in-memory state machine or not. A cluster is expressed this way: `allocator/reference@world`; 'allocator' and 'reference' are both storage backends names, and everything after the '@' is a client-defined string. A storage backend implements a subset of a RAFT cluster with a log application and its database engine. A storage backend does not know what other backends it is paired with, despite the fact that a cluster shares the same RAFT log for all of its backends. Although this is not currently the case, each backend can use different storage engines for their storage: A backend 'A' can use SQLite, and a backend 'B' can use LMDB without any issues.

A common RAFT log is shared by the cluster, storing all the cluster operations and some "meta" operations, like adding a node into a cluster or changing the default consistency of a cluster.

Each node maintains a global cache in order to resolve a cluster when needed quickly. This cache is stored in RAM, with an AVL tree, and stores all the addresses of the nodes composing a cluster and which node was the last known leader. When a leader changes, any node can redirect the caller to it, and the caller then updates its cache. Upon cluster deletion or re-creation, the node will return a "Does not exist" code, resulting in a cache flush for this particular entry on the caller's side.

## 2.2 Quorums

A cluster is a collection of nodes replicating the same database through a quorum protocol, RAFT [1]. It is usually a 3 to 5 nodes cluster but does not have a size limit. There are two roles in a cluster, a leader and followers. There is only one leader in a cluster at any given time: The leader takes the write operations and replicates them to the followers to ensure consistency and safety. On strongly-consistent reads, it is also the only node in the cluster that serves reads. The followers replicate the log, notify the leader when they do so, and are in charge of triggering an election if the leader is absent for some time. The election process is the same as the one described in the RAFT paper, with some tuning regarding who can be elected leader depending on the commit index of the last applied log. On eventually consistent reads, the followers can also reply to these requests.

## 2.3 Consistency

Hive provides multiple levels of consistency for reading and writing requests. For read requests, two are proposed: *strong* and *eventual*:

- `Strong read consistency` means all the reads are going through the leader, thus ensuring read-after-write on any write consistency above eventual. There is a performance penalty, as the requests may be redirected on a new leader in case of a re-election, leading to latency or not being served at all if there is no leader on a cluster.
- `Eventual read consistency` means any node in a cluster can handle a read-request. That leads to theoretical consistency issues, as a follower could return an out-of-date entry if it has not replicated it yet. It does not come with a performance penalty, as Hive can then load-balance the reads on all cluster members, and the data is still served even on a leader-loss.

For write consistencies, Hive provides four different levels:

- `Absolute write consistency` means all the nodes of a cluster must have replicated an entry before acknowledging it. This provides absolute certainty about data safety but will be especially slow since all the cluster members must respond to this operation.

- `Strong write consistency` means a majority of nodes ($(N/2) + 1$) must have replicated an entry before acknowledging it. This is is usually the best ratio between performance and safety for most operations.
- `Eventual write consistency` The leader must have written the entry locally before the acknowledgment. This leads to consistency issues but with better latency performance.
- `Hopeful write consistency` The log describing the entry must be present in memory before the acknowledgment. As the name suggests, this does not assure replication and consistency but an excellent latency even under high load.

Though Hive enables consistency on an operation basis on the API that have implemented it, it usually defaults to the consistency defined at cluster creation.

## 2.4 Election & Allocation rules

When a cluster is created, a rule must be emitted to the allocator node in order to fulfill some requirements [2].

- `Size` is the cluster defined size; the number of members in it
- `Location` is the cluster geographic location (usually expressed from a cloud-provider point-of-view, so fr-par or nl-ams for example, in the context of Scaleway). When a cluster is to be globally available, this value can be "world".
- `Failure domain` is the parameter used to describe the desired failure domain of a cluster. In the figure [2] it's "AZ" which means 3 availability zones (e.g., data centers) can go down before any impact on the cluster. On a globally available cluster, this value is usually "region". The possible failure domains are: chassis, rack, az, region.
- `Consistency` is the default cluster write consistency. When a write-operation does not provide a desired consistency, Hive will default on this value for this cluster.
- `Read Consistency` is the default cluster read consistency. When a read-operation does not provide a desired consistency, Hive will default on this value for this cluster.
- `Repartition` is an argument, that if provided will force the allocator to respect a fixed repartition for cluster creation. A value can be `repartition=fr-par:2,nl-ams:2,pl-waw:1`, in order to force an exact node allocation on this particular election.
- `Exclude` is an argument, that if provided, will exclude a chassis from an election.

