Final Project Report 50.041 Distributed Systems and Computing SUTD 2022 CSD

Group 5: SandDB
A Decentralized Structured Storage System

Filbert Cia (1004415)
James Raphael Tiovalen (1004555)
Ong Zhi Yi (1004664)
Yu Nicole Frances Cabansay (1004574)

Course Instructor: Sudipta Chattopadhyay

Table of Contents

1. Introduction

1.1 Motivation of Problem Statement

Data is the lifeblood of every business – which is why we need a database at the centre of it all. Database management in one machine is practically a trivial problem in computer science in today's age. In comparison, distributed databases are much harder to design and implement, simply because of their distributed nature, where consistency across database nodes is not guaranteed. However, we need distributed databases to meet the growing data needs of today's businesses. By utilizing distributed databases as the core data infrastructure of an organization, we can gain improved performance, enable massive scalability, as well as deliver round-the-clock reliability for all clients around the world.

Nowadays, large-scale systems such as social media and e-commerce platforms that span globally with thousands of nodes require high availability for users to be able to continuously access and use them without failure. At this scale, many components are almost guaranteed to fail continuously, even when they have a very low probability of failure in single systems, and thus, in the face of these failures, we need to build a reliable and scalable system that can be used by clients that would implement such large-scale systems. Therefore, we aim to implement a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing a highly available service with no single point of failure. The Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency.

1.2 Introduction to Cassandra

Apache Cassandra (Apache, n.d.) is a highly available and partition-tolerant distributed database system initially designed and developed by engineers at Facebook before being released as open-source software under the Apache licence in 2008. Apache Cassandra was designed to provide approximately linear horizontal scalability with respect to the number of nodes. It provides the best-in-class combination of Google Bigtable's data and storage engine model and Amazon's Dynamo's distributed storage and replication techniques. It focuses on performance, scalability, and reliability, but it can only provide eventual consistency. Its focus is on the AP side of the CAP theorem.

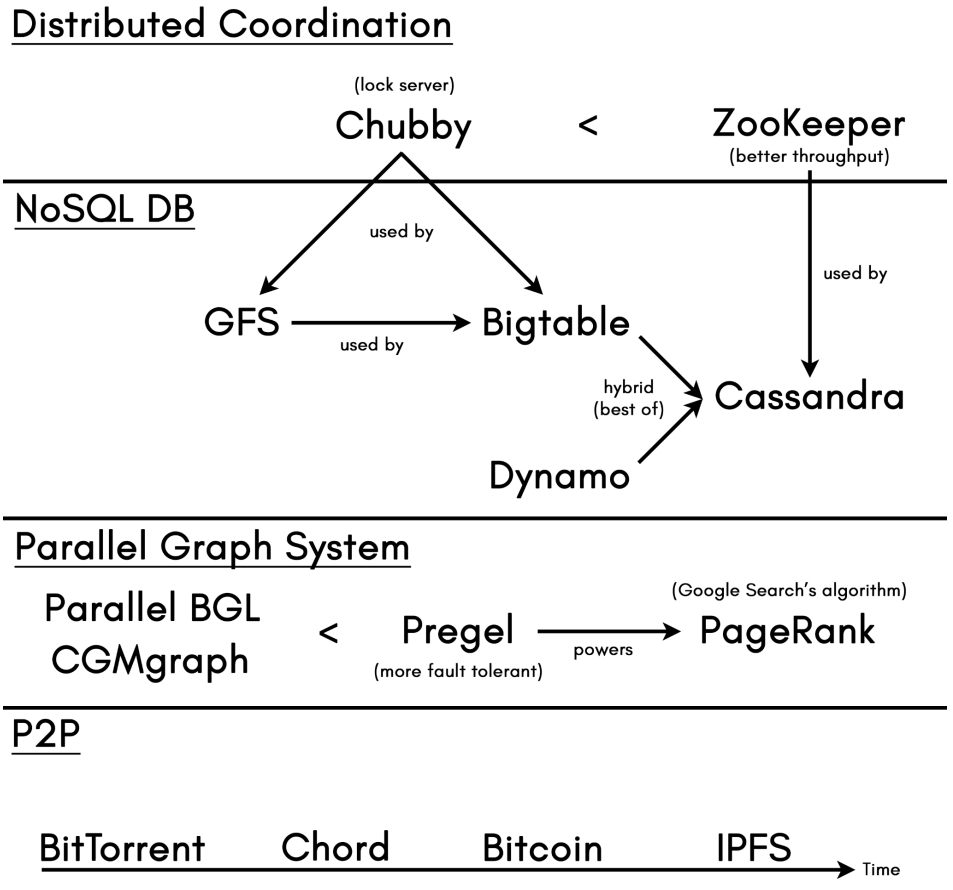


Figure 1: Several distributed protocols/systems and how they are connected/linked to each other.

Cassandra is still currently utilized by many big enterprise-scale companies, some of which are: Activision, Apple, Bloomberg, CERN, Cisco, Comcast, Coursera, Discord, eBay, Hulu, IBM, Instagram, McDonald's, Microsoft, Netflix, New York Times, Reddit, SoundCloud, Spotify, Target, Uber, Walmart, Yelp, and many more! It is ranked #11 out of 383 on DB-Engines (*DB-Engines Ranking - Popularity Ranking of Database Management Systems*, n.d.) in terms of popularity in May 2022 and it is #1 for wide-column store databases (*DB-Engines Ranking - Popularity Ranking of Wide Column Stores*, n.d.). It is a battle-tested, tried-and-true, mature protocol with extensive and rich community support. Similar ideas/concepts in the protocol are used as the foundational basis of other more recent and more performant NoSQL databases, such as ScyllaDB.

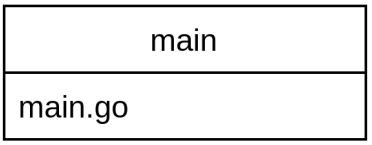
For our SandDB implementation, we take inspiration from Apache Cassandra and refer to the annotated version of the original paper (Ellis et al., n.d.), following closely its specifications where feasible and minimally diverting if necessary after taking the constraints of doing this project into account.

2. Overall SandDB System Architecture

2.1 Components

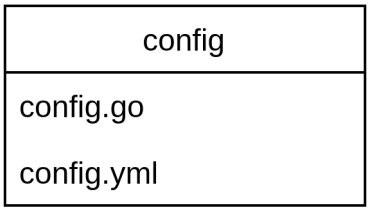
Our system consists of the following components, each of them segmented into a package.

2.1.1 main



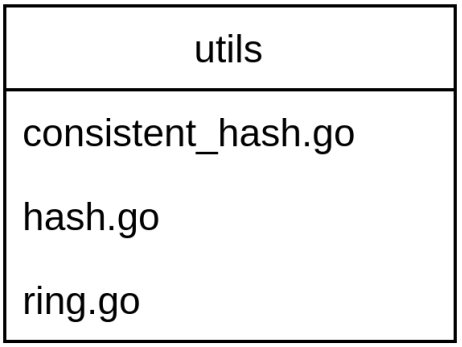
The root directory (main package) consists of the main.go file, which contains the logic to bootstrap and gracefully shut down a node. It also sets up the different URL routes to support the system. Finally, it also sets up the different handlers used for the larger components - db, read_write, and anti_entropy.

