Cloud Databases Milestone 5 Final Report

Group 12

Şevval Palabıyık*
Technische Universität München
Munich, Germany
sevval.palabiyik@tum.de

Başak Akan*
Technische Universität München
Munich, Germany
basak.akan@tum.de

ABSTRACT

The modern society collects data to learn, profit, construct systems and simplify life. Technology has advanced to the point that data can be acquired easily, and it is improving every day. To facilitate and use the data collected, we store it in databases. As a result of increasing variety of data types and cheap storing options, nonrelational (NoSQL) databases became popular. In this study, we describe our novel architecture of a key-value storage system which is one of the least complex types of NoSQL databases. A literature review based on key-value stores has been performed on cloud databases to provide a general vision. We designed and implemented an example system to introduce the advantages, disadvantages, and challenges of applying them. In conclusion, the performance of our implementation is presented and evaluated.

KEYWORDS

Distributed, Replicated, Key-value Store, Failure Detection, Password protection

ACM Reference Format:
Şevval Palabıyık and Başak Akan. 2022. Cloud Databases Milestone 5 Final Report: Group 12. In *2022, Munich, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The concept of a database existed long before computers were invented. To archive data in an easy-to-find layout, index file card systems were used. The concept of a Database Management System, or DBMS for short, was born in the 1970s when IBM chose to develop this idea into a computer solution. In 1974, IBM released System R, a prototype relational database model that subsequently became SQL [13]. Oracle, on the other hand, was the first to commercialize the technology in 1979, making relational databases the most used form of bulk data storage on the internet [16]. Traditional SQL databases give ACID (Atomicity, Consistency, Isolation, and Durability) guarantees, which are considered critical for several use cases (i.e. bank systems). However, in terms of data models and scalability, these databases are severely limited. According to the aforementioned criteria, SQL databases are incapable of processing large amounts of data.

In the 1990s, Object Oriented Databases became popular, leading to the present implementation of NoSQL [10]. NoSQL is gradually becoming more commonly used than relational database implementations. There are several types of NoSQL databases: Key-Value Stores, Wide-Column Stores, Document Stores, Graph Databases, and Search Engines. This technique has been adopted and promoted by companies like Amazon [12] and Google [11], as well as opensource groups like Apache [14]. Data replication is used by many of these systems to achieve availability and fault tolerance. BigTable [11] by Google was an early method that helped define the NoSQL key-value data storage market. Another solution is Amazon's Dynamo [12], which provides an eventually consistent replication mechanism with customizable consistency levels. Cassandra [14] and Voldemort [9], two open-source Dynamo versions, combine Dynamo's consistency methods with a BigTable-like data structure. To ensure a proper distribution of key ranges (data partitions, or shards) to storage nodes, these systems use consistent hashing. Contrary to the SQL databases ACID guarentee, NoSQL databases provide BASE (Basically Available, Soft-state and Eventual Consistent). NoSQL databases built by the partnership of academia and industry provided more available and efficient, but certain level of consistency. However there is a trade of between Availability/Efficiency and Consistency of NoSQL databases.

When embarking on our key-value store implementation we decided to target eventual consistency. One of our design goals was to use a standard data model and cross platform API. We thus developed our model based on the provided infrastructure. Another design goal was to use consistent-hashing based scheme for key range partitioning. We arranged all storage nodes clockwise in a logical ring topology. Each position in the ring corresponds to a particular value from the range of a hash function. Message Digest Algorithm 5 (MD5) used for all hash operations.

In this paper, we aimed to examine distributed and replicated key-value storage systems over our novel implementation.

Our key contributions in this paper are:

- A high performance data replication
- Distributed nodes over ring based consistent hashing
- Password protected key-value storage system
- Notification mechanism for specified keys

The remainder of this paper is organized as follows. In Section 2, we introduce the background information about distributed and replicated key-value databases. Section 3 explains the design of our novel implementation of key-value store. Section 4 discusses our failure detection and recovery strategies. Section 5 explains newly added improvements for explained key-value storage design.