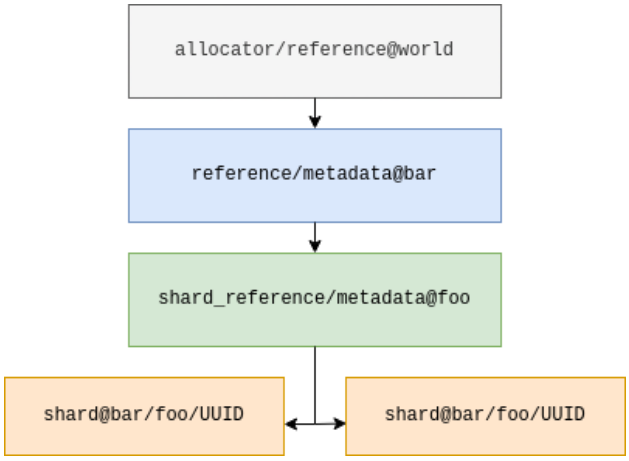


Figure 3: Multi level clusters in Hive

## 2.5 Multi-Level clusters

In order to not use sharding when it is not needed, Hive does a logical splitting of some clusters [3]. The idea is to separate data as much as possible to avoid cascading failures in production. A straightforward example is an S3 bucket: "The bucket 'foo' is stored in the region XX, and owned by the client 'bar.'" In this case, the level 1 cluster will be a database referencing all the client buckets "bar" (usually a UUID in real-life), and the level 2 clusters will be the entry point for the buckets storing references to object-shards and bucket metadata. With this logic, we can "resolve" a bucket with the following steps: Asks the level 0 cluster (the global cluster) for the IP of the account cluster, asks the account cluster for the IPs of the bucket cluster, then send the request. Of course, the result of those operations is mainly cached, so the cost of doing those calls will be minimal on a large number of requests.

A Hive deployment has two level 0 clusters: The Global cluster and the Regional cluster. Every cluster is referenced, directly or indirectly, by those clusters.

## 2.6 Global & Regional Cluster

In a Hive deployment, there are multiple clusters created by default. The first is the Global Cluster (or World cluster), a user-defined cluster: Members are explicitly written in configuration. This cluster is responsible for storing level 1 references to other clusters and handles election & allocations for cluster creations. It is usually a multi-region cluster to have the best availability possible but could be on the same rack if need be.

The regional cluster is a cluster that stores level 1 references to configuration clusters of a region. It is user-defined: Members of a regional cluster are explicitly written in configuration. Upon the first start, a Hive node will ask for a configuration cluster creation, with any member following those rules: It must be in the same region, and the allocator cannot elect a node on the same chassis. Once this operation is done, this configuration cluster will be used as a simple key-value store, storing the list of clusters that a node has joined. With this logic, a node can rejoin all of its precedent clusters even in the event of a total disk loss.

