Praktikum: Cloud Data Bases
Final Report

Aly Kamel*
aly.kamel@tum.de
Technical University of Munich

Iremnur Kidil*
ge34hoz@tum.de
Technical University of Munich

Ricardo Kraft*
ga84vax@mytum.de
Technical University of Munich

ABSTRACT

In today's world, the key to retrieve substantial information is communication. Software that enables messaging features, such as calling and video transmission as in Skype, sending emails with Google Mail, chatting via WhatsApp or Facebook and many more, ease the exchange of information and are therefore crucial for productive working in groups of many people. They are used for managing, discussing, preparing, executing an organizing parts of a project. On this basis we dedicated us in Milestone 5 to implement a feature that enables multiple clients to communicate via chatrooms.

This report begins with an introduction part, where different commercial databases are examined. Depending on our observations of how the commonly used database systems work, our motivation of developing the group chat extension is elaborated.

Moving on, the background containing CAP Theorem, ACID and BASE design approaches are clarified for the sake of a better understanding of the developed distributed and replicated database system.

Furthermore, our extension is examined focusing mainly on how the system works from the client's point of view. The features of the group chat and the implementation are studied in greater detail. In the implementation section a brief description of the replicated and distributed storage service from Milestone 4 is represented. The replication strategy is an important feature of database systems in general because it guarantees eventual consistency and basic availability which is crucial for distributed database systems.

The main idea with the group chat extension is that multiple clients can join a chatroom with a globally unique chatID, assigned individually for each room, and exchange messages there. Moreover, clients can perform read and write operations with the help of a chatbot while being in the chatroom.

Eventually we conclude our paper with a performance analysis by comparing results from our performance measurement tests and touch upon the advantages of providing a group chat.

1 INTRODUCTION

One of the main criteria when developing a key-value database(DB), is being able to handle large amounts of data while making the system available for multiple clients. DB systems, such as Amazon-Web Services, Microsoft Azure SQL and Oracle Database offer the combination of scalability and cost effectiveness.

Another important feature is elasticity which measures the ability of a system to dynamically scaled up or down. The key-value DB BigTable and PNUTS are great examples[3].

Amazon DynamoDB is a non-relational database system which provides single-digit millisecond latency [1], considering the system is used worldwide. Data records are stored with a primary or composite key and several attributes depending on client's request [8].

Apache Cassandra is another key-value based high scaled [2] DB system, which is known for managing some of the world's largest datasets on clusters with the help of thousands of nodes distributed amongst multiple data centres [6]. Cassandra offers more flexibility regarding fault handling and managing wide range data, unlike BigTable and Dynamo [8]. The system focuses on high level of availability by scaling millions of read and write requests per second[6].

Our inspiration mainly came from the DB Redis. It rose in popularity after its creation in 2009 and got deployed by many big companies such as Instagram [9] and Twitter [12], which require enormous DBs and fast responses to the millions of users they serve. One of its multiple features is the Pub/Sub system [11], added in March 2010 [10]. With it, clients are able to subscribe to a set of keys, namely topics, and receive an update whenever their associated values get overwritten. In this model, clients keep waiting for the server to send them an update. During that period, subscribers are expected to use only subscription related operations.

In our system, we wanted for there to be no distinction between publishers and subscribers with everyone being able to both publish and receive updates. Apart from that, access to the DB should still be possible.

Thus, the way we modelled our system in the end, was having a many-to-many relationship of all subscribed clients, effectively, implementing a group chat functionality.

2 BACKGROUND

In this section, we elaborate on some of the basic concepts that our key-value system depends on. First, we discuss the CAP theorem and analyse where our system lies on that spectrum. Then, we mention two design paradigms, namely BASE and ACID, and explain the differences between them.

2.1 Key-value store

Key-value stores are perhaps the simplest type of a NoSQL DB. Every information is saved in the form of key value pairs. Typically the key's memory space is very small compared to the one of the values. The key is used as an identifier to the actual data saved in the value. The client will always need to have a key in order to work with the DB.

In our case values are limited to strings and can be easily stored and retrieved just by providing the corresponding key, which is also limited to a string.

*Authors contributed equally to this work.

