ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 8 No. 2, 2023

CLOUD KEY-VALUE STORAGE

Oleh Pykulytskyi, Bohdan Havano

Lviv Polytechnic National University, 12, Bandera Str, Lviv, 79013, Ukraine
Author's e-mail*s: Oleh.Pykulytskyi.mKIKS.2022@lpnu.ua; bohdan.i.havano@lpnu.ua*

https://doi.org/10.23939/acps2023.02.133

Submitted on 06.10.2023

***Abstract*: The paper represents all the stages of designing, architecting, and developing cloud-based key-value storage. This work aims to bring new approaches to distributed data systems. The authors focus on the security and productivity of the project as well as security and maintainability.**

The authors have studied the use of hash tables in a multi-threaded environment. Architectural approaches and tools have been described. The general structure of the key-value storage server has been presented. The server algorithm has been presented. Our research delves into the intricate nuances of utilizing hash tables in a multi-threaded environment, shedding light on the intricacies and challenges of managing concurrent access to key-value data structures. The authors have explored the trade-offs between lock-free designs and traditional locking approaches.

***Index Terms*: cloud storage, multithreading, key-value storage, lock-free.**

I. INTRODUCTION

Nowadays, the vast majority of software solutions use some cloud storage. They are used for educational purposes, and store business analytics. This technology has become widespread in medicine. For example, it is used for storing biometric data from the human state remote monitoring tools 0, big data [2], Internet of Things [3]. It is evident that with the development and spread of technologies, more and more types of cloud storage offer their approach to interacting with them. This variety allows for choosing the cloud storage that best solves problems.

Cloud computing is a way for businesses to use technology without having to buy it themselves. Instead, they rent the technology from a company that already has it. This saves the business money because they don't have to spend a lot of money upfront. It also helps them save money on operating costs because they only pay for what they need. Cloud computing also makes it easy for businesses to grow and handle more customers because they can easily get more technology when they need it [4]. And finally, using cloud computing is convenient because businesses can access their technology from different devices like computers and phones. It also helps businesses because they don't have to worry about things going wrong with their technology. After all, the company they rent from takes care of that [5].

Local machines are becoming less and less adaptable, necessitating more regular updates to the hardware and software components. Program architecture deteriorates in flexibility with time and gets more complex. At the same time there are cloud platforms that enable resource optimization, guarantee scalability, and provide flexibility to address this issue [6].

As the number of internet users continues to rise annually, the costs associated with processing and storing user data have also increased. New tasks that require high-performance storage, capable of reading and writing data quickly, have also emerged. Relational databases are currently the most widely used type of storage, consisting of related data organized into tables with relationships established between them.

However, in many cases, relational databases may be excessive, and their relatively slow speed is a hindrance. Key-value storage provides a practical solution in such scenarios. By storing data in memory instead of on disk, key-value storage significantly reduces latency and response time per request, as RAM speed can be up to 50 times faster than that of hard disks. Databases also use concurrency to utilize multiple CPUs, by assigning transactions to different threads.

This type of storage is particularly effective for storing large volumes of unrelated small-sized data, which can be accessed, modified, or deleted quickly. Applications of key-value storage include caching, acceleration of application response, user session data storage, authorization key storage, and high-performance operation data storage, among others.

II. REVIEW OF RECENT PUBLICATIONS

Key-value data stores have been a topic of research for many years. This type of software system has become a fundamental component of modern data center infrastructure. During this time, many proposed solutions were oriented toward different parts of the system. Some researchers are focused on improving algorithms for persistent stores, while others propose different approaches to improve the performance or safety of key-value databases.

Intel Software Guard Extensions (SGX) [7] were proposed as a way to improve the safety of the in-memory key-value stores. SGX can provide secure cloud computing on remote systems under untrusted privileged system software. The main advantage of SGX is that it provides isolated execution environments protected from malicious operating systems and physical attacks, which can significantly improve the safety of the systems. However, as the paper shows, this approach can restrict the performance of the store.

This paper [8] proposes a design based on persistent memory (PM). The key idea is to decouple the role of a KV store into a volatile index for fast indexing and a persistent log structure for efficient storage. The authors proposed a compact log format and pipelined horizontal batching techniques to achieve high throughput, low latency, and multi-core scalability.

In this paper [9] authors try to achieve high single-node throughput by improving several parts of the system, like network I/O, parallel data access, and the overall design of key-value data structures. The authors emphasize the importance of high-speed nodes in an in-memory database since they may reduce costs by requiring fewer of them overall, reducing the cost and overhead of replication and invalidation.

As it can be observed from the review, key-value stores experience strong interest from researchers. Overall, the dynamic landscape of key-value data stores continues to attract a wealth of interest and exploration from the research community. These endeavors span a spectrum of objectives, ranging from fortifying security measures to advancing the performance and scalability of key-value databases through inventive design strategies, as exemplified in this section.