*Both authors contributed equally to this research.

Cloud Databases '22, February, 2022, Munich, DE

ACM ISBN 978-1-4503-XXXX-X/22/02…$15.00
https://doi.org/10.1145/1122445.1122456

Section 6 gives the experimental results. In Section 7 we conclude the paper. Section 8 discuss the future research directions.

2 RELATED WORK

Key-value stores are the simplest NoSQL databases. Every single record in the database is stored in a unique key together with its value. In this model of storage, no query language is required. Using simple put, get and delete commands, data is retrieved in a fast manner. Some of the most popular and widely-used databases are in key-value format due to its high performance, efficiency and scalability. This section gives a brief overview on some of the key-value stores.

Redis is an open source multi-purpose database that can also be used as memory cache and message broker. It provides many different data structures such as strings, bitmaps, geospatial indexes etc. Redis provides high availability, built-in replication, as well as automatic partitioning [3].

Amazon DynamoDB is a widely-used peer-to-peer key-value database that is fully managed by AWS. It has advantages such as automatic replication, scalability and high availability. The low latency it provides is result of its in-memory caching makes Dynamo a good choice for mobile, web, gaming and many more applications [4].

Memcached is an open-source general purpose memory object caching system intended to be used for speeding up web applications by caching data and objects in memory. It provides high performance, scalability and ease-of-use [6].

Apache Cassandra is an open-source NoSQL distributed database that is designed to provide scalability, high performance and high availability. It comes with a simple query language and offers replication as well [1].

To compare some of the aforementioned databases, they have different advantages and disadvantages which makes them all useful in different scenarios. For instance, Redis is more flexible than Memcached, however, Memcached has serious advantages when working with big data due to its multi-threaded nature [5]. DynamoDB can be a wise choice if the applications environment relies on AWS solutions. The main approach we follow in our work is providing persistent, scalable and replicated key-value store alternative that adds the features of password protection and Key Subscription.

3 DESIGN

The system is basically comprised of four primary components.

- **Client library**, which is used by client applications on the client-side,
- **Key-value storage server (KV Store)** that can run on multiple distinct physical servers,
- **External Configuration Service (ECS)** which mainly manages metadata information over distributed servers
- **Broker Service** which mainly manages notification mechanism over distributed servers and clients

The client library provides *get*, *put*, *delete* capabilities for client applications. Distributed server meta data information which can be provided from any server is stored and used for detecting and connecting to responsible server for each key-value tuple. Additionally, it provides *password protection* capability for high secure request transmission and *notification mechanism* for subscribing specified keys.

The KV Storage servers are responsible for handling requests from clients. It is design to satisfy concurrent *get*, *put*, *delete* requests from multiple client applications at the same time. When *password protection* is enabled it ensures client requests are protected by a password which is defined by clients. Each KV Storage server is responsible for a specific range which is calculated based on it's hashed address and port number.

The ECS is responsible for distributed KV Servers metadata information management. When a new KV Server is added to or removed from the system (can be planned or unplanned), ECS broadcasts the metadata information to all KV Servers in the system. Additionally, KV Server failures can be detected by ECS's heartbeat signals which are produced in every second separately for each KV Store in the system.

The Broker service is responsible for notification mechanism. Clients can subscribe to a key through this service and get notified by this server as well in case of any subscribed key updates and deletions.

3.1 Data Storage

Our data storage system consists of a cache mechanism that stores a predetermined number of key-value pairs in main memory and a persistent storage system that writes data on disk.

3.1.1 Persistent Storage. Our database system uses a generic data file to store each key-value pair on disk. Every put request creates a new file on disk with the name of the key, and inserts the value into the file as content. If a file with the given key already exists, the old value is deleted and the new value is inserted. This permits fast insertions.This approach allows fast lookups and deletions as well since a file is searched by its path which includes the unique key. The downside here is the slow disk I/O. When the key is not in cache, I/O access will occur. Compared to a hit in cache, it can be very slow.