2.2 Caching

Caching is a way of storing information so that information can be retrieved faster on demand. A storage method, namely write trough, enables to write the information into the cache as well as in to the disk store simultaneously. Thus, the data will not only be returned quickly because of caching, it will be saved in the disk ensuring that no data will get lost in case of any partitions in the network. Since the write operation is performed two times the storage system has redundancy. One of the upsides of redundancy is tolerating loss of data. However, it increases the latency by making the response times longer.

2.3 CAP Theorem

According to the CAP Theorem, distributed DBs have three substantial properties to consider: consistency, availability and partition tolerance[4]. Eric Brewer, the man behind the CAP theorem, stated that a distributed database can fulfil at most two of three properties[5]:

- Consistency:
  Clients are provided with fresh data, meaning the most up-to-date version of the data that got stored after the last write operation.
- Availability:
  Servers respond to the request of clients at all cases. Every non-failing node in the system must be able to serve the client in a reasonable amount of time[7].
- Partition tolerance:
  Whenever a server crashes, the rest of the system will still stay functional after the crash, without any information being lost.

Twelve years after from proposing the CAP theorem, Brewer mentioned that software engineers do not have to strictly abide to the 2 of 3 principle, it is rather a spectrum than a binary choice [4]. In other words, a distributed database system can favour high level of consistency and partition tolerance by having low level of availability. Thus, the initial theorem is improved by not having to sacrifice availability completely in this case.

There are two design approaches for distributed database systems, namely ACID and BASE. According to Brewer, these two design approaches may be referred as opposites of each other[4] because of their priorities and use cases. ACID approach is used most of the times for relational database systems (SQL) and focuses on consistency to maintain reliability. On the contrary BASE is more suitable for the non-relational database systems (NoSQL) concentrating on providing the client high level of availability[5].

2.4 ACID

ACID is a traditional design approach[4] when it comes to large-scaled distributed systems. The main goal of ACID is that despite the system having partitions, the client should always be provided with consistent values. It is an acronym which stands for the four following properties:

- Atomicity:
  A set of transactions succeed all at once or fail all together. In other words, it is an all or nothing strategy.
- Consistency:
  The system never contains any stale data. When a client wants to read from any server, the returned value must be the value from the last write commit by any client. All clients should always get the same result whenever they try to fetch the same information.
- Isolation:
  All transactions happen isolated from each other and do not affect each other. Thus, if one transaction fails, all others are undisturbed.
- Durability:
  Once a client is informed that a transaction has been successfully committed, even if a crash occurs, the transaction would still stay committed.

2.5 BASE

BASE, another design approach defined by Brewer, offers looser requirements than ACID[4]. Non-relational database systems concentrating on high availability make use of the BASE approach, which sacrifices strong consistency in favour of being always accessible. Examples include huge storage services such as Amazon's Dynamo, Facebook's Cassandra or Google's BigTable, where millions of active users always expect the service to be available[8]. Nevertheless, there must be trade-offs. By not guaranteeing the consistency at all time, the system cost is reduced, and clients are happier, but a client might not get an immediate response to his request.

- Basic Availability:
  Clients are guaranteed to get a quick response from the server without getting blocked, however the returned value may be stale, stated in other words, inconsistent with the latest version.
- Soft State:
  Since stale data is permitted, servers never truly know whether they are currently up-to-date or if they still contain invalid data.
- Eventual Consistency:
  If there are no updates in the system for a long time, then all servers will gradually become consistent. Since the time is not specified, servers are always in a soft state, as explained above.

3 KEY-VALUE SERVER

Before we get into the details of MS5, we would like to revisit the implementation of the distributed key-value storage system we were tasked to develop and briefly discuss some of its features. Figure 1 displays how a client interacts with its Connection Handle Thread (CHT), the helper thread assigned to serve it. In the following sections, we take a deeper look at how the client side and storage system visible in 1 are structured and how they function.

3.1 Client side

The client side consists of the three following main components as seen in figure 2:

(1) *ClientApp*: represents the client interface which allows input through the console. From there the client is able to issue

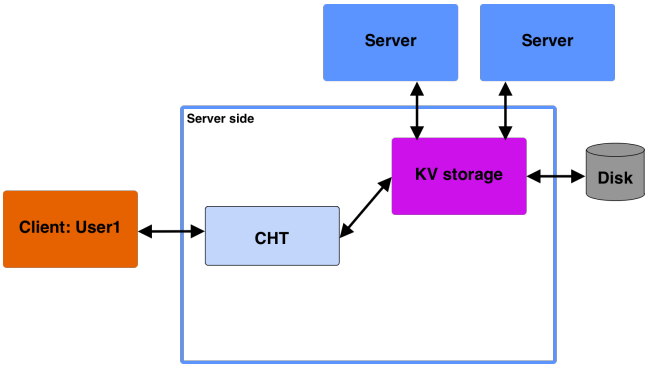


Figure 1: Key-value server architecture

Figure 2: Client side architecture

commands to connect to and disconnect from the system, interact with the key-value store and chat. The input is then checked and parsed before getting sent to the ClientLibrary. The result of the user's request gets displayed on the console through the ClientApp.