Furthermore, the emphasis on improving network I/O, parallel data access, and the overall design of key-value data structures in pursuit of high single-node throughput highlights the pragmatic importance of optimizing key-value data stores. This not only enhances performance but also has potential cost-saving implications by reducing the need for extensive replication and invalidation processes.

III. OBJECTIVES

The main objective of this paper is to develop a high-performance in-memory key-value database by embracing lock-free approaches to the storage process. In-memory storage allows for the improvement of the performance of many types of software systems. Since this type of software is very latency-sensitive, it is very important to provide fast access to the data.

While current solutions use locks internally, with the development of new lock-free algorithms and approaches, it is crucial to utilize them in data storage. To accomplish this goal, we have to quantify and evaluate the performance gains achieved by lock-free techniques in key-value stores, particularly in terms of throughput, latency, and scalability. Compare these results against traditional lock-based approaches to highlight the advantages and trade-offs. The resulting system should be capable of performing more than 300,000 operations per second while efficiently utilizing hardware resources, with a particular emphasis on achieving these benchmarks on a 16-core CPU. Additionally, the target latency should remain below 1 millisecond under normal operating conditions, ensuring that this high level of performance is consistently achievable across a multi-core processing environment.

Another important task is to investigate how lock-free mechanisms impact the concurrent access and scalability of key-value stores, with a focus on scenarios involving high contention and varying workloads. We need to conduct a comparative study between lock-free key-value stores and their lock-based counterparts, offering a holistic view of the benefits and trade-offs in different application contexts.

By addressing these objectives, this research paper aims to contribute valuable insights into the realm of lock-free techniques in key-value stores, providing a foundation for further advancements in the field and aiding practitioners in designing high-performance, scalable, and concurrent data storage systems. Implementing high-performance storage should allow companies to reduce their expenses on server infrastructure as well as increase the throughput of client traffic.

IV. DISTRIBUTED DATA STORAGE

To store data in memory, it is necessary to consider data types that allow maximum efficiency in working with them. For these needs, a hash table is perfect. A hash table is one of the implementations of an associative array. An associative array is a data type that stores a collection of pairs (e.g., key, value). Each key can appear only once in the collection, and the array supports operations like looking up a value and removing items from the array. Associative arrays have two essential properties. Only one key can have a given value, and every key can only appear once in the list.

A hash table usually uses a hash function to calculate an index, which is then used to look up a value in an array of slots. If the desired value is not found in the first slot in the collection, the hash table will look in the second slot, and so on. If the expected value is still not found, the hash table will check the third slot, and so on.

A hash table is a type of storage that uses buckets to divide the data into smaller groups. When looking up an item in a hash table, the system calculates its hash (a number representing its unique identity). Then it looks through the buckets, checking which bucket the hash belongs to. If the hash belongs to a bucket, the system grabs the item from that bucket and moves on. If the hash doesn't belong to a bucket, the system looks for an open bucket with room for the item. The linked list is then scanned to see if the desired element is present. If the linked list is short, this scan is quick. If the linked list is not short, the key is hashed again to find the correct bucket for the linked list. Then, the bucket is checked to see if the item is there, and finally, the desired element is added or removed from the bucket in the usual way for linked lists.

A. DISTRIBUTED HASH TABLES

Distributed Hash Tables (DHTs) are a distributed storage technology that allows data to be stored and retrieved across a network of nodes. DHTs use a hash function to assign data to nodes in the network, allowing for efficient storage and retrieval of data. DHTs have been used in various applications, including peer-to-peer networks, cloud storage, and distributed databases. This article will explore the design, research, and applications of distributed hash tables.

The design of distributed hash tables involves several vital components. These include the hash function used to assign data to nodes, the routing protocol used to find nodes in the network, and the data replication strategy used to ensure data availability. The hash function used in a DHT should be fast, collision-resistant, and evenly distribute data across the nodes in the network. The routing protocol used in a DHT should allow for efficient data routing between nodes, even in the face of node failures and network congestion. Finally, the data replication strategy used in a DHT should ensure that data is available despite node failures and network partitions.

Distributed hash tables have been used in various applications, including peer-to-peer networks, cloud storage, and distributed databases. In cloud storage, DHTs are used to store and retrieve data across multiple nodes in a distributed database system. When a user uploads data to the cloud, the DHT algorithm hashes the data's identifier to locate the appropriate node(s) where it will be stored. Subsequently, when a user requests access to the data, the DHT swiftly retrieves it by following the same hashing process, making cloud storage systems highly scalable and fault-tolerant. This not only enhances data access speed but also increases data redundancy, further safeguarding against data loss or system failures.

B. CONSISTENT HASHING

Consistent hashing (Fig. 1) is a shared hash table technique that allows data to be evenly distributed across shards while minimizing the number of data items that need to be moved when the number of shards changes.