3.1.2 Cache. In order to adapt to various use cases, three displacement strategies *Firs In First Out (FIFO)*, *Least Recently Used (LRU)*, and *Least Frequently Used (LFU)* strategies are implemented. All three cache strategies implements a base *Cache inferface* which provides following capabilities.

- **Put**, puts a key-value pair into the cache
- **Delete**, deletes a key-value pair from the cache
- **Get**, gets the value to a key from the cache
- **InitCache**, initializes the cache data structure for given maximum size
- **GetInstance**, return the cache instance based on initialization-on-demand holder idiom

When FIFO strategy is chosen as cache displacement strategy, the cache stores key-value pairs in a double ended queue (deque) data structure which is a linear collection that supports element insertion and removal at both ends. Most Deque implementations place no fixed limits on the number of elements they may contain, but used interface supports capacity-restricted deques as well as those with no fixed size limit. In order to provide efficient look up, we use a hash map that indexes the deque. As a result, when looking for a key, the map can be searched in constant time to retrieve a reference to the appropriate key-value pair. A similar design is

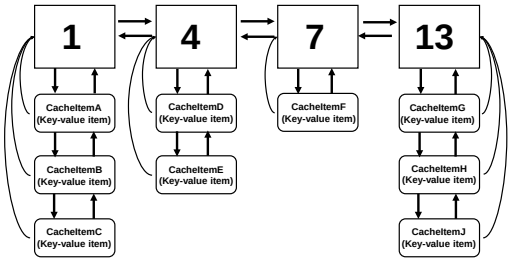

Figure 1: LFU Frequency List

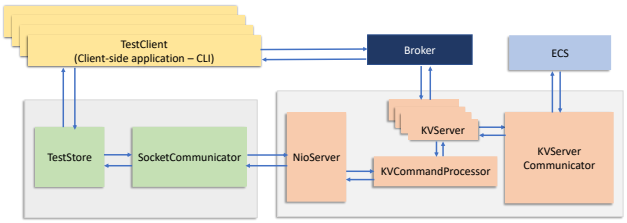

Figure 2: The Program Flowchart

used for the LRU strategy implementation, a deque data structure as well as a hash map data structure is used for same purposes. Since used deque data structure have the capability of supporting both FIFO and LRU strategy logic. In many places, the LRU or FIFO algorithm is preferred over LFU algorithm because of its lower run time complexity of *O(1)* versus *O(logn)*. We design our LFU cache displacement algorithm to have a run time complexity of *O(1)* for all its operations which we inspired from Matani et al. [15] and Eftimov et al. [7] implementations. A hash map is used to store all of the items with a key that is processed through hashing algorithm and the value as the actual key-value pair item. The frequencies of each key is stored in a linked list data structure. Each of the nodes in the this list have an item list which will contain all of the key-value pairs that have been accessed with the corresponding frequency. Additionally, each of the items in the item list have a pointer to their ancestor in the frequency list (See Figure 1).
Our cache displacement implementations for LFU, LRU, and LFU guarantees *O(1)* run time complexity for all put, get, delete operations.

3.2 Data Replication

In distributed systems, there is a trade off between consistency, availability, and partition-tolerance. The CAP theorem which is classically described as "pick out of three" suggests that in the event of a network failure on a distributed database, it is possible to provide either consistency or availability, but not both of them [8, 17]. Consistency guarantees that all reads receive the most recent write or an error message. Furthermore, availability can only ensure that all reads have data, which may or may not be the most recent data. Partition tolerance ensures that the system continues to operate despite network failures. Since partition tolerance is a must in the theorem [17], that leaves a decision between the other two consistency and availability. In our distributed and replicated key-value store, we have availability over consistency due to our specifications.
Key-value pairs are distributed over a bunch of storage servers through consistent hashing. Storage servers are responsible for their own subset of data and also serve as replicas for key-value pairs of other servers. If the storage server is directly responsible for the key-value pair is called *coordinator*, if the key-value pair is coordinated by another storage node is called *replica* and data is just replicated on this node. Replica nodes can only serve get request for key-value pairs that received from replication process. Replica nodes do not have permission to manipulate replicated key-value pairs. Each data item is replicated by the coordinator server to exactly two following servers. In our implementation replication is only active when at least three servers are up and running. In cases one or two servers are active, no replication is used. Eventual consistency is guaranteed with this replication implementation. The replication can be rebalanced when new servers are added or removed. In our implementation after ECS detects a new server is added, it send a rebalance request to newly added servers' previous server. It sends related key-value pair data to newly added server. Also replicates it's data to this new server since it is the following server. For each key-value pair added, updated or deleted from the storage, replica nodes are updated as well.

