Cloud Databases - Project Report

Group 7

Mohamed Elgamal
Nico Daniel Denner
Ahmed Nader Mahmoud

Abstract

We report our work developing a key-value database system. A scalable non-relational based application, as traditional databases can go through bottlenecks in various applications. We implemented a system capable of storing a unique key-value pair, including a major feature; External Configuration Service (ECS). Our system was developed over the course of four major milestones with a fifth milestone to add additional novel features which will be discussed within this paper.

Keywords: cloud databases, noSQL, distributed systems, java

ACM Reference Format:
Mohamed Elgamal, Nico Daniel Denner, and Ahmed Nader Mahmoud. 2022. Cloud Databases - Project Report Group 7. In *Proceedings of Master Lab: Cloud Databases.* ACM, New York, NY, USA, 6 pages.

1 Introduction

Database is an accumulation of a large collection of data. Data Base Management Systems is a coordinated way of storing and retrieval a large amount of data. Traditional relational databases were invented in 1970 by E.F Codd. The term relational comes from a set of tables containing data categories or columns with similar relation. Although being conventionally accepted and regularly used, relational databases have major drawbacks including but not limited to [7]:

- Relational databases struggle with scalability, upgrading hardware can support with high scalability but then data should be distributed
- Data are stored in table forms, which would increase complexity in some cases
- Although relational databases contains multiple useful features. It is reported that a good majority of these features are not used, leading to extra cost and complexity
- Relational databases use Structured Query Language (SQL), which is advantageous with structured data but extremely disadvantageous with unstructured data.
- With big amount of data, servers are used to distribute them. Joining tables between distributed servers is a very challenging task

Thus, a strong demand to store and retrieve large data in a more optimized manner was emerging, paving the way to the formation of non-relational databases [6]. As they do not need a pre-written schema to add data nor change schema to alter existing data. Simply, with keys defined, identification keys and their corresponding data can be fetched. Non-relational databases are mainly divided into 4 categories with these examples [6] [1]:

1. *Key-Value Databases*: The most basic NoSQL database stores, data consists of a Key (*String*) and actual data content as Value, in a Key-Value pair. E.g. Redis, Riak
2. *Document Databases*: Document storage structure, each Document Store differs in its data implementation. Documents are encoded in a standard format such as XML, PDF, JSON, etc. E.g. MongoDB
3. *Column Oriented Databases*: Column Store Databases, unlike the usual Row Databases, storing data in a column structure. E.g. Cassandra
4. *Graph Databases*: Also schema-less databases, using Graph theory concepts, storing data along with nodes, edges and other properties. E.g. OrientDB, FlockDB

Having covered some context, we built our project using *Key-Value Databases*. In this database, we have the following basic storing components as a starting point:

- Insertion of a Key-Value pair
- Look-up/Search of a pair given a Key
- Update value of a pair given a Key
- Deletion of a pair

2 Design

As aforementioned, NoSQL databases are typically distributed databases, which imply having multiple instances (servers in our case) acting as storage pillars. Our system is designed as a mixture of client-server and peer-to-peer network. The multiple servers act as peers, which are distributing the data - and thereby the responsibilities - between themselves. To enable such a peer-to-peer storage system, we needed a service which manages the network and goes by the name of the External Configuration Service (ECS). Users can interact with the database in a client-server manner by using a client to connect to one of the servers in our system. In the following we will go into more detail on the three individual components of our system.

2.1 Server

The servers are responsible for storing the key-value pairs inserted by the clients. Since our system consists of multiple

Master Lab: Cloud Databases, 05.08.2022, TUM
2022.

servers, we needed a way to distribute the responsibilities between them in an equal manner. This is achieved by the concept of a hash circle. In essence, this circle splits the key range into individual blocks [2]. Each block is then assigned to one of our servers. If a client wants to read or write a data object, they are redirected to the correct server, responsible for storing the key-value pair. If a server is entering or leaving, the responsibilities are reorganized and the data is transferred to the correct server.

2.2 ECS

The ECS is the central organization unit in our system. Each server will contact it to either distribute data into the network or to notify it that a server is joining or leaving the network. The ECS manages the metadata, which essentially describes the current status of the hash circle. In addition, the ECS is concerned with failure detection. It is responsible to detect when a server has failed or is generally unavailable and unable to serve any client tasks. After detecting this failure it brings the system back into a stable state in which it can operate normally.