The theory behind consistent hashing is based on the idea of a ring. Each node in the network is also mapped to a point on the ring. Data items are then stored on the node whose point on the ring is the closest to the point on the ring that corresponds to the hash value of the data item. This allows for the efficient distribution of data across the network, fault tolerance, and load balancing.

The implementation of consistent hashing involves several key components. These include the hash function used to map data items to points on the ring, the number of virtual nodes representing each physical node on the ring, and the mechanism used to handle node failures and additions. The hash function used in consistent hashing should be fast and produce a uniform distribution of hash values. The number of virtual nodes used to represent each physical node on the ring can affect the system's load balancing and fault tolerance properties. A more significant number of virtual nodes per physical node can lead to better load balancing and fault tolerance but also increase the system's complexity. The mechanism used to handle node failures and additions can involve the use of replicas or the migration of data to other nodes in the network.

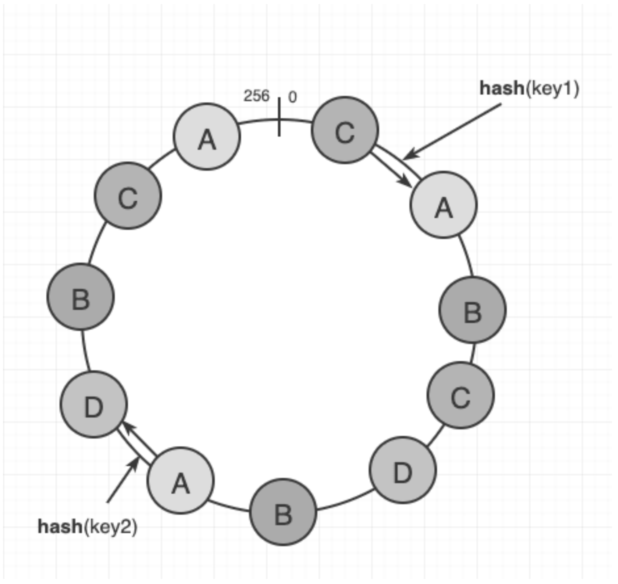

Fig. 1. Consistent hashing

One of the key benefits of using consistent hashing in these applications is its ability to provide fault tolerance and load balancing. Because data is distributed across multiple network nodes, consistent hashing can continue operating even in the face of node failures or network partitions. In addition, consistent hashing can handle large amounts of data and traffic, making it a highly scalable solution for distributed storage and processing.

V. SYNCHRONIZATION AND THREAD SAFETY

To handle multiple connections efficiently, it is necessary to distribute the load between connections and provide them with shared access to data.

Developing a high-performance, secure network application requires efficient use of threads. This task becomes one of the most critical and complicated when it comes to data storage, especially in-memory storage. To deal with shared data between threads, we must emphasize the importance of the synchronization primitive.

Synchronization primitives are mechanisms usually provided by an operating system to support thread or process synchronization. Many programming languages implement their API to interact with these mechanisms.

They can be divided into two large groups: those that use shared memory and those that allow sending information from one thread to another. Those that provide access to shared memory include mutex, semaphore, RWlock, condition variable, barrier, etc.

Mutex (from mutual exclusion) is a synchronization primitive that prevents the usage of shared data at the same time by two or more threads (execution processes). Mutex usually holds shared data inside it, known as a critical section. The mutual exclusion algorithm ensures that if a process is already performing a write operation on a data object (critical section). No other thread is allowed to write or read the same object until the first process has finished writing upon the data object critical section and released the object for other processes to read and write upon.

Any thread that successfully locks the mutex becomes the owner of the mutex until it unlocks the mutex. Every thread that attempts to write to the shared data or read this data has to wait until the owner unlocks the mutex.

Readers-writer lock is quite similar to a mutex, but unlike it, it allows multiple read operations simultaneously. Just like mutex, the RWlock is built on top of semaphore. This fact makes it possible to theoretically speed up access to shared data, given that replacing a mutex with RWlock is "seamless."

Synchronization primitives can have a significant impact on the performance of software applications. On the one hand, synchronization is necessary to ensure correctness and safety. However, on the other hand, excessive synchronization can lead to decreased performance as threads wait for access to shared resources.

One of the primary performance considerations with synchronization primitives is contention. Contention occurs when multiple threads attempt to access a shared resource simultaneously and must wait for access. This can lead to decreased performance, as threads wait for access to the resource rather than doing valuable work.

This becomes especially noticeable during intensive writing to the table, where the difference with a distributed table is up to 10 times. Distributed and lock-free hash tables demonstrate similar performance, but the distributed hash table provides much more security because it uses locks to access internal tables.