2.1.2 config



The config package consists of the *config.yml* file which describes the properties of the ring, along with the properties of each node in the ring. It also contains a *config.go* file which describes the Configuration struct to store for which the *config.yml* will be unmarshalled into.

2.1.3 utils

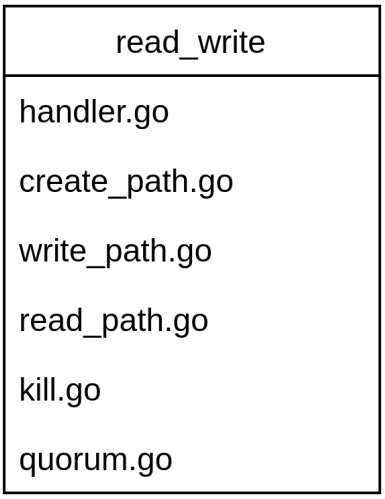


The utils package consists of mainly 2 sections - node and ring structs, as well as consistent hashing-related functions.

2.1.4 messages

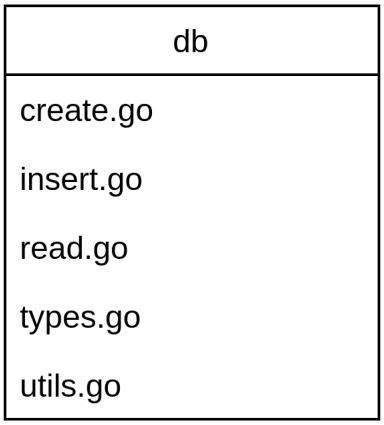
The messages package consists of the PeerMessage and various Request structs. PeerMessage is the struct sent among nodes for ACK messages and kill/revive messages, whereas Request structs are the ones which are marshalled into whenever a node receives a request, be it from the client or the coordinator node.

2.1.5 read_write



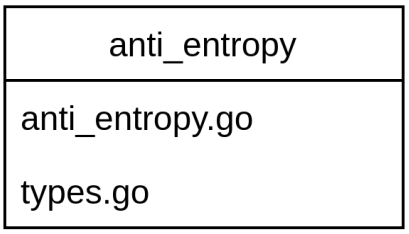
This package is the one responsible for handling all client requests and maintaining the quorum. In all paths (create, read, write paths), the read_write handler of the coordinator node, upon receiving the client request will unmarshal the HTTP request body into the respective Request struct. After computing which nodes it should forward the request to, it will then send an HTTP POST request to those replica nodes. Depending on the responses of the HTTP POST, the coordinator's read_write handler will then return a success or error HTTP response to the client. More specific error handlings are done on each underlying function.

2.1.6 db



The db package maintains the structs and functions to interface with the persistent storage of this system. It contains a file to handle each high-level function (create, insert, read). Since there is some reusable code, these functions are extracted to *utils.go* in the same package.

2.1.7 anti_entropy



The anti_entropy package provides the functionalities for the Anti-Entropy repair module. It includes all of the endpoints handled by the AntiEntropyHandler, 4 of which are internally used to facilitate the repair process and 2 of which are externally exposed to the client. The included *types.go* file provides the struct definitions for different request and response messages used by the repair process, as well as different repair status constants for feedback to the client.

2.2 Features

2.2.1 Partitioning via Consistent Hashing

Our SandDB implementation utilizes consistent hashing to manage the logical structure of the nodes. The nodes themselves are logically organized in a ring-based topology. The nodes are ordered based on their IDs. Each node is responsible (i.e., is the owner) for a range of tokens/hashes that the data can be hashed into, depending on the hash values of their IDs. We utilize MD5 as the hashing function.

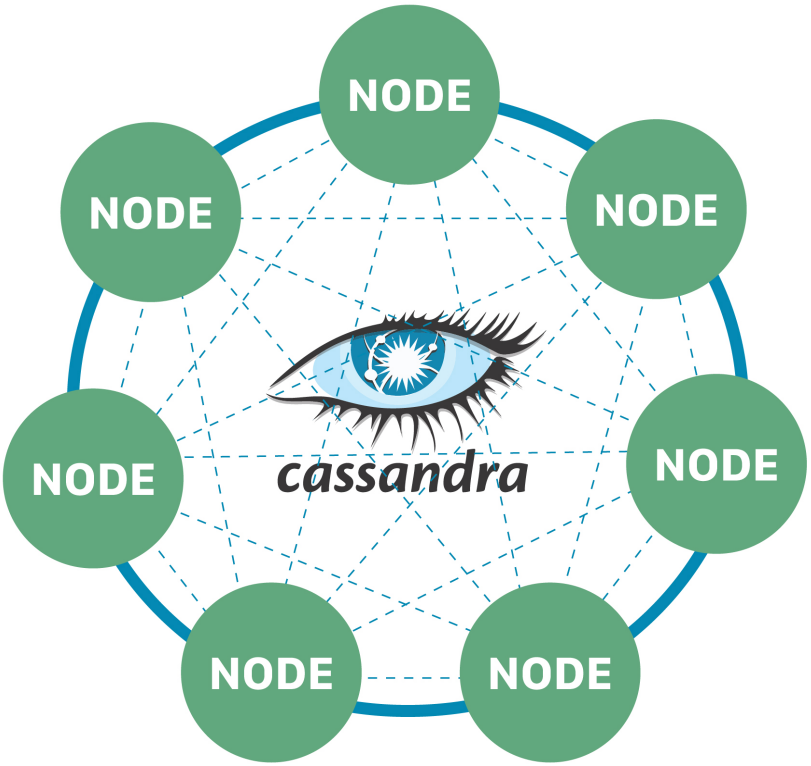


Figure 2: Overall logical structure of Cassandra nodes.

Assuming a replication factor of RF and N as the number of nodes, our implementation enforces a policy that RF ⩽ N. This is because if RF > N, the effective value of RF is N. Our implementation follows Apache Cassandra closely, which is to prevent/reject any nodes from starting up if this policy is violated.

For data that belongs to a specific node, our replication policy would be to also replicate said data to RF-1 subsequent nodes in the order specified by the ring. Hence, RF would indicate the number of replicas that each data is stored in for the sake of fault tolerance.

2.2.2 High Write Availability

Clients are able to GET/INSERT even amidst server failures. They also possess the ability to write to any server/node at any point in time. Following Cassandra closely, we can compromise some consistency to reduce lag. This is implemented

in SandDB via leaderless replication and the last-write-wins policy.