2.3 Client

The client can connect to any server and issue write or read requests on data objects. Most of the application complexity is intentionally distributed to the servers and the ECS to enable a light-weight client application, which can run on potentially any hardware.

3 Novel Extensions

During the fifth milestone we focused our attention on the refinement of the architecture that had been developed over the last few months. We thereby focused our efforts on making the system more consistent, stable and secure. This section will go into detail on each of the modifications.

3.1 Strong Consistency

Data consistency essentially describes the process of keeping your data up to date [10]. In a distributed storage system there is no one single location where a particular piece of data is stored, but potentially many locations since the data is replicated to multiple servers to ensure availability when a server dies. If an update to a particular data object is made, it takes some time until the whole network gets aware of this. Depending on your system and your use case there are different kinds of protocol which achieve data consistency, two extrem examples are the following:

Eventual consistency is a protocol which is quite relaxed when it comes to achieving data consistency [5]. The constraints are defined in a very vague way since the system only ensures that if there is no subsequent write to a data object, it will eventually reach consistency. This means that results are less consistent right after the update so there is a high probability that you will still receive the old data when issuing a read request. The benefit of this is that the overall latency of your requests are lower since the network is not so busy distributing the data update in the network. Our system used the eventual consistency protocol during the first four milestones and only distributed the data after some delay.

We changed our system to implement the strong consistency model during the last milestone. Strong consistency means that always the latest data is returned and therefore subsequent read request will receive the updated value [5]. Consequently, this method puts a higher load on the system since updates have to be propagated immediately to each entity. This protocol is often times favoured, in literature and industry, since data consistency is more important than network latency. Or put differently, there are other methods to reduce the latency of the system without compromising data consistency. This is the reason why we also decided to stick to strong consistency, since we did not observe a significant increase in latency. However, this might change depending on the concrete environment in which the system will eventually be used later on.

3.2 Gossip Protocol for Failure Detection

Failure detection is an essential task in any distributed platform. This enables us to get a clear picture of the network and to know which servers are still up and running and which are unavailable. There exist several different approaches to detect failing servers, each having its unique pros and cons.

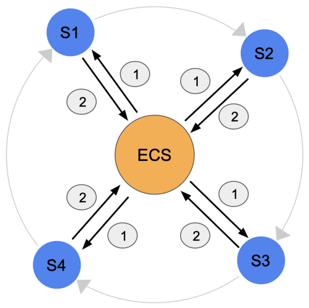

Figure 1. Schematic illustration of the heartbeat protocol. The ECS periodically sends out a heartbeat to each server and waits for their still-alive response.

Failure detection via heartbeat is one of the most simple protocols to recognize if a server is still alive or not [3]. The procedure is illustrated in figure 1. With this protocol, the network manager (in our case the ECS) periodically sends out heartbeat messages to the known servers in the system. Each server is then supposed to respond to this indicating that he is still alive. If such a response is not received in a pre-specified period of time, the manager assumes that the server has died and deletes it out of his list of running servers. This protocol is quite simple and easy to implement. However, it puts a significant load on the system by periodically sending heartbeat messages to each server. We tried to reduce this bottleneck by implementing a more light weight failure detection approach.

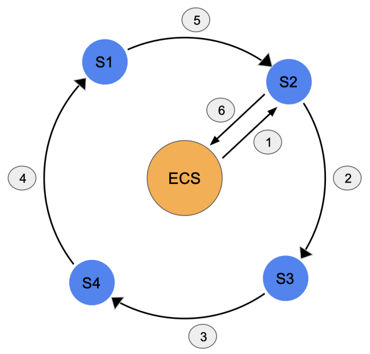

Figure 2. Schematic illustration of the gossip protocol. The ECS periodically sends out a message to one of the servers (server 2 in this case). This server then initiates a gossip round.