It is a reasonable choice to use lock-based data structures in high-throughput required systems. However, we need to mention that this approach creates a noticeable overhead when accessing the data structure. Usually, locks like mutex use a system call to block the execution on a thread that wants to acquire the lock while it is locked by a different thread. If this overhead is critical for the system, it is reasonable to consider a lock-free approach. Highly concurrent access to shared data demands a sophisticated 'fine-grained' locking strategy (Fig. 2) to avoid serializing non-conflicting operations. Such strategies are hard to design correctly and with good performance because they can harbor problems such as deadlock, priority inversion, and convoying. Lock manipulations may also degrade the performance of cache-coherent multiprocessor systems by causing coherency conflicts and increased interconnect traffic, even when the lock protects read-only data.

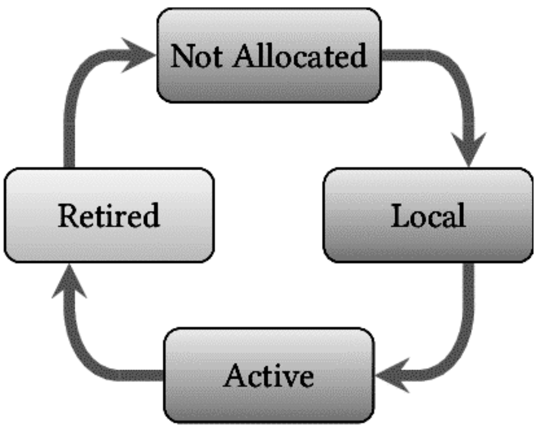

Fig. 2. The lifecycle of a record

A data structure is "lock-free" if and only if an operation is completed after a certain number of steps have been executed system-wide on the data structure. This guarantee of "system-wide" progress is usually achieved by having a process experience contention so that it can wait for the competing operation to finish before continuing with its work. This guarantees that every executing process always ensures the forward progress of an operation. This is very different from a lock-based algorithm, where a process spins or blocks until the competing operation is completed.

The biggest problem in designing lock-free data structures (and especially collections) is to deal with cases when one thread currently reads the data, and another thread tries to rewrite this data. In this case, when the writer rewrites the given data, it gets back the old data, which it needs to decide what to do with. A writer cannot easily deallocate it, since other threads may rely on this data.

This task is usually solved by embracing the memory reclamation mechanism.

The most straightforward is reference counting (RC). This approach allows the safe reclamation of pointers when there are no active references to them. It is a technique used to automatically determine when a piece of memory (such as an object or data structure) is no longer in use so that it can be safely deallocated. Each pointer that needs to be reference-counted has a counter (usually an atomic unsigned integer) associated with it. Each time the thread wants to access this pointer, it increments the counter by one. When a pointer is not needed (when data goes out of scope), a thread decrements this counter by one. When the counter reaches 0, data can be finally deallocated. Even though these operations require only one additional operation (increment or decrement) (and on some architectures, even one instruction), this method is not very efficient because memory management decisions are made on a per-reference basis. This means that memory may be reclaimed and objects may be deallocated more frequently than in other garbage collection strategies, leading to higher memory management overhead. In addition, each decrement operation requires a compare-and-swap (CAS) operation to prevent data races and double-free problems.

A hazard pointer (HP) is a single-writer multi-reader pointer that can be owned by at most one thread at any time. Only the owner of the hazard pointer can set its value, while any number of threads may read its value. A thread that is about to access dynamic objects optimistically acquires ownership of a set of hazard pointers to protect such objects from being reclaimed. The owner thread sets the value of a hazard pointer to point to an object to indicate to concurrent threads — that may remove such object — that the object is not yet safe to reclaim.

They are used to inform the other threads of what data structures are being accessed by a thread and that thus cannot be reclaimed by others. Hazard pointers usually imply a significant overhead caused by threads having to share their location every time they move on the data structure.

Epoch-based reclamation (EBR) takes a different approach. The scheme builds on limbo lists which contain a retired object until no stale references can exist. It uses a global epoch counter to determine when no stale references exist to any object in a limbo list. Each time a process starts an operation in which it will access shared memory objects, it observes the current epoch. When all processes have observed the current epoch, the limbo list that was populated two epochs ago can safely be reclaimed. This now-empty list can be immediately recycled and populated with retired nodes in the new epoch.

When a thread wants to operate on the data structure, it first sets its "active" flag and then updates its local epoch to match the global one. If the thread removes a node from the data structure, it adds that node to the retired list for the current global epoch. When it completes its operation, it clears the “active” flag. Each thread puts the current epoch E in a reservation at the beginning of operations, reserving all objects retired on and after epoch E.

EBR is much more efficient than HPs because it only introduces a small amount of overhead at the beginning of each operation. On the other hand, HPs require costly synchronization every time a new record is accessed. However, the drawback of EBR is that processes have limited knowledge about which records might be accessed by a potentially crashed process, as writing to memory only occurs at the start of each operation instead of every time a new record is accessed.