3.3 Communication Protocol

3.3.1 Client-Server Communication. As seen in Fig 2, the client commands are forwarded to client library TestStore where they are redirected to the client communication module. SocketCommunicator provides communication with the server. After acquiring a response from the server, SocketCommunicator sends this response to TestClient through TestStore.Client also has connection to the Broker which will be explained in Section 5.

3.3.2 Server-Side Communication. The communication of KVServer-KVServer and KVServer-ECS are represented in Fig 2. Client requests are firstly accepted by NioServer then it is forwarded to KVCommandProcessor. KVCommandProcessor is responsible for parsing client requests and routing those requests to corresponding methods in KVServers. ECS-KVServer communication is managed by KVServerCommunicator which provides a error prone and healthy connection environment without interrupts. Also every KVServer has a connection with Broker Service as well. In cases that notification mechanism is enabled, KVServers sends key-value updates to the Broker Service to be transmitted to subscribers. The details of this service is explained in Section 5.

4 FAILURE DETECTION

Failure detection for database nodes are done through

(1) Heartbeat Signals
(2) Data replication Timeout Mechanism

Data replication timeout failure detection mechanism is implemented on KVServers. Every coordinator server is responsible for replicating the received key-value pair to its two successors for database consistency. In case of coordinator server to successor server connection problems exceeds the 50 second timeout, the successor server is considered to be a failure. A request is sent by the coordinator server to ECS that the successor server cannot be contacted/answered. Then ECS removed this failed server from the system and broadcast metadata information to all remaining servers in the system.
Heartbeat failure detection mechanism is managed by ECS. A ping message send by ECS to all the server nodes in the system in one second frequency. After each ping message, every server node responsible to reply this heart beat message. In case that a server exceeds the timeout limit of 1100 milliseconds, it will be counted as failed and will be removed from the system. After removal process, the updated metadata information will be served to all server nodes in the system by ECS.

5 EXTENSION

5.1 Password Protection

Controlling access is the basis of all security. A closed system with no external interaction may not necessarily need authentication, however in today's world, one needs to accommodate the security needs of users for all types of environments. Our extension to the cloud database we have implemented throughout milestones one to four is password protection.
In the scope of milestone five, we provide the users of our database the option to protect their inputs with a password (See Figure 3). Here the decision of protecting the key-value pairs with a password is left to the user. That is, the user has the option to use the system without the extension. In the case that the user would like to set a password and protect her input, she can do so with the command "handleWithPassword". After this command, the system acts in accordance with the password protection feature and sets the password that user has entered. From this point on, the user can only get, update or delete the key if she provides the same password. Here, it is important to emphasize that the password is key-specific, which means a password can be set for the specified key, not for the user.
The Validation of the password is performed on the server. If the entered password is correct, user gets the corresponding success message from the server. The user gets a warning message from the server if the entered password is not correct. We also provide a wrong attempt counter in order to check the number of times a user has entered a wrong password. In the event of detecting that user has entered a wrong password three times in a row, the system disconnects the user.