Our approach uses a gossip-style protocol to detect failures in the system. In general, gossip protocols are often described to behave like a virus [9]. They propagate information in the network by sending it to its neighbors, or also to random servers in the system. Our approach is more deterministic and utilizes the unique server structure of our system. Since our servers are arranged in a hash circle, as described in section 2.1, each server knows about his successor. We use this information to reduce the communication between the servers and the ECS. The procedure is exemplary illustrated in figure 2. Similar to heartbeat, the ECS periodically initiates a request to one of the servers to see if they are still alive. Different to heartbeat, in this case the ECS only contacts one of the servers (here server 2). This server then initiates a new gossip round. He contacts his successor to see if he is still alive. If this is the case he will in turn contact his successor and so on. Eventually, because of the circle, server 2 will receive a message from its predecessor. This symbolizes the end of one gossip round and now we know that all servers are still alive. The server then returns a success message back to the ECS. The main benefit of this protocol is that the main load is between the individual servers and not the ECS. Since the ECS is a single entity in our system, it is beneficial to minimize the load which is put onto him.

The scenario of an actual failure detection is shown in figure 3. This time the ECS contacts server 2 for a new gossip round. When server 4 tries to contact his successor (server 1), he does not respond. It then contacts the ECS that a failure has occured and the ECS acts accordingly and deletes the server out of the hash circle.

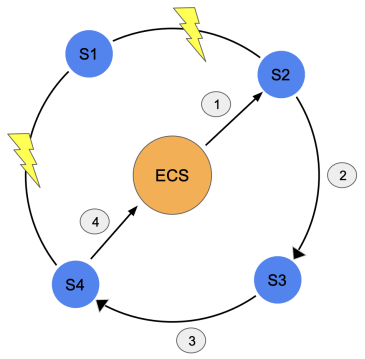

Figure 3. Schematic illustration of a failing gossip round because server 1 is not available.

3.3 Client Authentication

As a third feature, we built a client authentication method into our system. Our goal thereby was to enable restricted write access to the data. Only the author of a particular data object is allowed to alter or delete it, while other users are only allowed to read it. This is achieved through a two phase protocol, namely registration and login.

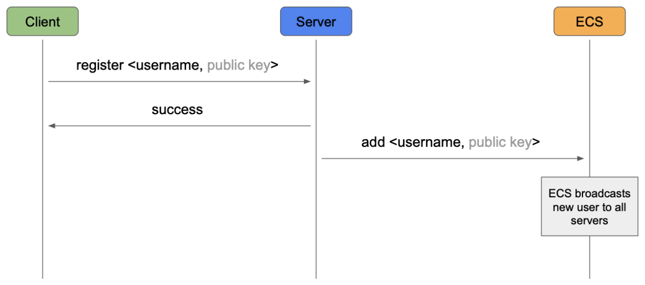
Client
Server
ECS
register <username, public key>
success
add <username, public key>
ECS broadcasts new user to all servers

Figure 4. Sequence diagram of the register protocol for client authentication.

The registration procedure is illustrated in the sequence diagram of figure 4. A client can connect to an arbitrary server in our system and can register a new user by providing its username and the public key of a newly generated PGP key pair. The server checks if a same-named user already exists in the system and accordingly will return a success (if no such user is available) or failure (if there is already a user with this name) message to the client. In case of success, he will add the corresponding username and public key pair to the system. This is done by sending it to the ECS which in turn then broadcasts the newly added user to all servers. The user is now able to login using the procedure illustrated in figure 5.

After connecting to any server, the user can login with this username. The server checks for the username and if it's found, it encrypts a random message with the saved public key corresponding to that username. The client receives this

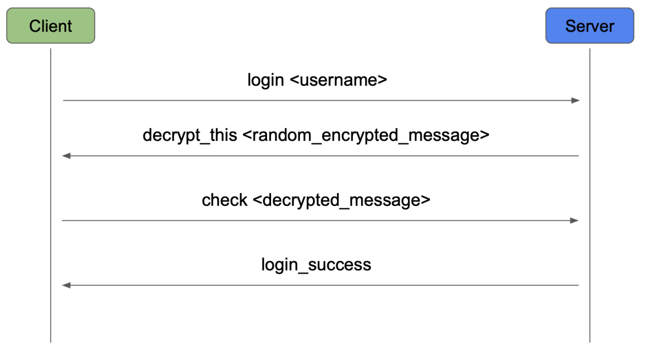
Client
Server
login <username>
decrypt_this <random_encrypted_message>
check <decrypted_message>
login_success

Figure 5. Sequence diagram of the login protocol for client authentication.

encrypted message and has to decrypt it using his private key and sends the decrypted message back to the server. The server compares the random message and the decrypted message from the client and if they match, then the client is logged in successfully. It has to be said that the protocol steps mentioned above all happen automatically after the initial login message was sent by the client. Each data object inserted into our system is now associated with a particular user, so that on every write request, it can be checked if the currently logged-in user has the right to alter the existing data object.

