Advanced DataBases | INFO-H415

Benchmarking Key-Value Stores - A Comparative Analysis of Redis and etcd

Mayr Dionisius

Suau Thomas

Elezi Herma

İşlek Rana

Prof. Esteban Zimányi

2023

Contents

1 Introduction

1 Background

The choice of the right database management system technology is crucial for the success of any application. Key-value store databases have gained prominence for their efficiency in storing and retrieving objects in a simple way, they are nowadays present in the most diverse scenarios.

2 Motivation and Objective of the Project

The goal of our project is to simulate a design decision: which Database Management System (DBMS) should be used for a certain application. To do so, we will be comparing two different technologies: traditional relational databases and key-value stores. Our objective is to prove that our chosen technology is the best for this application. Additionally, we will provide arguments for the design decisions as well as a performance benchmark of the three database systems being compared.

3 Scope and Limitations

The scope of our project is centered around benchmarking two key-value store databases, namely Redis and `etcd`, for their suitability in real life scenarios. The evaluation will consider factors such as performance, scalability, and reliability.

Limitations of our project include the focus on only two key-value store databases, which may not cover the entire spectrum of available options. Additionally, the benchmarks will be specific to the chosen use case of a cache application and may not be directly applicable to other types of applications. We decided to implement the cache application only for the one database which has the highest performance as a result of our benchmarking, to observe in real life application.

Despite these limitations, our project aims to provide valuable insights into the strengths and weaknesses of key-value store databases, assisting developers in making informed decisions for their applications.

2 Key-Value Stores

A key-value store is a type of database management system that stores data as a collection of key-value pairs. In this model, each data item is associated with a unique key, and the system retrieves or stores data based on these keys. Key-value stores are characterized by their simplicity, high performance, and scalability.

1 Definition and Characteristics

Key-value stores operate on a simple principle: a unique key corresponds to a specific value, forming a key-value pair. These systems are schema-less, allowing flexibility in the types of data they can store. Characteristics of key-value stores include high-speed data retrieval, horizontal scalability, and minimalist data models.

2 Importance in Modern Database Systems

Key-value stores play a crucial role in modern database systems due to their ability to handle large volumes of data with high throughput. They are particularly suitable for scenarios where quick and efficient data access is essential, such as caching, session storage, and real-time analytics. The simplicity of the key-value model also aligns well with the requirements of many modern applications where it is necessary to store and retrieve generic objects.

3 Advantages and Use Cases

3.1 Advantages

Key-value stores are known to offer several advantages, including:

- **High Performance:** Key-value stores excel in terms of read and write operations, making them ideal for applications requiring low-latency data access;

- **Scalability:** These systems can easily scale horizontally by adding more nodes, allowing them to handle increasing amounts of data and traffic;
- **Flexibility:** The schema-less nature of key-value stores enables the storage of diverse data types without the need for predefined structures.

3.2 Use Cases

Key-value stores find application in various use cases, such as:

- **Caching:** Key-value stores are commonly used for caching frequently accessed data to improve application performance and even save costs;
- **Session Storage:** Storing session data in a key-value store ensures quick retrieval and efficient management of user sessions;
- **Autocomplete:** The fast read and write capabilities, together with the data structures provided make Redis suitable for auto-completing applications (such as recent contacts).

3 Redis

1 Overview

Redis, short for Remote Dictionary Server, is an open-source, in-memory data structure store. It provides a key-value store with support for various data structures such as strings, hashes, lists, sets, and more. Redis is designed for high performance, with data residing entirely in memory, making it suitable for use cases that require low-latency access.

Redis is a versatile database that serves various purposes, including acting as a database, cache, streaming engine, message broker, and more. It is known for its completeness, focusing on fast writing and accessibility [Red23].

2 Architecture

The architecture of Redis is centered around an in-memory data store. It follows a client-server model, where clients can connect to the server and perform various operations. Redis supports replication for data durability and high availability. The system is single-threaded, but its event-driven architecture allows it to handle a large number of concurrent connections.

3 Features

Redis comes with a rich set of features, including:

- **Data Structures:** Redis supports a variety of data structures such as strings, hashes, lists, sets, providing flexibility in data modeling;
- **Persistence:** While bein primarily an in-memory store, Redis offers options for persistence, allowing data to be saved to disk for durability;
- **Replication:** Redis supports master-slave replication, enabling data redundancy and high availability;

- **Partitioning:** Redis can be horizontally scaled through sharding, distributing data across multiple nodes.