5.2 Notification Mechanism

Users need to send get commands to learn current value of a given key. We implemented a notification mechanism to notify the clients about updates on specified keys. This mechanism ensures that clients receive updates about any modifications (update or delete) to clients subscribed keys.
In our implementation process we considered some special cases

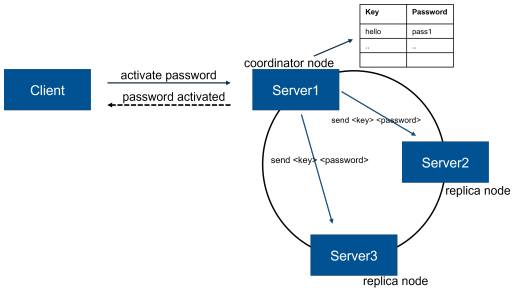

Figure 3: Password Protection Enabling Process

that occurs with the nature of a distributed and replicated system. First special case is that in event of a failure of coordinator node, subscribed keys updates must continue to be served. Second case is when a new coordinator node added to the system for some or all of the subscribed keys, notification mechanism should also continue to work. Third case was how can we determine a client connection since clients can constantly reset their connections to different servers. To satisfy those cases, we decided to implement a Broker Server that is responsible for notification processes. Broker Server is implemented based on Publisher-Subscriber pattern (See Figure 4). It create a separate channel between clients and servers, wherein the servers publish their changes and the clients registers for certain keys. This component provides a server socket and has to be started before KVServers. Broker server provides four capability. The first capability allows clients to subscribe to a key with *subscribe <key>* request. Second capability allows clients to unsubscribe from a key with *unsubscribe <key>* command. Third one is *subscribe_delete <key>* command which is received from servers and immediately transmitted to relevant clients. Last one is *subscribe_update <key> <value>* command that also received from servers and notifies all necessary clients.
The connection between Broker and publishers (KVServers) are managed by KVServerCommunicator which is also very similar to client to server communication logic. The updates for subscribed keys are send to the Broker server through this connection. Then Broker sends these updates to the relevant clients. The communication between subscribers (clients) are managed by Java NioServer SelectionKeys which creates an individual channels for each client. This provided us the capability that client can connect to any key in any of the servers easily. Also client can subscribe for many keys and a key has many different subscribers as well.

6 EVALUATION

6.1 Experiment Setup

We tested our system using a laptop, its configuration details are given below.

Processor	Intel(R) Core(TM) i7-4700HQ
Clock Speed	2.40GHz
Number of Threads	8 Threads
RAM	12 GB

For performance assessment tests Enron Email Dataset [2] is used. This dataset contains data from about 150 users, mostly senior

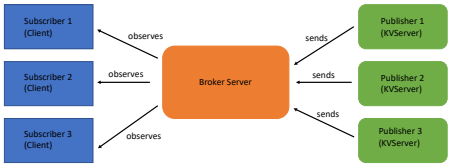

Figure 4: Notification Mechanism

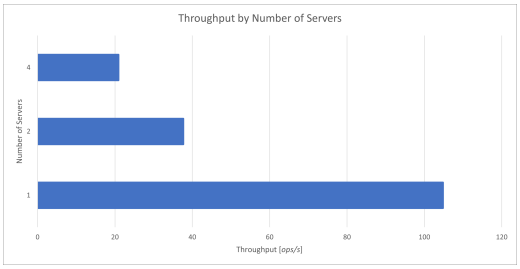

Figure 5: Single Client Multiple Server Throughput

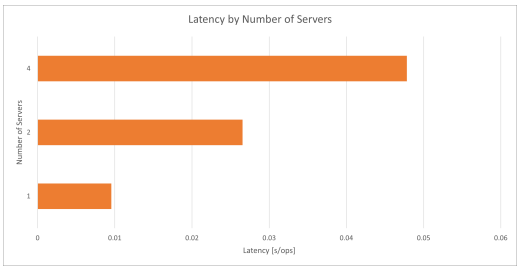

Figure 6: Single Client Multiple Server Latency

management of Enron, organized into folders. The corpus contains a total of about 5000 messages.