4 Evaluation & Analysis

We ran a number of tests to gauge and benchmark our system's performance on multiple fronts. Our testing machine is a laptop with the specifications in Table 1. It hosted all services, launched clients, servers and ECS. *Completion Time (Second)* is the main performance metric we used.
Carrying out performance tests we needed to populate our application with mock data. Firstly, we resorted to the Enron Data set [4], it contains more than 500,000 files. Each with varying content and text size. 1000 files were chosen randomly to operate on. The results of the tests using the Enron data set were not descriptive due to the hardware limitations (will be discussed later) and the large size of the Enron files. Eventually, the tests were redone using one thousand arbitrary key-value pairs whose size and content were randomly generated.

Table 1. Testing Machine

Processor	AMD Ryzen 7 3700U, 2.30 GHz (4 cores)
Disk	Toshiba XG6 512GB
Memory	8 GB – DDR4

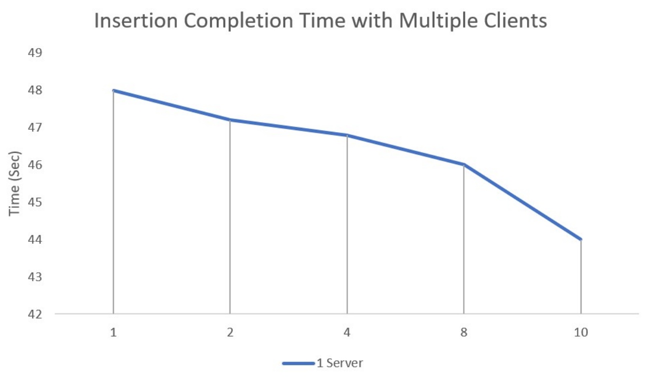

Figure 6. Performance evaluation (Completion time in seconds) with one server and increasing number of clients

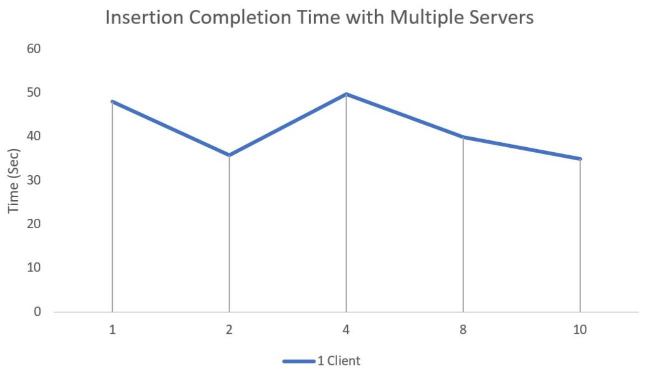

Figure 7. Performance evaluation (Completion time in seconds) with one constant client and increasing number of servers

4.1 Data insertion

Testing the performance for data insertion with a fixed number of requests, we covered the following scenarios:

- Single server hosting a single or multiple clients
- Single client connected to single or multiple servers
- Multiple clients with multiple servers

Keeping other parameters constant along these experiments, 100 as cache size and **FIFO** as a caching strategy, the results of the first scenario depicted in Figure 6 indicate that there is a slight improvement in terms of completion times when the number of clients hosted by one server increases. This scenario indicates that with a similar number of requests, the server handles requests divided among different clients and executed concurrently better than requests sequentially executed by one client.

The second scenario is presented in Figure 7 showing an interesting behaviour. With the increase of number of servers the faster the insertion takes place; 2 servers complete insertion faster than 1 server. However, when 3 or more servers are launched we observe a slight increase in completion time.

This is due to the shrinking hash range of servers, which forces the client to change servers when the inserted key does not belong to the current server. The introduction of the replication protocol led by the ECS can also be culprit in this scenario. Following that bump, there is an obvious linear-like performance improvement until 10 servers are launched.