(2) *ClientLibrary*: serves as a bridge between the client and the server. It contains an instance of the hash ring and is responsible to connect to the correct server.
(3) *ActiveConnection*: abstracts the TCP socket connection to the server which allows the client to exchange data with the CHT without worrying about the underlying structure.

3.2 Server side

As previously mentioned, the CHT waits on the socket for the client to send a request. Afterwards, the request is forwarded to the key-value processor (KVP), which parses the client's string and translates it into direct DOPs. The ServerRing, which contains information about the hash ring, is called to check whether the current server is responsible for performing the requested operation on the provided key or not. In the former case, the request gets transferred to the KVStore. There, read operations get passed on to the cache and only then to the DiskStore, in case of a cache miss. Write operations instead are redirected to the cache, the DiskStore and the Replication Manager (RPM). The DiskStore creates at the start of each server a directory containing three subdirectories, one for storing the keys it has write access to and one for each of the key sets it has only read access to. A file inside one of those directories represents a key-value pair with the key being the file name and the value the file content. The RPM is in charge of both sending updates to the server's replicas and receiving updates sent by the responsible servers.

[moved parts from background to here] (not complete) Since we do not block clients from accessing the DB, CAP Partition: This is mainly secured by replicating data across multiple servers. ACID Durability : This is ensured by persisting data permanently on disk.

With Milestone 4, replication is added to the system with the intention of increasing the availability by distributing the data records to the two replica servers. Redundancy comes along with the replication strategy, however when a node crashes or fails, read operations can be processed via replicated servers. Our system conforms to the first property of BASE by being generally available, because servers respond to clients requests even if a latency occurs. The group chat extension, later to be discussed thoroughly, preserves the same requirements.

Eventual consistency is achieved within the system due to requirements of Milestone 4. If there are no updates for a long time all the replicas will become eventually consistent by updating key ranges of the coordinator and replica nodes placed on an hashring. In order to maintain strong consistency as described in 2.4, a client performing a PUT operation would have to also wait until the value gets sent over to both of the servers replicas, which would result in slower PUTs. Since our system is not designed to handle important real-time transactions, such as in a banking system, eventual consistency satisfies our database needs while allowing faster response times and higher availability.

Since the implementation of Milestone 3, it is possible to monitor key-value stores continuously. The ECS pings each key-value server every 700 ms to be informed about the servers' availability. If a server does not respond to the ping within a given period, it is considered shutdown. With replication active, single failing server nodes can be tolerated, thus partition tolerance is provided as well. Since each piece of data exists on three different servers, the only way information can get permanently lost is if all three responsible servers crash simultaneously with no time in between to transfer data to a running server.

One of the downsides of our system is that the chat system runs just one of the requested servers by the client. Thus, partition tolerance may not be covered in case of that serving crashing. As in the most database systems there are trade-offs between the properties: availability, consistency and partition tolerance.

4 GROUP CHAT

With the MS5 we extended our program with a group chat feature which enables clients to exchange messages with each other through chatrooms. Additionally, we developed our group chat in such a way that clients can access the DB while being in the chatroom with the help of a chatbot.

In the following, the basic functionalities of the group chat are described step by step, subsequently the chat system and its components are examined in depth. Lastly, we discuss how client can access the DB while chatting.

4.1 Basic Functionalities

To start off, in the same sense as Milestone 4, the External Configuration Service (ECS), where the storage servers are monitored and controlled, is started. Following that a number of servers are created. A client must connect to one of the servers.

4.1.1 Unique Username. After successfully connecting to a key-value server, the client has to either enter a username or use the

command "QUIT" to have a username randomly assigned to him. Usernames are implemented as globally unique identifiers for the clients, which prevents different clients from having the same username in different chatrooms. That way, users are guaranteed to know who they are communicating with, as long as their partner is connected to the system. In order to avoid unnecessary extra connections, the ECS stores a list of users. Whenever a client connects and tries to set its username, the request gets sent through the server to the ECS, which then checks if the username has been already taken by a client connected to any of the servers online. The end user either receives a welcome message with the username displayed if the operation was succesful, or an error message.