6.2 Performance Evaluation

In a distributed system where various servers may be providing services to multiple customers at the same time while propagating a large stream of data, it is critical to assess how the overall system performs when this consistency and its associated performance metric are taken into account. Throughput which is the number of handled requests in a time frame *(operations/completion time)* and Latency which is the time that takes for a request to be handled *(completion time/operations)* metrics are used for the systems performance assessment.

6.2.1 Scaling Number of Servers. For this assessment we used Enron Email Dataset. The total number of handled request number is 127981 for all test cases. We tested throughput and latency metrics of one server, two servers, and four servers from a single client application. For this test assessment throughput is calculated as number of operations divided by completion time in seconds, latency is calculated as completion time in seconds divided by number of operations handled.

Nr. of Clients	Nr. of Servers	Throughput [opr/ms]	Latency [ms/opr]
1	1	0.13	7.51
1	2	0.10	9.62
1	4	0.07	14.54
2	1	0.17	6.03
2	2	0.16	6.32
2	4	0.11	8.85
4	1	0.15	6.78
4	2	0.24	4.16
4	4	0.10	10.38

Table 1: Latency and Throughput for Varying Numbers of Client/Server

The throughput results of put requests based on number of running servers are given in Figure 5. We observe significant throughput decrease when scaling up the servers cluster. In case that scaling from one server to two servers, it has *x2.78* decrease of in throughput. When four servers are running in the cluster, the throughput results are *x1.80* and *x5.01* worse comparing to two server and single server test cases respectively. For latency test assessment results, increasing number of running servers also increases the time frame of responding to client (See Figure 6).

For both test cases are affect adversely by scaling up the system. One reason that causes this result is that the more servers are in the cluster, the more frequently client has to reconnect to a different server when trying to put different key-value pairs. Due to restriction that only coordinator servers are allowed to accept incoming put requests to these key-value pairs.

6.2.2 Scaling Number of Clients and Servers. We used Enron Email Dataset for this performance assessment tests as well. The total number of handled requests are 200, which means total number of 200 key-value pairs are randomly selected from Enron Email Dataset for each test case. Total number of nine test cases performed as seen in Table 1. The competition time which is used for throughput and latency calculations is in unit of milliseconds. According to our performance assessment it can be said that in case that number of connected clients are increasing for constant number of servers, the throughput is increasing and latency is decreasing as seen in Figure 7 and Figure 8 respectively. However scaling up the server nodes in the cluster lowers the throughput and increases the latency results independently from number of clients. In Figure 7, the throughput appears to peak for four clients are connected to two servers. Four clients send an equal number of requests to cluster that consists of two server nodes. Requests are created with randomly retrieved data from the Enron dataset. It is expected that it will have more throughput than a four server cluster system because there is less possibility of server swapping (disconnecting/reconnecting).

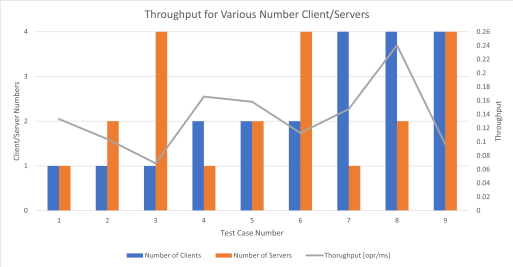

Figure 7: Throughput for Various Numbers of Client/Server

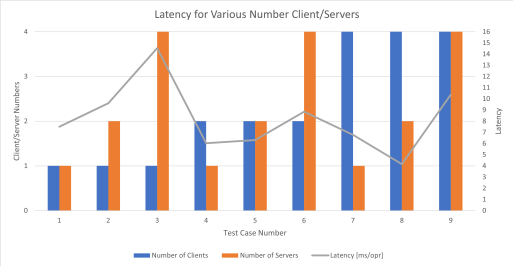

Figure 8: Latency for Various Numbers of Client/Server

7 CONCLUSION

The shift to NoSQL is not coincidental; NoSQL has numerous advantages over traditional relational databases. NoSQL databases are recognized to be more fault-tolerant and horizontally scalable. Due to the lack of a set schema, NoSQL is more flexible and easier to deploy. There are certain drawbacks to adopting NoSQL, such as the fact that each type of NoSQL implementation is unique. The evaluation results represent there is a signifiant performance leap when scaling the from single server to multiple servers. The throughput generally scales more than expected since it has higher change for server swapping. Our implementation provides scale up/down ability for big datasets as well as small datasets. It can satisfy ever growing user requests easily. Also our fast replication process provides users high availability. With password extension for key-value pairs, the security concerns are satisfied for key-value pairs which might contain critical information.