When multiple threads or processes access a hash table simultaneously, synchronization is necessary to prevent race conditions and ensure consistency. Different synchronization primitives can be used to protect hash tables from concurrent modifications, including locks, semaphores, and mutexes. In the context of hash tables, different synchronization primitives have varying impacts on performance.

To take measurements, we need to perform the following steps:

1) Generate the mixture of operations we will be running, which consists of a certain percentage of reads, inserts, erases, updates, and upserts. We pre-generate the mixture to avoid computing which operation to run while timing the workload.

2) Pre-generate all keys we will be inserting into the table. We can calculate an upper bound on the number of keys inserted based on the prefill percentage and the number of inserts and upserts we will perform. Again, pre-generating the keys avoids doing it while timing the workload. We do not need to pre-generate the values since the values are the same for all operations.

3) Initialize the table.

4) Fill up the table in advance to the specified prefill percentage.

5) Run the pre-generated mixture of operations and time how long it takes to complete all of them.

6) Report the details of the benchmark configuration and the quantities measured, including time elapsed, throughput, and (optionally) memory usage samples.

To decide what type of synchronization primitive to use, we conducted a series of benchmarks to test the performance of different synchronization primitives paired with different hash tables.

Three measures were conducted for a set of hash table + synchronization primitive with other loads to simulate real situations (Fig. 3, Fig. 4). Measurements were carried out in the Rust programming language and the bustle measurement library. Bustle runs four tests for each hash table on a different number of threads to measure performance in different environments.

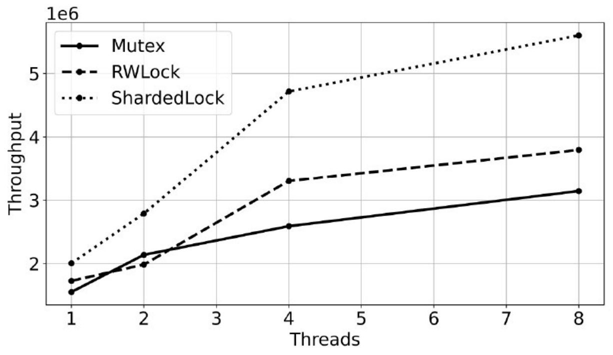

Fig. 3. Throughput for different lock types

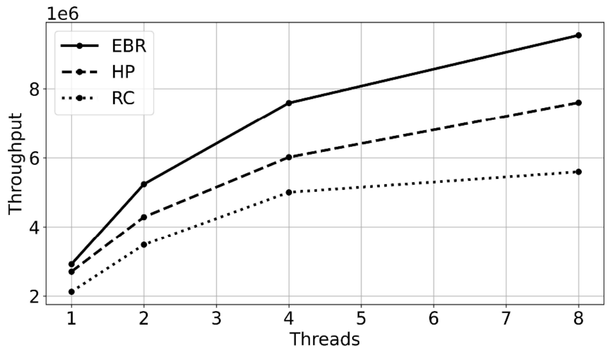

Fig. 4. Throughput for different memory reclamation strategies

The above benchmarks show that all used tables show similar performance in a single-threaded environment. However, when we talk about multithreading, mutexes, and RWlocks in combination with a standard hash table becomes very inefficient.

This becomes especially noticeable during intensive writing to the table, where the difference with a distributed table is up to 10 times. Distributed and lock-free hash tables demonstrate similar performance, but the distributed hash table provides much more security because it uses locks to access internal tables.

VI. SERVER ARCHITECTURE

The server's general structure (Fig. 5) represents each spawned server instance in the cloud. Every server instance can handle multiple clients simultaneously while efficiently processing them. Since each server is multi-threaded, there are no performance vulnerabilities. All incoming connections are held in the program's main thread asynchronously. This fact allows not to block the execution of the program on each connection, which significantly reduces the average time of establishing a connection.