## 2.7 Backends

Backends are specific storage implementation behind an RPC service. They implement their storage implementation and gets forwarded logs based on their type. Multiple backends can live in one cluster, they all share the same log, but every backend has a dedicated database for data. Each backend is unaware of which backend it is paired with, and Hive is not aware of the storage implementation. It is currently usually an LMDB database, but it could be anything. A backend can also choose to maintain a state machine based on the logs. Hive currently implements the following backends:

- `allocator`: This backend is responsible for the gossip protocol and nodes in the cluster. It can be asked for nodes to be elected, and this election will be done depending on node score, and the election rules provided. This backends keeps a near real-time map of all the nodes in the cluster, their IPs, regions and scores.
- `metadata`: This backend is for simple key-value operations, like PUT / GET / DEL on simple value couples, e.g., string-string. It is mainly used to store unstructured metadata.
- `redis`: This backend exposes a redis-like API over protobuf. It is mainly used as a key-value store, especially for set-if-not-exists strongly consistent operations.
- `reference`: This backend is responsible for storing references to other clusters. It is simply a key-value store with a cluster name as a key, and a couple IP:Port, along with other information, as values.
- `multipart`: This is a specific S3 storage backend, that is used to handle multipart uploads operations.
- `shard_reference`: This is a specific storage backend for S3, that handles reference to shards. It is a unified interface into a bucket, and handles when to split or merge shards, and which to ask for a specific key.
- `shard`: This is a specific storage backend for S3, that handles objects. It is, by design, only responsible for a specific range of keys, as keys are sharded. It is a key-value store, with the object name as key, a bson object as value, which is storing versions, locations, ACLs, tags.

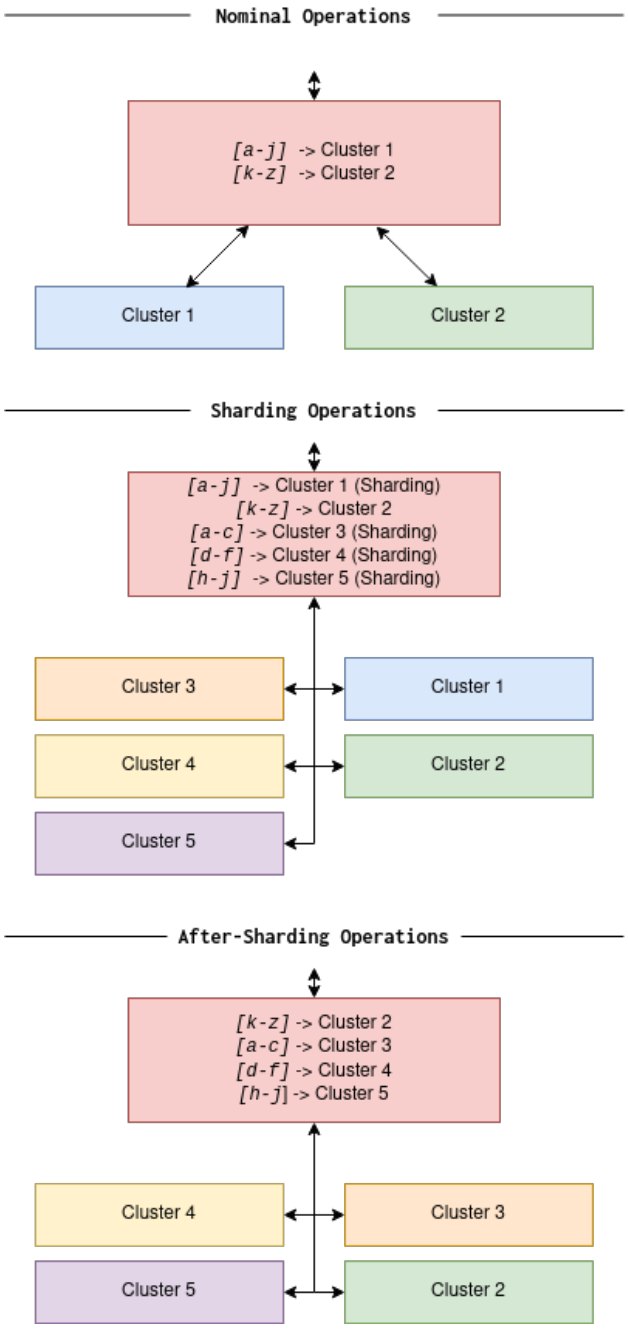


Figure 4: On-the-fly sharding operations

## 2.8 Scaling & Load repartition

In a Hive cluster, each node has a dedicated score; This score represents different metrics, like machine load, disk usage, or cluster occupancy. When possible, this score is used on an election first to take the node with the lowest score to ensure equal cluster loads on all nodes.