4.1.2 Chatrooms. In order to use to start chatting, the client has to join a chatroom. This is done by typing `chat` and providing the ID of the chatroom. In case the room does not exist, the user is allowed to create one. The next time a user executes the `chat` command with the same chatID, he is either granted access to it immediately if it is a public room, or is otherwise first required to enter the password chosen by the person who created the room. The password is hashed on the client side for safety measures and only the hash is required to validate the password attempt. Each server owns a list containing the active chatrooms that it is responsible for depending on its position on the hash ring. In order for a client to join one of those chatrooms, is that it would first need to connect to that server, similar to the way storing key-value pair works.

4.1.3 Saved Messages. Messages sent in a chatroom get stored into a text file under the directory of the responsible server. We provide this feature, in case a network partition occurs in the system, so that we are able to restore messages get lost.

4.1.4 Additional Commands. The `WSP` command allows a private message to be sent to multiple users. Whenever a client joins or leaves a chatroom, all users inside get notified. This helps the users keep track <- add something here Otherwise, users could use the `ACTIVE` command to obtain a list of all users currently in the chatroom.

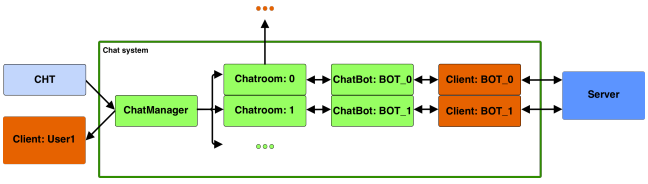


Figure 3: Chat system

The chat system consists of three main components as seen in the above figure**??**:

(1) *ChatManager*, who is in charge of providing all chat functionalities to the client, including connecting to a chatroom and sending messages.
(2) *ChatRoom*, which is identified by a chatID. A Chatroom object contains a map of all connected users and their sockets, in order to be able to forward messages.
(3) *ChatBot*, which is responsible to execute PUT and GET commands during a chat session. The chatbot acts exactly like a client, meaning it first has to connect to the responsible server and then send its request.
By utilizing a chatbot, chat requests containing a key-value operation are slowed down, since only one "client" is performing those operations. However, if each chat user is tasked with executing the commands himself instead, then whenever a key is located at a server different from the server responsible for the chatroom, the client would need to reconnect twice. Not only that, but also all chat messages sent while the user was disconnected would either get lost or will be displayed after the user has reconnected, which would make the user temporarily unavailable. As mentioned in section 2.3, our system focuses on being available.

4.2 Database Access

In order to provide the chatting functionality, the server would need a socket to send to and receive messages from. However, the client also requires a socket to connect to the DB. The obvious design decision would be to add an extra socket to the client responsible for just chatting. The drawback of that idea is that the amount of TCP connections per client would get doubled, which would increase the network traffic to a large extent. In our intended use case, clients are not expected to utilize the DB heavily during chat sessions. For that reason, we could allow clients to quickly leave the chatroom to perform the DOPs and then rejoin. However, that might cause the client to miss messages while he is away, which affects the system's availability guarantee described in

The approach we decided to follow in the end was creating a chatbot, whose sole purpose is executing DOPs for chat users. The chatbot acts mostly similar to a client by being able to send read and write requests to the DB. Every chatroom gets a chatbot assigned to it, which scans all incoming chat messages for DOPs, signaled by the keywords `PUT` and `GET`. Those requests are then sent to the DB and the result is returned to the chat user. In case of a read request, the `GET` operation in the message is replaced by the retrieved result. Only after this translation will the chatroom send the now modified message to every participant in the chatroom. This allows users to use stored values for communication.