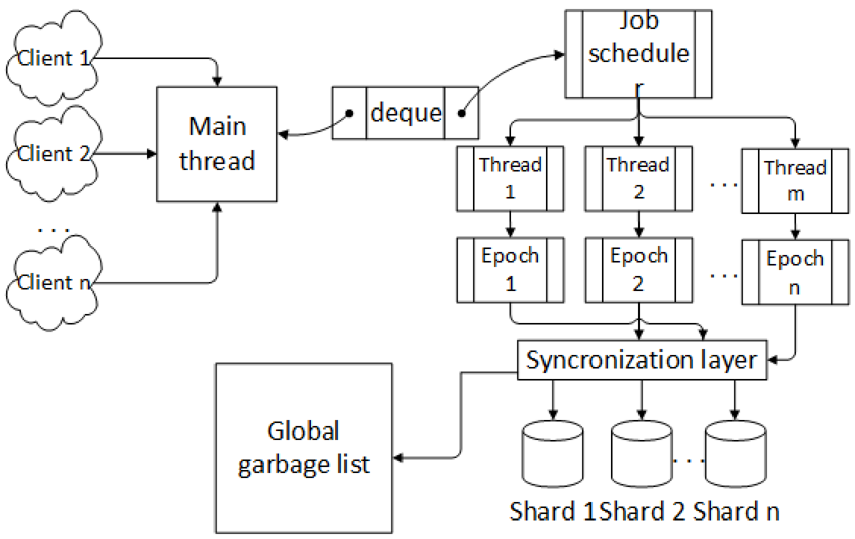

Fig. 5. Server structure

After establishing the connection, the server starts receiving incoming frames. Each received command represents one or more available commands. After receiving, each frame is put into the deque.

A double-ended queue (deque) is a data type whose elements can be added or removed at the beginning and end. It allows us to put data in the main thread and receive them in one of the threads in the pool.

Deque supports the following operations:

- Adding an element to the end of the queue.
- Adding an element to the beginning of the queue.
- Selection of the last element.
- Selection of the first element.
- Checking the first element (without removing it from the deque).
- Checking the last element (without removing it from the deque).

A deque can be implemented using an array or a linked list. When using an array, two indices are maintained to keep track of the front and back of the deque. When using a linked list, the deque is implemented as a doubly linked list, with pointers to both the front and back nodes.

Deque implementations can vary depending on the specific use case. For example, when implementing a deque in a user interface, the deque might be implemented as a circular buffer to allow for efficient scrolling through a list of items. In our case, the deque might be implemented using a linked list for better memory efficiency.

In addition to the essential insertion and deletion operations at both ends, deques can support other operations, such as peeking at the front or back element, checking if the deque is empty, and iterating over the elements in the deque. These operations can also be implemented efficiently, typically in O(1) time.

Insertion and deletion at the front and back of the deque can be performed in constant time, O(1) when using an array or a linked list. This is because the position of the front and back of the deque is always known, and inserting or deleting an element at either end requires updating one or two pointers.

Each server instance has a set of threads in a thread pool. When the thread completes another task (processing the client request), it turns to deque. From the end, it reads the last task and removes it from the deque. On the other side main thread listens to all the connections and adds the incoming commands to the front of the deque.

This approach allows the server to process incoming requests in the order in which they come. Thread pools can be faster than creating new threads for each task because the overhead of thread creation and destruction is limited to the initial pool creation. If there are too many threads in reserve, this can waste memory and slow down performance. Associating tasks with threads that live over multiple network transactions can keep them more efficient.

In addition, this approach makes it possible to dynamically add and remove threads to optimize the cloud server for the current load and number of simultaneous connections. A separate thread is not tied to the task that comes to it for execution, which allows us to change the process of processing a client request in the future without changing the logic of the thread pool. When the client is disconnected main thread can remove all the tasks related to this connection from the queue.

VII. SERVER ALGORITHM

The server's algorithm (Fig. 6) describes the process of handling each connection. As it was mentioned earlier, the client communicates with the cloud server through frames. A frame represents the intermediate layer between the command which is handled by the server, and raw bytes, which are sent via TCP. A frame can contain either one command or several using a "bulk" parameter. On a transport layer, when a TCP frame is sent from the sending host, it is first encapsulated in an IP packet containing the IP addresses of the sending and receiving hosts. The IP packet is then sent to the receiving host over the network.

When the receiving host receives the IP packet, it first checks the destination IP address to determine if it is intended for it. If the packet is intended for it, the receiving host extracts and processes the TCP segment from the packet. The receiving host uses the information in the TCP header, such as the sequence number and acknowledgment number, to ensure that the segments are delivered reliably and in the correct order.

If a segment is lost or corrupted during transmission, the receiving host sends a message to the sending host requesting that the missing or corrupted segment be retransmitted. This process ensures that the data is delivered reliably and in the correct order, even in network errors or congestion.

Each frame consists of data and a data type identifier. The following table shows the supported types and their corresponding identifiers. The type identifier comes first in the message and occupies one byte.

Table 1

Supported commands

Command name	Command description
GET	finds a key in storage and returns it back
SET	insert value in the storage, or update if an entry exists
AUTH	authorize current connection
INCR	increments the counter by 1
DECR	decrements the counter by 1
EXISTS	returns true if the key exists in the storage
CLOSE	close the current connection, purge all the keys
DEL	delete the key-value pair from the storage

To represent a single command, a client can use a bulk string by putting a command name, key, value, and other fields individually. For example, to get the key's value, the client sends the following message: "0x5\r\nGET\r\nkey\r\n". All available commands supported by the server are listed in the following table. A client can combine several commands in a single frame. For this, the client needs to use an array, each element of which is a separate command. All supported commands are listed in Table 1.