2.2.3 Eventual Consistency

We want the client to reflect the right values of data in the event that writes to replicas are not completed on time. While Cassandra's consistency level is tunable, depending on the needs of the application developer, for our project we would just select one particular consistency level "convenient" for us due to time constraints. Hence, in our implementation, we chose the QUORUM consistency level to implement strict quorum, which is the most relevant consistency level to this Distributed Systems course, along with hinted handoffs and read repairs to help preserve consistency.

2.2.4 Limited Local Persistence

For our project, we implemented a limited version of the local storage persistence implemented by Cassandra. Managing the CommitLog in the hard disk and the Memtable cache in RAM would be quite tedious and they are not the focus of the course anyway. The binary data storage structure formatting in SSTables, while optimised, is also quite troublesome to implement and it is also not the focus of the course. Hence, we would just directly write to JSON files on the hard disk. That said, without full local persistence, we acknowledge that our database would be severely limited when facing system crashes that would wipe out the RAM, and thus, our version of Cassandra can only rely on its other features such as fault tolerance and eventual consistency to somehow enact error correction and hopefully ensure as much correctness as possible.

2.3 SandDB Architecture

For our SandDB implementation, we are focusing on the following highlighted sections of the stack:

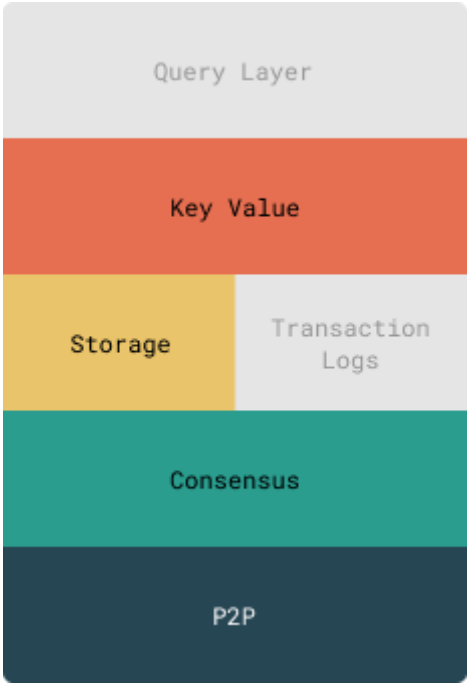


Figure 3: Implemented sections of the stack.

The following diagram shows how the packages are linked to one another. The arrows in the diagram show how one package is dependent on another.

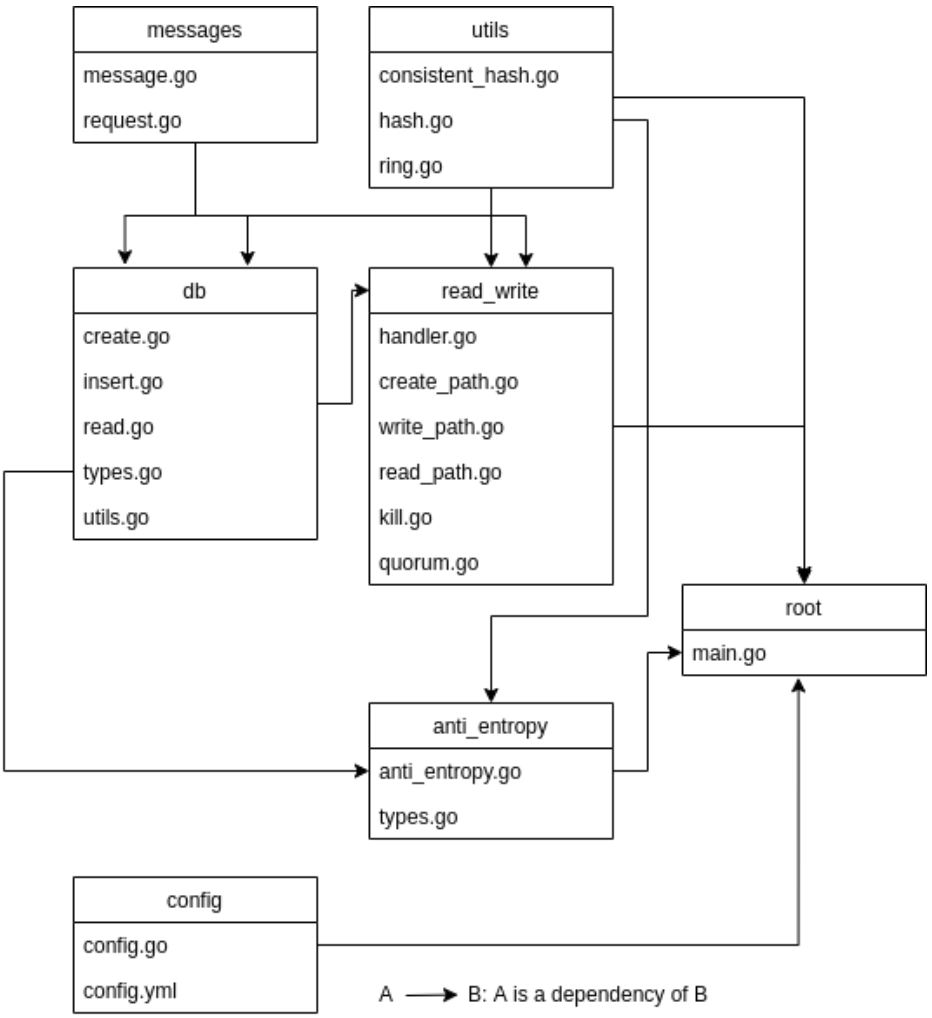


Figure 4: Package dependency graph.

2.4 Operation Flow Diagrams

2.4.1 Read Path

SandDB exposes a READ API with the following specifications:

HTTP Method: POST

URL: https://localhost:<port>/read/

Sample Request Body:

```
{
     "table_name": "hospitals",
     "partition_keys": ["1","GENERAL"],
     "clustering_keys": ["AA-1"]
```

}

- table_name: name of the table to be inserted/updated
- partition_keys: values of the partition keys
- clustering_keys: values of the clustering keys

The read path for such an operation is as follows:

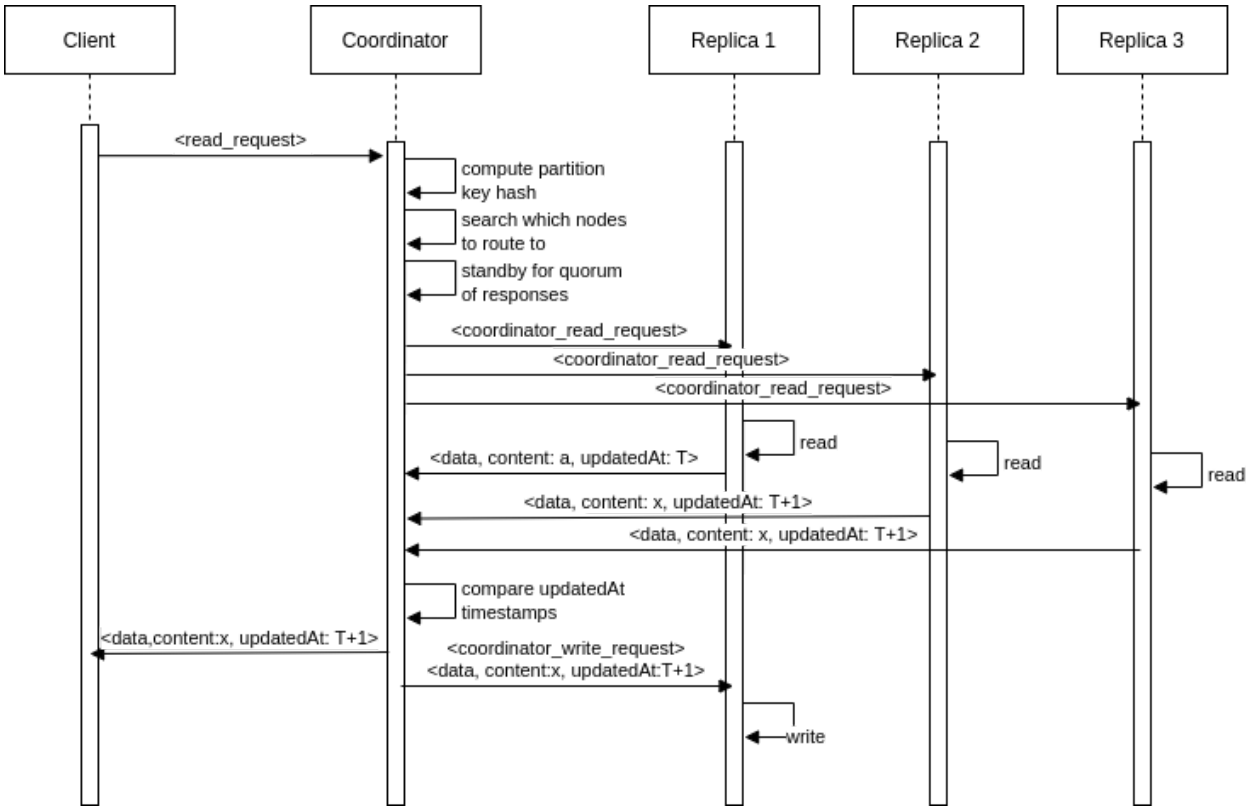


Figure 5: Read path for the READ operation.

Last Write Wins:

In SandDB, the client sends the read request to any node at random, and the node that receives the request becomes the coordinator for the request. Upon becoming coordinator, the node computes the petition key hash based on the petition keys provided and searches for nodes which contain the data. It then starts a quorum for the request and sends a read request to each node containing the data. Each node that is not the coordinator, upon receiving the read request, will send back to the coordinator the data that it has stored. Once the coordinator node detects that it has received enough replies

to pass the quorum, it will select the most updated write out of all the replies that it received based on the updatedAt timestamp of each data to send back to the client. The nodes synchronize their clocks by utilizing some kind of clock synchronization protocol such as Network Time Protocol (NTP). Apache Cassandra utilizes last-write-wins instead of vector clocks (like Dynamo's implementation) to resolve concurrent write conflicts for the sake of improved performance and simpler application design (Ellis, 2013).

Read Repair:

Following the reply to the client, the coordinator node will also detect nodes that contain outdated writes and send the most updated version of the data to those nodes. In the case that quorum is not passed, the coordinator node will not send any data to the client, but will still perform a read repair on nodes that have outdated writes.

2.4.2 Write Path

SandDB also exposes an INSERT API with the following specifications:

HTTP Method: POST

URL:http://localhost:<port>/insert/

Sample Request Body:

```
{
     "table_name": "hospitals",
     "partition_keys": ["1","GENERAL"],
     "clustering_keys": ["AA-1"],
     "cell_names": ["Bed","Oxygen Tank"],
     "cell_values": ["3","10"]
}
```

- table_name: name of the table to be inserted/updated

- partition_keys: values of the partition keys
- clustering_keys: values of the clustering keys
- cell_names: column headers to be added to the row
- cell_values: values of the columns to be added to the row

The write path for such an operation is the following:

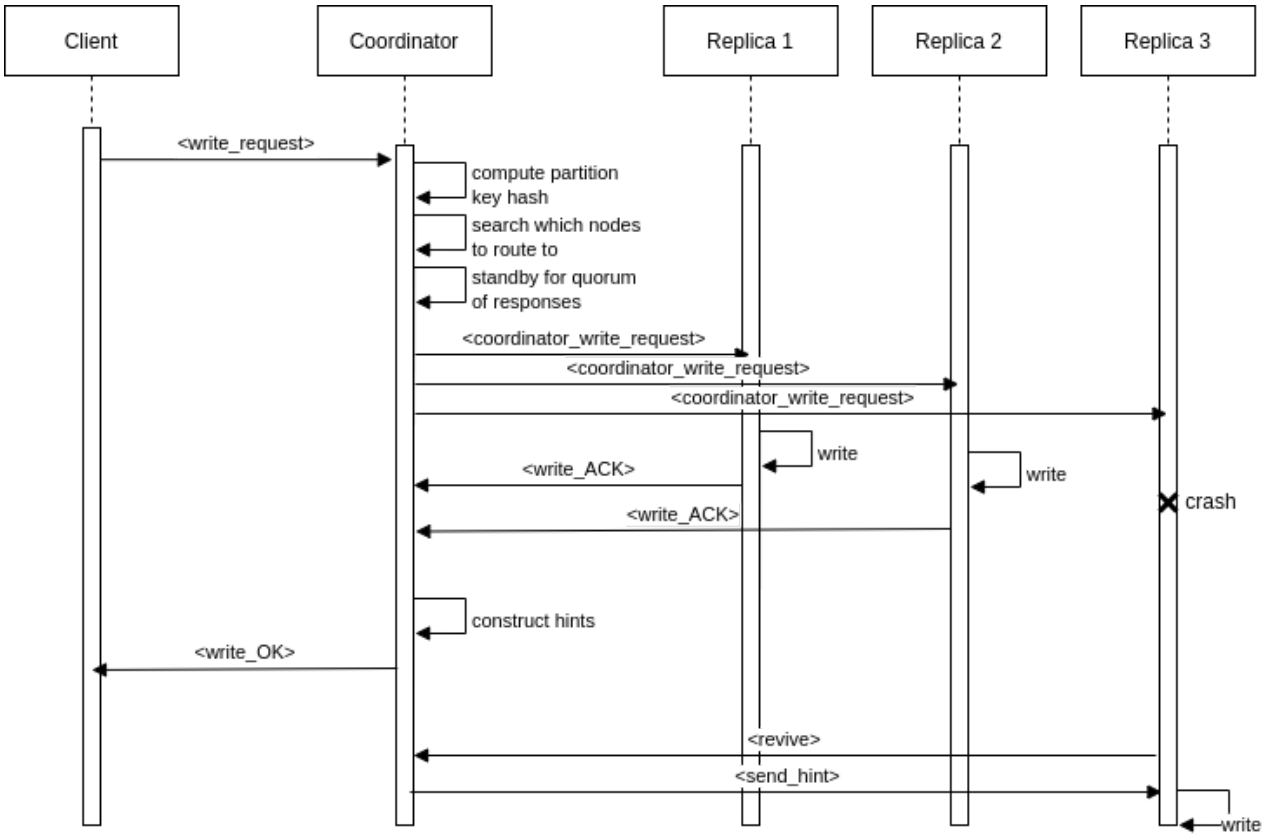


Figure 6: Write path for the INSERT operation.