## 2.9 Sharding

In order to split the load evenly on many machines and avoid large on-disk databases, the key sets are split when they are large enough. Currently, and this is configurable, a shard is split in 3 when more than 40 000 keys are set. Three new clusters are allocated, the data is migrated from the old database to the new ones. All write requests are doubled [4] to ensure data safety even on sharding failure. Since everything is happening on the fly, a client should not notice when a database is being sharded, except for a slight increase in tail latency.

# 3 HIVE FOR S3

S3 is an Amazon API, which is, in essence, a key-value store. The key is an object name; the value is some binary content. In addition, some metadata can be set by the user on an object: tags, access control lists, or simple flags. Alongside this, the actual backend has to store the data's physical position if one does not store the object content alongside its metadata. Hive was created for this purpose: It exposes a key-value store HTTP API to an S3 API gateway and handles most S3 operations under the hood. Hive also enables the use case of huge buckets, with billions of objects, without a performance penalty. Hive is a database, but it is not designed to store the object content, only the metadata. In order to have a full stack, one can use a storage-specific solution like Ceph Rados or HDFS. Likewise, Hive is not an S3 API. Its HTTP API is related to S3 API calls but cannot be exposed directly to the client. A third party must handle some features like signature, ACLs and data storage.

## 3.1 Object storage API

Hive exposes an HTTP API for S3 in the form of a one-route body-action JSON API. The value may depend on the type of call, but the global rule is that every client-facing S3 call has an equivalent Hive call. The idea is for the gateway to do as little work as possible, leaving Hive the specific behavior of S3 in some cases: Versioning, Multipart Upload, and other operations. Even though HTTP comes with a bit of overhead, it was chosen to ease the client implementation; Indeed, every language has some library to implement an HTTP client with JSON payloads.

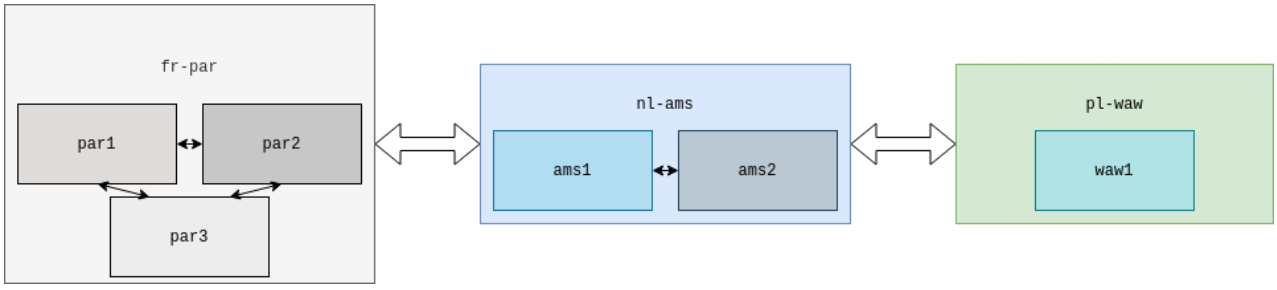


Figure 5: Multi-region hive deployment

## 3.2 Key unicity across multiple regions

A key S3 feature is the fact that a bucket name is unique across all S3 regions. In order to ensure strongly consistent writing and unicity, Hive is used. The safety calls and two-step-commits are automatically used on the 'CreateBucket' API call, leaving almost no work to do for the gateway.

In order to ensure unicity, a Redis-like backend is used, with a set nx=1, which means a key is only set if it does not already exist. It exposes a Redis-like API over protobuf but cannot be accessed directly by the caller. The cluster is automatically created on the first bucket, with a worldwide failure domain configured [5].