Combining results from both scenario 1 in Figure 6 and scenario 2 in Figure 7, we reach the third scenario as demonstrated in figure 8. Showing that the increase in number of servers is directly proportional to improving the performance in terms of completion time. Similarly with the increase in number of clients.

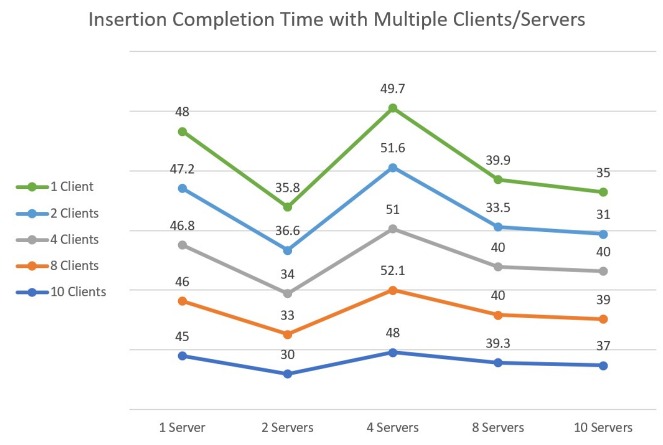

Figure 8. Performance comparison with different client/server combinations, completion time in seconds

4.2 Data Retrieval

Another major feature of this database application is the retrieval of the inserted data. We wanted to inspect how much quicker it would be to retrieve data with replication as a key component between servers. Retrieving also the same 1000 values inserted earlier. We took also 3 scenarios to cover most possibilities as shown in Figure 9.

- 3 Servers hosting 3 Clients (With replication feature not implemented)
- 3 Servers hosting 3 Clients (With replication feature implemented)
- 2 Servers hosting 2 Clients (With replication feature implemented)

In the first scenario, there is no replication implemented, 3 servers are launched hosting 3 clients. This scenario scores similar retrieval results as the third scenario. When only 2 servers are launched replication is not triggered as replication needs a minimum of 3 servers. In case of the second scenario with replication implemented and 3 servers launched, data replication is triggered and retrieval is then greatly improved. So it can be concluded that replication highly improves the retrieval performance.

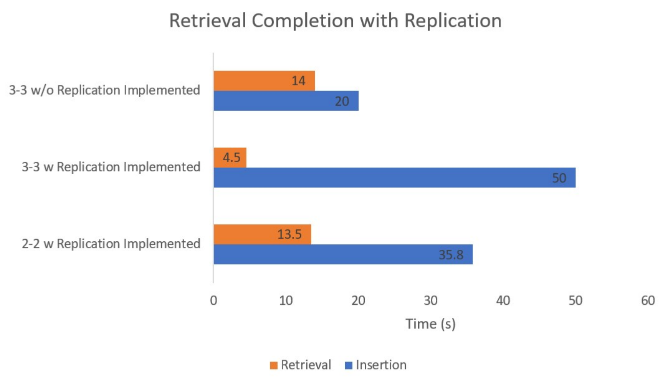

Figure 9. Data Retrieval after data replication. Blue bar represents the ordinary insertion completion time for context

4.3 Caching

Since previous experiments had the same caching settings, we evaluate our application with different caching parameters in this section, keeping the number of clients and servers constant (3 clients and 3 servers). As illustrated in Figure 10, the increase in cache capacity or size is directly proportional to a performance boost in terms of completion time. More data cached, less disk access, the faster the completion time. However, with different caching strategies, the 3 FIFO, LFU, LRU strategies all scored nearly similar results.

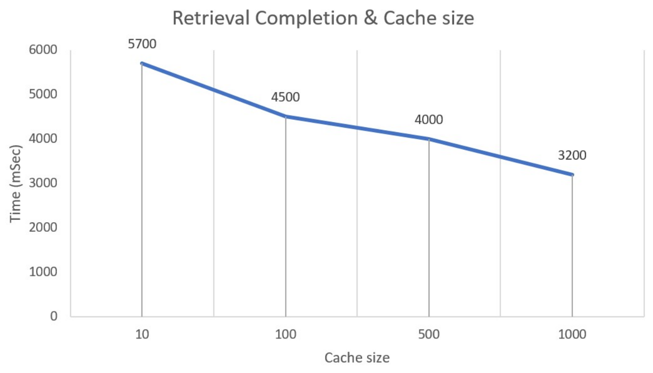

Figure 10. Different cache sizes affecting retrieval completion performance

5 Limitations & Future work

Throughout the system implementation, there were some limitations that would give large room for improvement if they were overcome efficiently. Three limitations and how they could be incorporated in future work are briefly discussed.