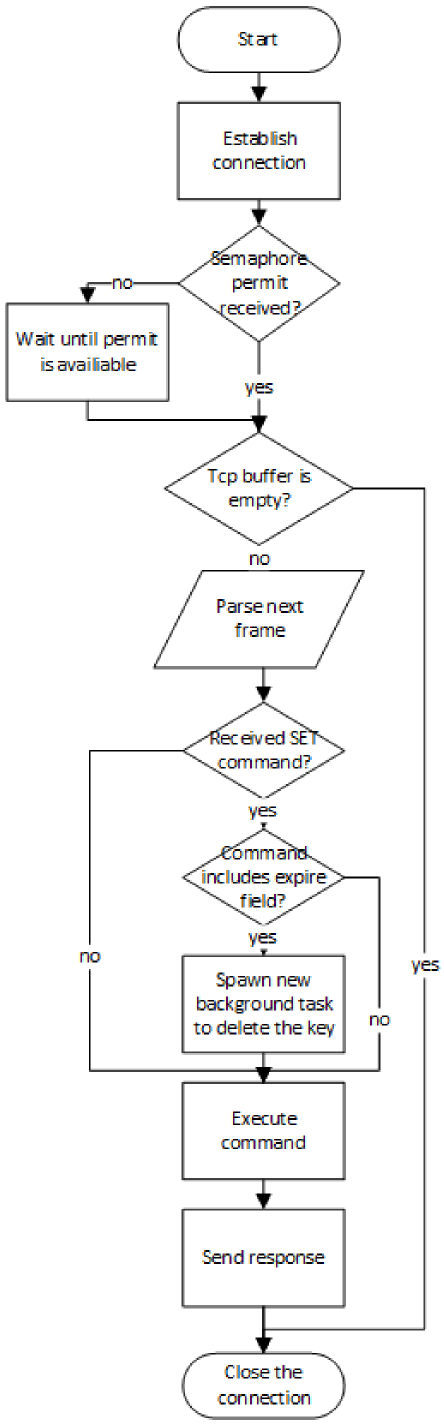

Fig. 6. Algorithm for handling connection

VIII. ANALYSIS OF RESULTS

Latency is indeed a critical metric in data storage systems. It represents the time delay between a request for data and the moment that data is delivered or retrieved. Low latency is often desirable in various applications, such as databases, cloud computing, and real-time data processing, where timely access to information is crucial.

The most pivotal and important metric in data storage is latency. There are diverse factors that impact latency and optimizing any of these factors seems to be effective.

The arrival rate of keys is a parameter that describes how an unbalanced load affects the performance and in particular latency of the system. In this experiment, the arrival of keys was concurrent and was increasing by a constant number of keys each time per unit (1 sec). This allows us to test the system under different, gently increasing workloads and draw conclusions about average latency for a single key.

This test was conducted only for valid keys, deliberately omitting missed keys, since missed keys usually are handled much faster, which could affect the validity of the test. By focusing solely on valid keys, we ensure that the experiment accurately reflects the system's ability to handle the ordinary, expected workload without the influence of exceptional cases.

As illustrated in Fig. 7, the proposed system can show a latency of less than 1 msec for the arrival rate of keys up to 90. From the Fig. 7, we can notice that latency increases gently when Kps is less than 30.

This suggests that the system is not only optimized for high loads but also maintains reasonable latency even when the workload is lighter. This versatility in latency performance is valuable because not all workloads are consistently high, and some applications may require low latency even during periods of lower data demand.

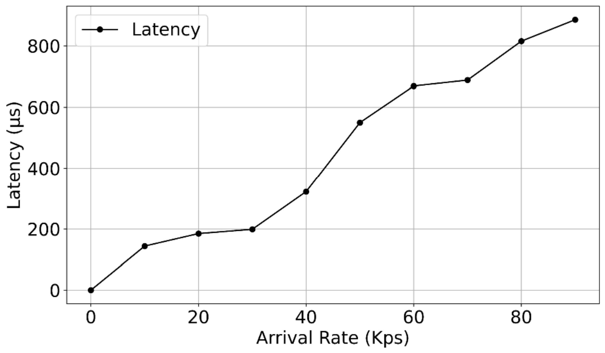

Fig. 7. Evolution of latency when the average arrival rate of keys varies from 0 Kps to 90 Kps

Another metric we should consider is throughput. To measure the throughput of the system, we have run the server with different numbers of connections, to measure how high-parallel workloads affect the performance. The results of this test are presented in Table 2. From this data, it's evident that the system's throughput is influenced by the number of connections and threads. This information is valuable for understanding the system's limitations and for capacity planning in real-world deployments.

As we can see from Table 2, the system is capable of handling up to 322580 operations/sec. It is noticeable that when the number of connections increases by 10 times, throughput drops only by less than 2 times, which indicates the high level of sustainability of the system.