In SandDB, the client application could send the request to any node. The receiving node will then be the coordinator for this particular request.

Once the coordinator receives the write request, it will then compute the hash value of the partition key(s) and search which replica nodes should handle the write request based on the consistent hashing function. It will then standby for responses from the replicas.

The replicas, upon receiving the coordinator's write request, will then persist the data into its local table. The replica

will compute the hash value of the clustering key(s) to determine which partition to update/insert. Within the partition, the replica will then update the relevant cells from the write request and persist them.

After successfully persisting the data, the replicas will send a write_ACK message to the coordinator. The coordinator will then keep track of which replicas did not send a write_ACK and will construct hints based on this information.

Once the replica revives, the coordinator will then send the hints to the replica and the replica will persist the data to ensure consistency.

2.4.3 Node Bookkeeping

To simulate a node kill in SandDB, a request is sent to all the other nodes. The receiving nodes will update their own copy of the ring and change the status of the node that sent the kill request. This removes the node from the ring. To simulate a reviving node, a similar concept is applied. When a node is revived, the node informs all the other nodes in the ring of the revived node’s existence. The other nodes in the ring update their own copy of the ring and the revived node’s status.

2.4.4 Anti-Entropy Repair

This is the overall sequence diagram for a full anti-entropy repair of a SandDB distributed database instance, assuming that a full repair is initiated by a node with ID 0, the replication factor is RF, and the number of nodes is N:

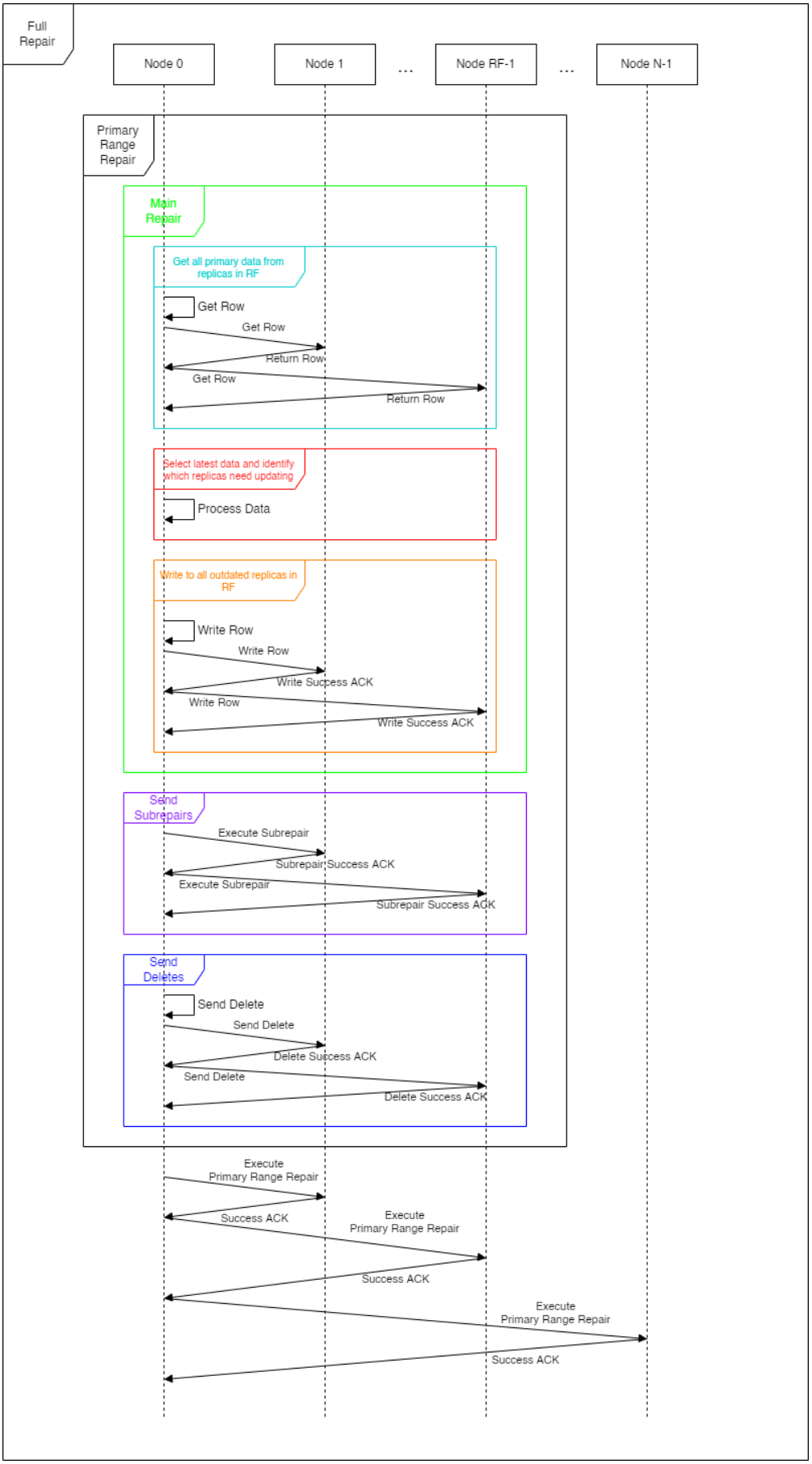


Figure 7: Overall sequence diagram for a full anti-entropy repair.

In our implementation, the client can only choose either a full repair or a primary range repair for the Anti-Entropy module. A full repair is composed of multiple primary/partitioner range repairs executed on a rolling basis, starting from the node that initiates the full repair (i.e., the one that received the /full_repair request from the client). A primary range repair will simply repair the 'primary' token ranges of the node that is executing the repair. A primary range is just a token range for which a node is the first replica (i.e., the main 'owner') in the ring. By chaining multiple primary range repairs in a sequential fashion to form a full repair, we avoid redundant/duplicate work.