- **Hardware limitations**
  It has been very evident that the testing machine used through out performance evaluation has majorly contributed into limiting the capability of the this work's evaluation. Running multiple instances of servers on the same hardware will hardly reflect the correct display of the system's performance. Particularly when each of those servers run on different threads. Those threads may reach an eventual bottleneck situation or context switching. In general these limitations were also reasons why distributed systems were introduced in the first place.
  Distributed testing would come in handy to conquer the performance limitation. It is basically done by running multiple instances of the clients and servers on different workstations [8]. However, this could also turnout to be a costly operation. Another solution would be to use a virtual private server on the cloud with more hardware potential.
- **Single Point of Failure (ECS)**
  Our application depends on one single instance of an ECS, which subsequently makes the whole system prone to failure once that ECS fails leading to scenarios like servers failing to join the network, absence of metadata or replication failure.
  This could be remedied with a peer to peer network of multiple ECSs, however that would require a complete redesign of the our system.
- **Dynamic number of Replicas**
  Currently the replicas are set statically to three per each server data chunk. Ideally, we aspire, in the future, to replicate that one server's data into a dynamic number of servers depending on the available resources of the servers or the importance of the to-be-replicated data. Bringing more servers into the picture would speed up data reading tasks.

6 Conclusion

The recent transition from traditional relational databases to non-relational databases has not been accidental, NoSQL databases are known for their scalability and high fault-tolerance. They cover many different particular use cases and are fairly easier to deploy and construct compared to relational databases.

In this project we have implemented an end-to-end distributed and replicated **Key-Value** Database application with additional functionalities. **Stability** by detecting failing servers efficiently. **Consistency** by keeping strong data consistency without any loss. **Security** by establishing an authentication protocol for each client.

Acronyms

ECS: External Configuration Service
FIFO: First In First Out
LFU: Least Frequently Used
LRU: Least Recently Used

References

[1] Cristina Băzăr and Cosmin Sebastian Iosif. 2014. The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase. *Database Systems Journal* 5 (2014), 49–59.

[2] Juan Pablo Carzolio. 2022. A Guide to Consistent Hashing. https://www.toptal.com/big-data/consistent-hashing.

[3] Wei Chen. 1997. Heartbeat: A Timeout-free Failure Detector for Quiescent Reliable Communication. In *In Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97), Saarbruecken, Germany, Lecture Notes on Computer Science 1320, Springer-Verlag, September 1997, pp. 126-140.* (in proceedings of the 11th international workshop on distributed algorithms (wdag'97), saarbruecken, germany, lecture notes on computer science 1320, springer-verlag, september 1997, pp. 126-140. ed.). pp. 126–140. https://www.microsoft.com/en-us/research/publication/heartbeat-timeout-free-failure-detector-quiescent-reliable-communication/

[4] William W. Cohen. 2015. *Enron Mail Dataset.* Retrieved July 30, 2022 from http://www.cs.cmu.edu/~enron/

[5] Google. 2022. Why you should pick strong consistency, whenever possible. https://cloud.google.com/blog/products/databases/why-you-should-pick-strong-consistency-whenever-possible.

[6] Cornelia Győrödi, Robert Gyorodi, and Roxana Sotoc. 2015. A Comparative Study of Relational and Non-Relational Database Models in a Web- Based Application. *International Journal of Advanced Computer Science and Applications* 6 (11 2015). https://doi.org/10.14569/IJACSA.2015.061111

[7] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. 2012. A Survey and Comparison of Relational and Non-Relational Database.

[8] Omar Rafiq and Leo Cacciari. 2003. Coordination algorithm for distributed testing. *The Journal of Supercomputing* 24, 2 (2003), 203–211.

[9] High Scalability. 2011. Using Gossip Protocols For Failure Detection, Monitoring, Messaging And Other Good Things. http://highscalability.com/blog/2011/11/14/using-gossip-protocols-for-failure-detection-monitoring-mess.html.

[10] Scylla. 2022. What is Database Consistency? https://www.scylladb.com/glossary/database-consistency/.