## 3.3 Prefix & Delimiter listing

S3 database usage is generally perfect for a key-value store: Most operations set a key or get one. However, some operations might be costly on naive implementations: Listing objects by prefix and delimiter or listing versions come to mind. In order to guarantee good performance on those calls, Hive does have specific routines handling this. One specific optimization was doing as few reads as possible on listing calls, even when a request might involve thousands of different shards.

| Run | p50 | p75 | p95 | p99 | p999 | Rate (op/s) |
|---|---|---|---|---|---|---|
| 1 | 2.32ms | 2.75ms | 5.02ms | 11.93ms | 24.10ms | 25540 |
| 2 | 2.49ms | 3.04ms | 5.31ms | 11.94ms | 32.552s | 23658 |
| 3 | 2.52ms | 3.06ms | 5.02ms | 11.81ms | 23.63ms | 25633 |
| | 2.45ms | 2.95ms | 5.12ms | 11.89ms | 26.76ms | 24944 |

Figure 6: s3-object-put-100k, replicas=5, write_consistency=strong, read_consistency=strong

| Run | p50 | p75 | p95 | p99 | p999 | Rate (op/s) |
|---|---|---|---|---|---|---|
| 1 | 1.52ms | 1.79ms | 2.48ms | 3.83ms | 8.38ms | 49000 |
| 2 | 1.49ms | 1.92ms | 3.66ms | 6.79ms | 11.95ms | 44246 |
| 3 | 1.45ms | 1.76ms | 2.51ms | 3.95ms | 8.47ms | 50407 |
| | 1.49ms | 1.83ms | 2.88ms | 4.86ms | 9.60ms | 47884 |

Figure 7: s3-object-get-100k, replicas=5, write_consistency=strong, read_consistency=strong

# 4 EVALUATION

We first measure Hive's performance with respect to replication, transactions and availability. We then provide some data for our real-world use case with S3.

## 4.1 Local Benchmarks

Figure [6] and Figure [7] present some S3 benchmarks, done locally on a single computer, with a five node cluster. The CPU used for those benchmarks is an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, with a 500GB NVME SAMSUNG MZVLB512HAJQ-000L7. Clients were run on the same machine, thus impacting the tail latency on the results, but without latency between the clients and Hive. Operations were a standalone read (or write) of a sample object, containing around 2KB of data, which is the average of what we have in production.

In both testing suites, three runs were done in order to have representative latency and operations average. To see real-world latency values, see the 4.3 Section.

## 4.2 Availability

Multiple tests were done in order to ensure availability and data-safety:

- `Machine loss`: Minimal impact on tail latencies if the chassis was hosting a follower; noticeable impact if the node was a leader; Write-Requests are hanging while the cluster elects a new leader, which can take up to a second.
- `Disk loss`: In case of a disk loss, the throughput impact is about the same as a machine loss. Since disks never go out but rather go read-only for the system in the real world, the leader will change if need be after failing to write new entries. Upon disk change, the nodes on the chassis replicate the lost databases in a reasonable amount of time, mainly depending on the network connection and the disk bandwidth.
- `Datacenter loss`: For cluster with an AZ failure domain, same impact as a machine loss.
- `Network partition`: If a leader is unable to reach its followers for more than a few seconds, it automatically steps down from leadership to ensure there is no network partition. In that time, strongly consistent writes will fail as there are no quorums on those operations. Clients might experience a few internal errors in that time window if all the internal retries budget is spent.

## 4.3 On production

At the time of writing, Hive has been in production on the Scaleway S3 product for around a month. In that time, 5 billion entries have been written, with an average write P90 around 1ms and an average read of P90 around 150us. The aggregate availability (return_code != 5xx / total_requests * 100) is 99.9998%, which is less than what we hoped for, but due to bugs and crashes expected with a first deployment. Overall, we are pretty happy with those numbers, as the node crashes almost had no impacts on client-facing calls, besides the occasional spikes in tail latency.