4 Use Cases

Redis finds application in various scenarios, including:

- **Caching:** Due to its in-memory nature, Redis is widely used as a caching layer to store frequently accessed data for quick retrieval;
- **Session Store:** Storing session data in Redis ensures fast access and efficient management of user sessions in web applications.

5 Strengths and Weaknesses

5.1 Strengths

- **Performance:** Redis excels in terms of performance, with low-latency access to data;
- **Versatility:** Its ability to function as a database, cache, message broker, etc., makes Redis versatile for various use cases.

5.2 Weaknesses

- **Single-threaded:** The single-threaded nature may become a bottleneck for certain workloads that aren't IO bound.
- **Complex Queries:** Redis may not be the best choice for scenarios that involve complex queries, as it prioritizes simplicity and speed.

Single-thread

Although it might seem counter-intuitive at first, Redis is indeed a mostly single-threaded application. The key concept here is that Redis is still IO bound, and thus it wouldn't benefit much from the performance improvement added through parallelism. Besides, making it single threaded makes it simpler to achieve desirable features, like implementing lockless atomic actions [Zul17].

To be more precise, Redis has began somewhat recently to move towards a multi-threaded architecture. Some small thread safe contexts are being moved to separate threads, but at any time Redis will still be serving a **single** request. This actually makes Redis more performant for most workloads, since you avoid thread synchronization, which is quite costly [Spe12].

In the case where your workload is having the CPU as a bottleneck, one alternative could be to use multiple Redis instances, separating your data across those instances.

4 etcd

etcd serves as a distributed configuration store, offering a reliable and distributed platform for managing key-value pairs. etcd is most well known for being a core component of Kubernetes. etcd is also used as a central control plane for distributed systems.

1 Overview

etcd, short for "distributed et cetera", is an open-source distributed key-value store designed for reliability, consistency, and simplicity in distributed systems. The distributed key-value store etcd is specifically designed for managing configuration data, coordinating distributed systems (clusters), and ensuring strong consistency through the Raft consensus algorithm [OO14]. In the case of Kubernetes, it mainly stores and manage state data, configuration data and metadata.

2 Architecture

The architecture of etcd is built around the Raft consensus algorithm, providing a fault-tolerant and consistent distributed key-value store. It operates in a cluster model, where multiple etcd nodes communicate to maintain data consistency. The Raft algorithm ensures that a leader is elected to coordinate and replicate data across the cluster, providing fault tolerance even in the face of node failures. The architecture is very robust, but there is a loss in write speed.

Figure 4.1 represents a read workflow graphically. This diagram and more are available on etcd diagrams Github.

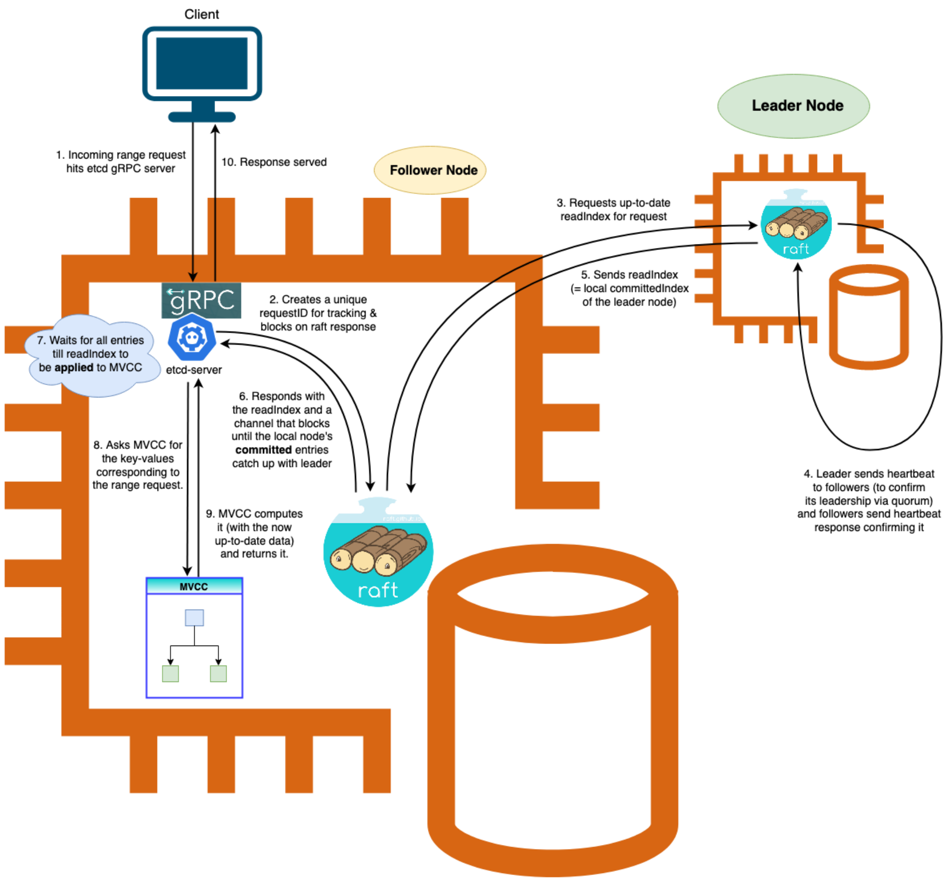
Client
1. Incoming range request hits etcd gRPC server
10. Response served
Follower Node
Leader Node
3. Requests up-to-date readIndex for request
5. Sends readIndex (= local committedIndex of the leader node)
2. Creates a unique requestID for tracking & blocks on raft response
gRPC
etcd-server
7. Waits for all entries till readIndex to be applied to MVCC
6. Responds with the readIndex and a channel that blocks until the local node's committed entries catch up with leader
8. Asks MVCC for the key-values corresponding to the range request.
9. MVCC computes it (with the now up-to-date data) and returns it.
MVCC
raft
4. Leader sends heartbeat to followers (to confirm its leadership via quorum) and followers send heartbeat response confirming it