A primary range repair consists of 3 main phases:

- Main Repair: In the main repair phase, the data contained in each row that belongs to the primary node (i.e., the node currently executing this main repair) will be requested from the primary node itself and its replicas. Then, on the primary node, it will compare these data and select the latest/most updated version based on the updatedAt timestamp (i.e., following the last-write-wins policy). While the precision of the timestamp is in the nanoseconds and thus it is very unlikely that multiple data will have the same timestamps, in the case whereby multiple data have the same timestamps, then we will select the greater data value in terms of the actual data bytes, as per Apache Cassandra's implementation (Spitzer, 2017). After that, the primary node will send a data write request to all of the replicas, including itself, to update the data value of that row with the latest data. For replicas that indicated that they do not contain this particular row, the row data will be added/inserted into their data storage file.
- Subrepairs: In the main phase, the primary node relies on the data that is visible to itself to perform the repair. However, since all replicas are technically equal in Cassandra, it is possible that data that is owned by a

node might not be present in its data file due to some kind of failure (assuming that QUORUM has passed without the primary node successfully performing the write to itself). Therefore, we need some kind of mechanism to synchronize data that might be present in the other replicas. To do this, the primary node sends a series of subrepairs, whereby it will request all of its assigned replicas according to the Replication Factor to perform this subrepair. A subrepair is essentially the same as the main repair, with the only difference being that the subrepair is performed on behalf of the primary node, instead of for data owned by the node executing the subrepair.

- Deletes: Finally, since there might be tombstones that exist in the primary node and its replicas, the primary node will send a delete request to all of its replicas and itself to remove all tombstones with a deletedAt value older than the threshold (GC_GRACE_SECONDS). Due to network asynchronicity, there might be edge cases whereby some replicas have passed the deletion threshold while some other replicas have not, leading to some inconsistency. However, since tombstones are not returned to the client in any form, this situation is still acceptable as **eventually**, all replicas will pass the deletion threshold and remove their tombstones in the future. Hence, this still preserves eventual consistency.

When performing a repair, several strong assumptions are being made:

- Client requests are deferred until a repair is complete (or that the client's requests are not frequent enough). A background thread could potentially handle this repair, but additional care needs to be taken when resolving conflicts between client requests and repair requests (such as by comparing timestamps).
- No network partitions occur DURING the repair process. This also assumes that all messages eventually arrive at their designated destinations.

- No non-Byzantine or Byzantine failures, such as node crashes or wrong computations, occur DURING the repair process.

There are 2 client-facing endpoints available at each node that the client can utilize to initiate a repair:

- /repair: This will inform the specified node to start executing a primary range repair.
- /full_repair: This will inform the specified node to start executing a full repair.

There are 4 internal endpoints utilized by the Anti-Entropy module to facilitate inter-node communication to properly perform repairs:

- /internal/repair/get_data: Used to get row data from replicas.
- /internal/repair/write_data: Used to write/update row data to replicas.
- /internal/repair/trigger_delete: Used to trigger deletion of old tombstones.
- /internal/repair/missing_subrepair: Used to initiate a subrepair of data owned by the requestor node in the destination node.

Several constants relevant for the Anti-Entropy module include:

- Internal Request Timeout: This is the timeout value used internally by the nodes when performing inter-node communication to facilitate the repair process. The default value is 30 seconds.
- Overall Repair Timeout: This is the timeout value used by the client when issuing a repair request to any node. The default value is 8 hours.
- Garbage Collection Grace Seconds (GC_GRACE_SECONDS): This is the threshold used to determine whether a tombstone should be deleted or not. The default value is 10 days.

3. Design and Implementation

3.1 Correctness

To ensure correctness, our SandDB system utilizes 3 main layers:

3.1.1 Read Repair

The first layer of defence against intermittent faults is read repair. Read repair is conducted by the coordinator node upon receiving an outdated value from one of the replicas. This feature supports correctness as the current coordinator will attempt to ensure that all owner replicas maintain the most updated data during reads.

3.1.2 Hinted Handoff

The second layer of defence against intermittent faults is hinted handoffs. Hinted handoff is carried out by the coordinator node when it detects that some replica nodes do not receive the write request. This is detected through the absence of write_ACK by the coordinator. Once the replica revives, the coordinator node will send the Hint that contains the write operation that the replica did not manage to receive and the replica will maintain the most updated value of the data. This feature helps to correct the replica’s data when it faces an intermittent fault.

3.1.3 Anti-Entropy Repair

When nodes permanently crash, it might take a while for them to be restarted manually/automatically or for human operators to notice. As such, they might miss some hinted handoffs. If there are infrequent reads, the data stored in such nodes might also be outdated since they might not be corrected via read repairs. Hence, the last layer of defence is to perform manual anti-entropy repairs. For Apache Cassandra, it is

recommended to perform anti-entropy repairs routinely and regularly to ensure node health and data integrity. Full repairs should be executed weekly to monthly and since it is quite costly, they should be scheduled during low-usage hours (*When to Run Anti-Entropy Repair | Apache Cassandra 3.0*, 2022).

3.2 Fault Tolerance

3.2.1 Consistent Hashing

Consistent hashing is implemented to mitigate faults during write operations. Every data to be written to a node will be replicated to other nodes based on the replication factor. When a write request is being made, the data content will be redirected to the node based on the hashed value of its partition key(s), making it the coordinator.

3.2.2 Last-Write-Wins and Leaderless Replication

The last-write-wins policy is adopted during read operations, where the coordinator selects the latest write based on the updatedAt timestamp. Last-Write-Wins supports fault tolerance as it takes into account the possibility that there could have been intermittent failure between nodes and those nodes may not have received the latest write. It thus ensures that the client will receive the latest write even when there are intermittent failures.

Leaderless replication means that the client is able to make a read request with any node, meaning that any node can become the coordinator (or leader). This supports fault tolerance as there is no single point of failure in the case of a single appointed leader, and thus read requests can still be carried out in the event of node failure.

3.3 Scalability

3.3.1 Implementation Design

SandDB is a scalable system due to the following implementation details:

3.3.1.1 Message Passing Model

As a distributed system, message passing is utilized to pass/transmit information between nodes. This provides significant decoupling between operations performed in each node. A QUORUM consistency level was chosen to ensure an acceptable tradeoff between consistency and performance.

3.3.1.2 Customizable Parameters

Certain parameters such as the replication factor and the timeout values can be easily controlled by the user.

3.3.1.3 Ease of Adding/Removing Nodes

Nodes can be added/removed by simply appending or deleting the corresponding {ID, IP Address, Port} entries in the *config.yml* file. This serves as an efficient and easy-to-use method to modify the resources used.

3.3.2 Performance Testing

A scalability test was conducted to test the performance of SandDB with different numbers of nodes.

Testing parameters:

- Quorum Timeout: 20 ms
- Request bodies for reads and writes are the sample requests from 2.4.1 and 2.4.2.