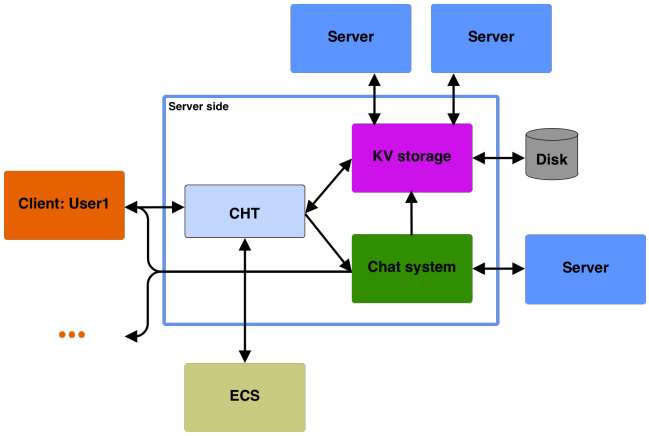


Figure 4: Server side architecture

delete?? chat system: All chat related requests get transferred by the CHT to the ChatManager. This has a list of Chatroom objects. When a client tries to connect to a chatroom, the ChatManager checks the list whether a chatroom with the provided chatID already exists or not. In the former case, if the chatroom is public, the client is granted access or, in case of a private chatroom, requested to enter a password. In the event of no chatroom found with the given chatID, the client is allowed to create the chatroom by specifying its type and providing a password, if a private chatroom is chosen. In order to increase security, the password gets hashed on the client side and only the hash of the password is stored at the server side and used for validation.

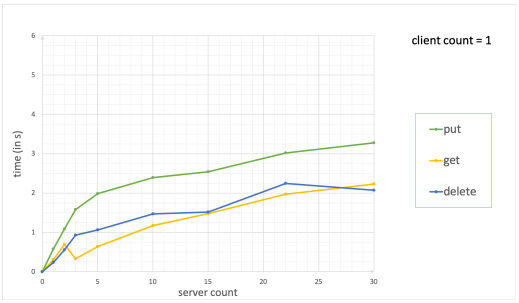
5 PERFORMANCE ANALYSIS

After we were done with the implementation of MS5, we wanted to test our system to 1) analyze the response times of normal read and write operations, and 2) check the impact of the chatting functionality on the response times.

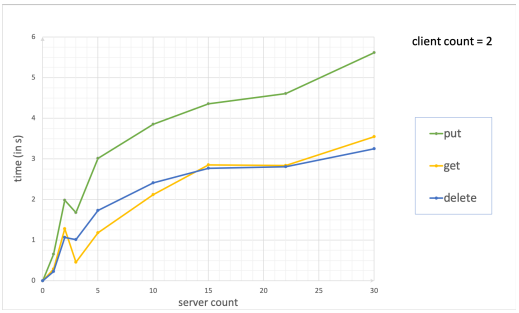
320 key-value pairs are used as input for the tests, which were derived from the Enron email set. The way the test works is that clients execute PUTs on all keys, then they try obtaining them back with GETs and in the end, all keys get deleted. Each of the three different operations got timed seperatly, so that their efficiency could be compared. The cache replacement strategy in place was Least Recently Used (LRU), which is the default strategy, and the cache size was set to 16, which represents 5% of the storage size. Before a new set of commands got started, the list of keys got shuffled, which causes both the degree of cache utilization during read operations and the chance of reconnecting to the responsible server to be based on luck. In order to avoid skewed results, each data point is calculated by taking the average of five test runs.

In figure 5 we vary the amount of key-value servers and measure the time 320 operations took. At first glance, it is clear that adding an extra client in 5b leads to lower response times, since double the amount of requests need to get processed by the servers. Otherwise, in both situations PUT operations take the most time, because each time the disk has to be accessed and a file has to be written. Even though deletetion manipulates data on disk as well, files just have to be located and removed from the directory, with no need to open them. GETs usually took the least amount of time, since disk access is not always necessary. A significant drop of read response time can be noticed at the three server mark due to replication just getting started. At that point, no reconnection is required for the read commands, owing to the fact that all three servers are now responsible for read operations on all keys. The chance of a reconnection being needed is equal to $1 - \frac{3}{N}$ with N being the amount of key-value servers running. As apparent by the formula, the chance decreases as the server count increases, leading to the rise seen later on in the graphs.