8 FUTURE WORK

Despite the implemented database system being well-established and ready-to-use product, a range of valuable improvements to the system can be defined.

Since ECS service is not included in replication process, it's failure might cause failure of the whole system. ECS is the main service that orchestrates all server nodes and has significant importance for metadata information broadcasting and server replication. As a future plan ECS can be replicated to a slave node which also can overtake the heartbeat signaling process as well to solve this issue. In this way, the load of the ECS is relieved and the service becomes only responsible for server node management processes.

According to evaluation, the latency is increasing due to network latency which is caused by server swapping. As a future plan proxy servers can be implemented for preventing users to wait more time. Proxy servers are responsible for finding the right server to connect and transmit the users request. Also with this implementation database system becomes faster as well as the server side is completely abstracted from the client side. In case of an failure of a server node, proxy nodes can redirect request to the new responsible server node.

REFERENCES

[1] [n. d.]. Cassandra. https://cassandra.apache.org/_/index.html. Accessed: 2022-02-04.
[2] [n. d.]. Enron Email Dataset. https://www.cs.cmu.edu/~enron/. Accessed: 2022-02-05.
[3] [n. d.]. Redis. https://redis.io/. Accessed: 2022-02-04.
[4] [n. d.]. Redis vs DynamoDB. https://severalnines.com/database-blog/redis-vs-dynamodb-comparison. Accessed: 2022-02-04.
[5] [n. d.]. Redis vs Memcached. https://www.imaginarycloud.com/blog/redis-vs-memcached/. Accessed: 2022-02-04.
[6] [n. d.]. What is Memcached? https://medium.com/swlh/what-is-memcached-d1498623db3b. Accessed: 2022-02-04.
[7] [n. d.]. When and Why to use a Least Frequently Used (LFU) cache with an implementation in Golang. https://ieftimov.com/post/when-why-least-frequently-used-cache-implementation-golang/. Accessed: 2022-02-04.
[8] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.
[9] Aditya Auradkar, Chavdar Botev, Shirshanka Das, Dave De Maagd, Alex Feinberg, Phanindra Ganti, Lei Gao, Bhaskar Ghosh, Kishore Gopalakrishna, Brendan Harris, et al. 2012. Data infrastructure at LinkedIn. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1370–1381.
[10] Kristi L Berg, Tom Seymour, Richa Goel, et al. 2013. History of databases. *International Journal of Management & Information Systems (IJMIS)* 17, 1 (2013), 29–36.
[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.
[12] Giuseppe DeCandia, Deniz Hastonrun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. [n. d.]. Dynamo: Amazon's Highly Available Key-value Store [SOSP'07]. ([n. d.]).
[13] Hershel Harris and Bert Nicol. 2013. SQL/DS: IBM's First RDBMS. *IEEE Annals of the History of Computing* 35, 2 (2013), 69–71. https://doi.org/10.1109/MAHC.2013.28
[14] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[15] Dhruv Matani, Ketan Shah, and Anirban Mitra. 2021. An O (1) algorithm for implementing the LFU cache eviction scheme. *arXiv preprint arXiv:2110.11602* (2021).
[16] Robert Preger. 2012. The Oracle Story, Part 1: 1977-1986. *IEEE Annals of the History of Computing* 34, 4 (2012), 51–57. https://doi.org/10.1109/MAHC.2012.54
[17] Salomé Simon. 2000. Brewer's cap theorem. *CS341 Distributed Information Systems, University of Basel (HS2012)* (2000).