Table 2

Throughput benchmark

Number of connections	Number of threads	Throughput (operations/sec)
1	16	322580
10	16	187312
20	16	168771
30	16	99915

IX. CONCLUSION

Cloud storage key-value offered a powerful and flexible solution for storing and accessing data in modern computing environments. Cloud key-value stores were based on a distributed architecture that enabled data to be stored and accessed across multiple servers. This architecture allowed for the creation of highly available and fault-tolerant systems that could handle large volumes of data and support complex data structures.

The advantages of using key-value stored in the cloud extend to scalability and performance, allowing them to seamlessly adapt to dynamic workloads and concurrent requests. As the foundation of modern distributed systems, their design, research, and development were paramount to realizing the full potential of cloud computing.

The use of key-value stores in the cloud also offered significant benefits in terms of scalability and performance. By distributing data across multiple servers, these systems can handle many simultaneous requests and scale dynamically to accommodate changing workloads.

In summary, the design, research, and development of cloud key-value stores were critical topics for modern distributed systems. These systems offered significant benefits in terms of scalability, performance, and reliability, but they required careful consideration of the design and implementation to ensure that they were robust and secure.

By leveraging the appropriate synchronization primitives, hashing algorithms, and security approaches, developers can build highly available and fault-tolerant systems that meet the demands of modern distributed applications in the cloud. The system's throughput was measured under various conditions, including different numbers of connections and threads.

It exhibited strong performance with a high throughput of up to 322580 operations/sec. As the number of connections increased, the throughput remained substantial, showcasing the system's ability to scale and efficiently process requests. The system demonstrated low latency, with values consistently below 1 msec for arrival rates of keys up to 90. This indicated that the system was capable of responding quickly to data retrieval requests, even under moderate to high loads.

REFERENCES

[1] A. Melnyk, Y. Morozov, B. Havano and P. Hupalo, (2021). Protection of Biometric Data Transmission and Storage in the Human State Remote Monitoring Tools. *11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Cracow, Poland, vol. 1, pp. 301–306, DOI: 10.1109/IDAACS53288.2021.9661047

[2] Mazumdar, S., Seybold, D., Kritikos, K., & Verginadis, G., (2019). A survey on data storage and placement methodologies for Cloud-Big Data ecosystem. *Journal of Big Data*, vol. 6, pp. 1–37. DOI: https://doi.org/10.1186/s40537-019-0178-3

[3] Awan, I., Younas, M., & Benbernou, S., (2021). Convergence of cloud, Internet of Things, and big data: new platforms and applications. *Concurrency and Computation: Practice and Experience*, vol. 33, no. 23, pp. 1–3. DOI: 10.1002/cpe.6668

[4] Sadeeq, M.A., Abdulkareem, N.M., Zeebaree, S.R., Ahmed, D.M., Sami, A.S., & Zebari, R.R., (2021). IoT and Cloud Computing Issues, Challenges and Opportunities: A Review. *Qubahan Academic Journal*, vol. 1, no. 2, pp. 1–7. DOI: https://doi.org/10.48161/qaj.v1n2a36

[5] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa Zarlenga, M., Lin, R., Liu, Z., Padilla, J., & Delimitrou, C., (2019). An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18. DOI: https://doi.org/10.1145/3297858.3304013

[6] Vladyslav Kotyk, Yevhenii Vavruk, (2022). Comparative Analysis of Server and Serverless Cloud Computing Platforms, *Advances in Cyber-Physical Systems*, vol. 7, no. 2, pp. 115–120. DOI: https://doi.org/10.23939/acps2022. 02.115

[7] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh, (2019). ShieldStore: Shielded In-memory Key-value Storage with SGX. *Proceedings of the Fourteenth EuroSys Conference 2019*, March 25–28, 2019, Dresden, Germany. ACM, New York, NY, USA, pp. 1–15. DOI: https://doi.org/10.1145/3302424.3303951

[8] Chen, Y., Lu, Y., Yang, F., Wang, Q., Wang, Y., & Shu, J., (2020). FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1077–1091. DOI: https://doi.org/10.1145/3373376.3378515

[9] Lim, H., Han, D., Andersen, D.G., & Kaminsky, M., (2014). MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, vol. 39, no. 4, pp. 429–444. DOI: 10.5555/2616448.2616488

Oleh Pykulytsky received the B.S. degree in Computer Engineering at Lviv Polytechnic National University in 2022. Now he is a fifth-year computer engineering student at Lviv Polytechnic National University. His research interests include networking, architecture, system design, concurrency and back-end development.

Bohdan Havano received the B.S. degree in Computer Engineering at Lviv Polytechnic National University in 2015 and M.S. degree in system programming at Lviv Polytechnic National University in 2016. He has been doing scientific research work since 2017. His research interests include architecture and data protection in cyber-physical systems.