After we have evaluated the response time for the normal database operations, we try executing these same commands but inside a chatroom. For our testing purposes, we had two clients in the same chatroom perform those commands. The difference here is that, unlike in 5b where each client directly contacts the key-value servers, only one common bot is utilized by both users. After each command, the user would send a certain number of chat messages



(a)



(b)

Figure 5: Key-value servers performance

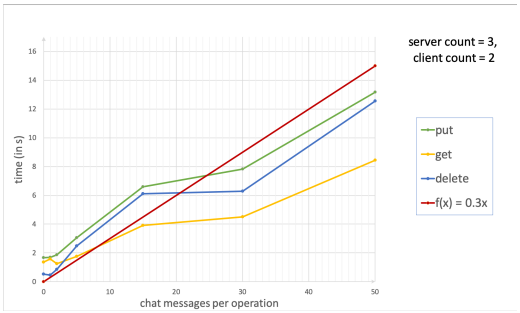


Figure 6: Key-value servers performance in chat mode

as depicted on the X-axis, in order to simulate a general use case scenario of our system. Graph 6 depicts the increase in response time in relation to the chat messages sent. When few messages are sent, the extra time required compared to the normal database operations seems unnecessary. However, the response times do not grow at the same ratio as the chat activity, but rather roughly follow the path of the function $0.3x$. That means that for each additional chat message per operation per client (equal to overall $320 * 2 = 640$ extra messages), the system is slower by just about 300 milliseconds. In one extra second, more than 2100 messages can get exchanged.

6 SUMMARY

To conclude our paper, the benefits and weaknesses of our implementation are evaluated thoroughly.

The decision, to make the chatrooms accessible only from one server has both its advantages and disadvantages. For one, it reduces

the complexity of the system because otherwise servers would need a way to exchange updates regarding the active chatrooms and chat users whenever a user joins or leaves. Our idea for the chatting functionality was for it to be as light-weighted as possible with clients entering and leaving chatrooms regularly.

A problem with our implementation is that if the chatIDs are not equally distributed across all servers, which may occur due to the unpredictable nature of the hashing function, a single server could then be in charge of most chatrooms. This would cause that a server to be overloaded with requests and would lead to greater response times and, in the worst-case scenario, would result in a bottleneck for the whole system. In order to combat this issue, we limit the number of chatrooms belonging to one server to 15 and the number of users in a single chatroom to 30. This means a server is responsible for up to 450 chat users. These limits could also be easily changed depending on the intended use case of the system.

Considering the implementation of the chatbot, it is a bottleneck in the system. If the maximum amount of clients start doing their own DOPs, all operations will be executed just by one chatbot. Thereby, the workload of 30 sockets will be pressed into one. This leads to a significant efficiency loss. Heavy modification and updates of the DB should be done outside of chatrooms, to make full use of the clients individual socket.

Another downside of our system is that the chat only allows words as strings. Nowadays, it is standard to be able to send other data types such as pictures, videos or files. Therefore, our system is more favourable for people who do not need to use data types other than strings.

Finally, we highly recommend our extension especially for light-weighted communication.

REFERENCES

[1] [n.d.]. *"What Is a Key-Value Database?"*. "https://aws.amazon.com/tr/nosql/key-value/"

[2] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[3] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J Elmore. 2011. Database scalability, elasticity, and autonomy in the cloud. In *International Conference on Database Systems for Advanced Applications*. Springer, 2–15.

[4] Eric Brewer. 2012. CAP twelve years later: How the "Rules" have changed. *Computer* 45, 2 (2012), 23–29.

[5] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '00)*. Association for Computing Machinery, New York, NY, USA, 7. https://doi.org/10.1145/343477.343502

[6] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. 2015. A big data modeling methodology for Apache Cassandra. In *2015 IEEE International Congress on Big Data*. IEEE, 238–245.

[7] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News* 33, 2 (2002), 51–59.

[8] Sultana Kalid, Ali Syed, Azeem Mohammad, and Malka N Halgamuge. 2017. Big-data NoSQL databases: A comparison and analysis of "Big-Table","DynamoDB", and "Cassandra". In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)(*. IEEE, 89–93.

[9] Mike Krieger. 2011. *Storing hundreds of millions of simple key-value pairs in Redis*. https://instagram-engineering.com/storing-hundreds-of-millions-of-simple-key-value-pairs-in-redis-1091ae80f74c

[10] Salvatore Sanfilippo. 2010. *Redis*. http://oldblog.antirez.com/post/redis-weekly-update-3-publish-submit.html

[11] Redis Development Team. 2020. *Pub/Sub Specification*. https://redis.io/topics/pubsub

[12] Yao Yu. 2014. *Scaling Redis at Twitter*. Youtube. https://www.youtube.com/watch?v=rP9EKvWt0zo