# 5 FUTURE WORK

In the future, we aim to work on extensions of Hive:

- An etcd compliant server, using hive sharding and consistency features to spread the load on mutualized etcd clusters.
- A redis compliant server, that can be used with little to no modification for existing clients, while getting

the advantages of sharding and the different consistencies implemented in Hive.

- Fully handle load-balancing reads based on locality. Right now, a read might cross a continent, even though the data is a few server away.
- Leadership election with set times: With write patterns, it might be helpful to transfer leadership to a locality A when locality A has a strong write pattern on specific clusters.
- A generic SQL implementation: Something like a PostgreSQL compliant server, in order to have some form of a structured database on top of hive.
- Directly store small objects in hive, rather than storing bigger references to it.

## 6 RELATED WORK

Amazon's ShardStore [2] has provided consistent and replicated shard storage for S3. For a more key-value storage architecture, KeyDB [3] and Redis come to mind. For a global replicated store, Spanner [4] and DynamoDB [5] are the leading solutions. Lastly, CockroachDB [6] uses the same quorum protocols as Hive, with an extensive PostgreSQL compatibility layer.

## 7 CONCLUSION

To summarize, Hive aims to provide a globally distributed key-value store with configurable consistencies and failure domains for various usage. It is optimized and used as an S3 backend object-store. It provides excellent latencies combined with high throughput while ensuring data safety and replication.

The modular design of Hive allows for rapid development of many features, with a simple production deployment, as it internally handles load-balancing, redirections, and elections.

## 8 ACKNOWLEDGEMENTS

Many people have helped improve this paper: Hana Khelifa, Yohann Bacquey, Jean-Baptiste Bernard, Eloi Cousinet. Our management has supported both this project and published this paper: Gaspard Plantrou, Synda Ben-Jerad, Jean-Baptiste Bernard. Finally, a colleague who was on our team and helped immensely with this project: Vincent Auclair.

## References

[1] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. 2014. URL: https://raft.github.io/raft.pdf.

[2] James Bornholtand Rajeev Joshiand Vytautas Astrauskasand Brendan Cullyand Bernhard Kragland Seth Markleand Kyle Sauriand Drew Schleitand Grant Slattonand Serdar Tasiranand Jacob Van Geffenand Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. 2021. URL: https://assets.amazon.science/07/6c/81bfc2c243249a8b8b65cc2135e4/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3.pdf.

[3] EQ Alpha Technology. The faster redis alternative. 2019. URL: https://keydb.dev/.

[4] James C. Corbettand Jeffrey Deanand Michael Epsteinand Andrew Fikesand Christopher Frostand JJ Furmanand Sanjay Ghemawatand Andrey Gubarevand Christopher Heiserand Peter Hochschildand Wilson Hsiehand Sebastian Kanthakand Eugene Koganand Hongyi Liand Alexander Lloydand Sergey Melnikand David Mwauraand David Nagleand Sean Quinlanand Rajesh Raoand Lindsay Roligand Yasushi Saitoand Michal Szymaniakand Christopher Taylorand Ruth Wangand Dale Woodford. Spanner: Google's globally-distributed database. 2021. URL: https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi2012.pdf.

[5] Giuseppe DeCandiaand Deniz Hastorunand Madan Jampaniand Gunavardhan Kakulapatiand Avinash Lakshmanand Alex Pilchinand Swaminathan Sivasubramanianand Peter Vosshall and Werner Vogels. Dynamo: Amazon's highly available key-value store. 2007. URL: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

[6] Rebecca Taftand Irfan Sharifand Andrei Mateiand Nathan VanBenschotenand Jordan Lewisand Tobias Griegerand Kai Niemiand Andy Woodsand Anne Birzinand Raphael Possand Paul Bardeaand Amruta Ranadeand Ben Darnelland Bram Gruneirand Justin Jaffrayand Lucy Zhangand and Peter Mattis. Cockroachdb: The resilient geo-distributed sql database. 2020. URL: https://dl.acm.org/doi/pdf/10.1145/3318464.3386134.