The performance test results are:

1. Assuming a constant replication factor of 5:

Number of Nodes	Time to Read (ms)	Time to Write (ms)
5	31	32
10	30	31
15	30	34
20	32	35

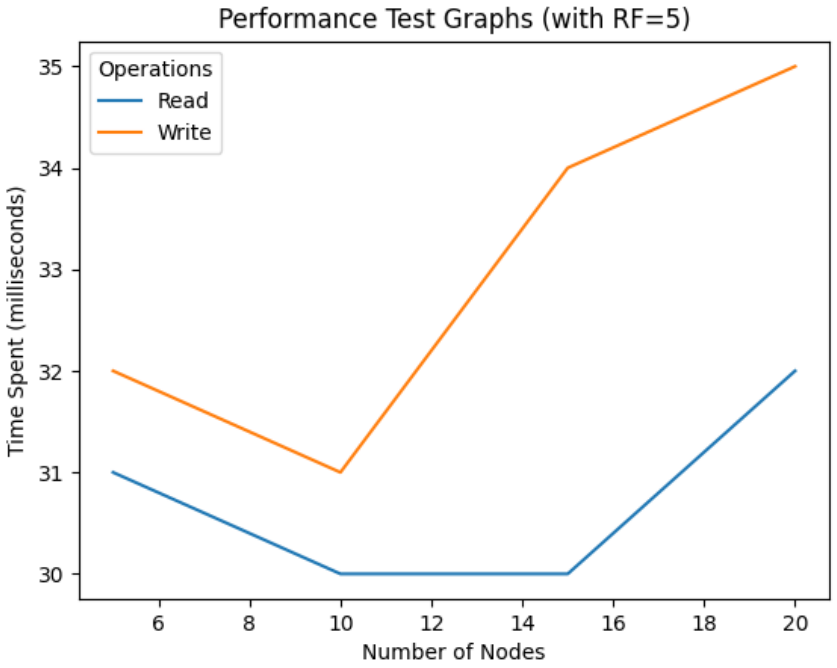


Figure 8: Graph of time spent against the number of nodes.

2. Assuming a constant number of nodes of 20:

Replication Factor	Time to Read (ms)	Time to Write (ms)
5	28	27
10	27	30
15	32	36
20	32	38

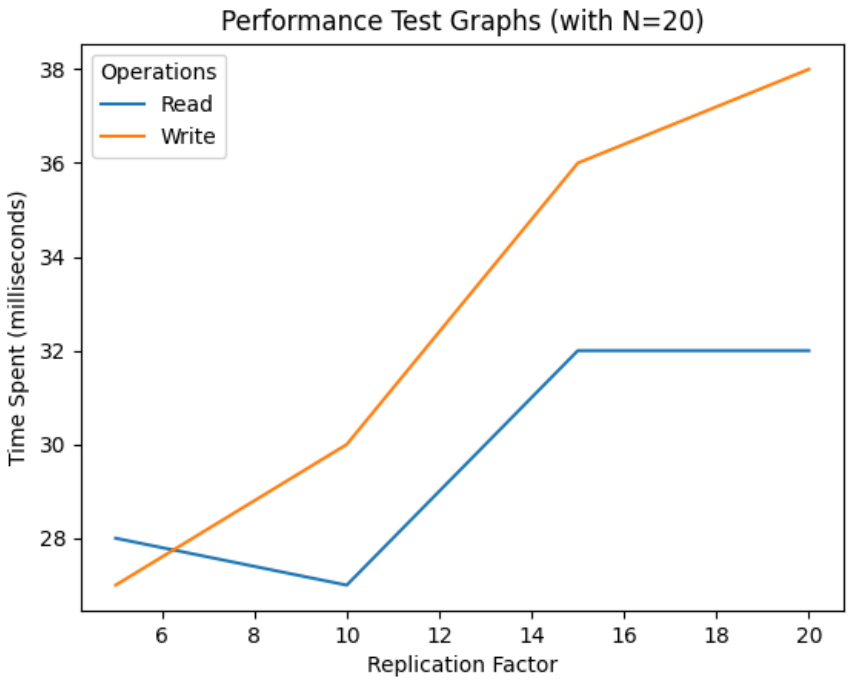


Figure 9: Graph of time spent against the replication factor.

As can be seen from the results, an increase in the number of nodes maintains a relatively constant amount of time for reads/writes. However, an increase in the replication factor results in a general increase in the amount of time taken to read/write the data. This is because the higher the replication factor value, the more number of replica nodes the coordinator will have to send to and the minimum number of responses to wait for.

Higher values of the replication factor also result in a longer time taken to read because the coordinator will have to parse through the responses to compare the values of the data. Reads also generally take a shorter time compared to writes, as write operations are also composed of reading values and replacing them with updated/new values in the RAM.

From this performance test, we could see that despite the increase in the number of nodes and replication factor, the performance of SandDB remains relatively constant.

Upon further testing, the performance remains relatively constant with about 30 rows per replica. Thus, this demonstrates that SandDB is scalable.

4. Code Repository

Our code repository can be accessed here: https://github.com/FolkLoreee/SandDB

5. Limitations and Future Work

5.1 Better Node Bookkeeping

For node revival and killing, our implementation currently utilizes direct message passing via HTTP for notifications. By right, this is not the case in a real-life production system. This is because the Gossip Protocol was a stretch goal of our project due to time constraints. As such, future improvements can include implementing the Gossip Protocol to properly manage additions and removals of nodes, implementing proper fault detection via timeouts, as well as implementing proper data redistribution during changes in the number of nodes.

5.2 Client's Delete Functionality

Currently, while the Anti-Entropy module provides the functionality to delete tombstones, a proper client-facing deletion functionality has not been implemented yet. For future work, a capability for the client to execute deletions can be added.

5.3 Anti-Entropy Implementation Optimizations

In our project, the Anti-Entropy module simply uses the MurmurHash3 hash function to hash each row and compares each row with its corresponding replicas. Further optimizations such as implementing a full Merkle Tree-based comparison method could be explored in the future. Various micro-optimizations within each sub-component of the Anti-Entropy module could also be implemented, as indicated by the various *TODO* comments in the *anti_entropy.go* source code file.

5.4 More Configurable Scopes for Anti-Entropy

Currently, the client can only choose either a full repair or a primary range repair for the Anti-Entropy module. More configurable repair scopes with varying ranges and granularities could be implemented in the future, whereby the client could choose which repair scope to execute. Taking inspiration from Apache Cassandra, other forms of repair could potentially include range repairs (with specifiable token endpoints), incremental repairs, and parallel repairs.

5.5 Fault-Tolerant and Efficient Storage System

In Apache Cassandra, the storage system is composed of append-only CommitLogs, in-memory write-buffer Memtables, and immutable SSTables on disk. The storage engine pipeline of Apache Cassandra is also highly optimized and consists of several components, all of which can be considered for future work:

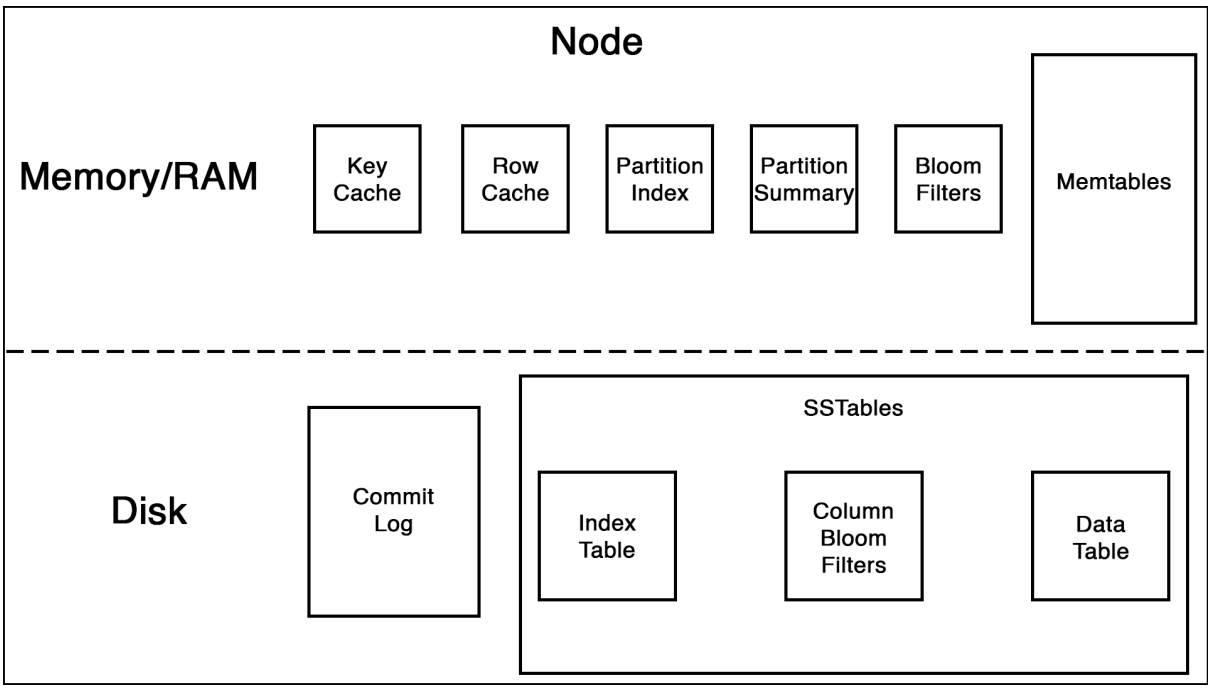


Figure 10: Apache Cassandra's overall storage engine and its components.

Meanwhile, our SandDB storage system simply utilizes several JSON files. This choice was made since we aimed to focus on the distributed protocols aspect of the project. Further optimizations could definitely be made to make data access from storage more performant and fault-tolerant.

5.6 More Consistency Level Options

In Apache Cassandra, the write and read consistency level can each be configured by the client to manage availability versus data consistency. Future work can provide several options other than QUORUM, such as ALL, ANY, or some fixed number of nodes specified by the client.

5.7 Virtual Nodes

For future work, virtual nodes could also be implemented. This is to allow each physical node to support multiple, non-contiguous token ranges. This would allow for better

performance during data redistribution/rebalancing and it also allows a more even distribution of data.

5.8 Hyperparameter Tuning

Hyperparameters might need to be tuned to achieve optimal performance. These hyperparameters include but are not limited to:

1. Garbage collection grace seconds
2. Request timeout values
3. The hash function used to distribute nodes and data

In our implementation, we have used basic or default settings for these hyperparameters. In real-life use cases, users should ideally profile the system using multiple different hyperparameter values and perform some prior benchmarking before deploying the system to production. The hash function used should also ideally be uniform so as to balance the request loads as much as possible.

5.9 More Extensive Testing

Due to the current physical limitations, the entire testing phase was carried out using different ports within the same machine. This might have materialized as biased/compromised performance results due to the limited processing ability of concurrent processes in a single machine, as well as the negligible network delay/latency of communication between local nodes. Hence, the recorded time taken for experiments may vary when separate machines are used. Further testing might be required to obtain more accurate readings of our SandDB system’s performance.

5.10 Byzantine Fault Tolerance

We would also need to handle the data stored in the database properly, in the case of potential malicious actors. This would certainly trade performance for the sake of security.

Apache Cassandra's initial design does not take Byzantine faults into consideration. To be fair, among the big, industry-used distributed databases in production today (e.g., DynamoDB, Bigtable, MongoDB, Cassandra, Elasticsearch), none of them is BFT. Indeed, almost all wide-area distributed systems in production are not BFT, including military, banking, healthcare, and other security-sensitive systems. That said, one paper (Friedman & Licher, 2016) proposed several methods to harden Apache Cassandra against Byzantine failures, and thus, future work could be done to implement these defence mechanisms.

6. Conclusion

Designing a distributed database system is essential to fulfilling the business needs of many companies and enterprises in today's day and age. This is because distributed, large-scale applications usually require a reliable and fault-tolerant method to store and process data. One way to do so is by following the design decisions specified by Apache Cassandra, as shown in our project.

In conclusion, we managed to implement a basic, proof-of-concept version of Apache Cassandra in this project as outlined by the Cassandra paper. The system faces many challenges such as consistency and fault tolerance, which a distributed system would typically encounter. The original paper has highlighted some methods to ensure consistency and fault tolerance with respect to database server nodes. However, some specific design aspects are not explicitly addressed in the original paper, such as how to handle data deletion via tombstones (Ellis, 2009) and a specific implementation of an anti-entropy mechanism. Implementing rudimentary and yet working versions of these unaddressed situations serve as our main contribution, hopefully inspiring and promoting further development in this area.

7. References

Apache. (n.d.). *Apache Cassandra*. The Apache Software Foundation. Retrieved May 4, 2022, from https://cassandra.apache.org/

DB-Engines Ranking - popularity ranking of database management systems. (n.d.). DB-Engines. Retrieved May 4, 2022, from https://db-engines.com/en/ranking

DB-Engines Ranking - Popularity Ranking of Wide Column Stores. (n.d.). DB-Engines. Retrieved May 4, 2022, from https://db-engines.com/en/ranking/wide+column+store

Ellis, J. (2009, March 7). *[#CASSANDRA-1] Remove support*. ASF JIRA. Retrieved May 4, 2022, from https://issues.apache.org/jira/browse/CASSANDRA-1

Ellis, J. (2013, September 2). *Why Cassandra Doesn't Need Vector Clocks*. DataStax. Retrieved May 4, 2022, from https://www.datastax.com/blog/why-cassandra-doesnt-need-vector-clocks

Ellis, J., Lakshman, A., & Malik, P. (n.d.). *Facebook's Cassandra paper, annotated and compared to Apache Cassandra 2.0*. DataStax Docs. Retrieved May 4, 2022, from https://docs.datastax.com/en/articles/cassandra/cassandrathenandnow.html

Friedman, R., & Licher, R. (2016, October 10). *Hardening Cassandra Against Byzantine Failures*. arXiv. Retrieved May 4, 2022, from https://doi.org/10.48550/arXiv.1610.02885

Spitzer, R. (2017, June 14). *Cassandra concurrent writes*. Stack Overflow. Retrieved May 4, 2022, from https://stackoverflow.com/a/44535904

When to run anti-entropy repair | Apache Cassandra 3.0. (2022, February 18). DataStax Docs. Retrieved May 4, 2022, from https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/operations/opsRepairNodesWhen.html