Figure 4.1: Consistent read workflow

3 Key Features

`etcd` contains several features that make it well-suited for distributed systems:

- **Raft Consensus Algorithm:** `etcd` employs the Raft algorithm to ensure strong consistency and fault tolerance, providing a reliable foundation for distributed systems;
- **Distributed Configuration:** `etcd` excels in managing dynamic configurations across distributed applications, ensuring that all nodes have access to the latest configuration settings;
- **Atomic Transactions:** `etcd` supports atomic transactions, allowing a series of operations to be executed as a single, indivisible unit based on If/Then/Else block statements;
- **Watch Notifications:** Developers can watch for changes in `etcd` key-value pairs, enabling real-time notifications when configuration data is updated.

4 Use Cases

`etcd` finds application in scenarios that require strong consistency and distributed coordination:

- **Kubernetes clusters management:** `etcd` is the core component of Kubernetes to store and manage state data, configuration data and metadata;
- **Service Discovery and Load Balancing:** `etcd` can be used as a central registry to store information about available services, their locations, and health status. Each service instance registers itself in `etcd`, making it discoverable by other services;
- **Distributed Locking and Coordination:** `etcd` can coordinate distributed systems by providing a consistent view of shared data and managing distributed locks.

5 Strengths and Weaknesses

5.1 Strengths

- **Consistency and reliably:** `etcd` ensures strong consistency through the Raft consensus algorithm, making it reliable for critical data management.
- **Fault Tolerance:** The distributed nature of `etcd`, combined with Raft, provides fault tolerance, ensuring system reliability in the face of node failures.

5.2 Weaknesses

- **Performance for Key-Value Storage:** While `etcd` is robust in consistency and coordination, it may not match the raw performance of in-memory key-value stores like Redis, especially in scenarios prioritizing low-latency access. This will be discussed in the methodology chapter 5.
- **Specialized Use Case:** `etcd` is designed for specific use cases. `etcd` won't be efficient for other application purposes.

Raft Consensus Algorithm

One of `etcd`'s key strengths lies in its use of the Raft consensus algorithm. Raft ensures that `etcd` maintains a consistent state across nodes, making it highly reliable and fault-tolerant. In Raft, a leader is elected to manage the replication of log entries to followers, ensuring that all nodes are agree on the current state of the system. This approach provides strong consistency guarantees and makes `etcd` suitable for use cases where data integrity is essential. The Raft algorithm's simplicity and intelligibility contribute to `etcd`'s reliability in distributed environments [OO14].

In summary, `etcd` stands out as a specialized solution for scenarios requiring strong consistency, fault tolerance, and distributed coordination, making it an ideal candidate for managing configuration data and coordination in distributed systems. For our cache application, `etcd` would offer a high level of reliability and consistency, at the cost of its read/write performance. As `etcd` works with clusters of nodes and with leader nodes and followers nodes we have a high level of scalability for our cache. Moreover, each data is replicated over the nodes network. So data consistency is certified by replication of the data over the network.

5 Methodology

1 Yahoo! Cloud Serving Benchmark

We are going to use the YCSB (Yahoo! Cloud Serving Benchmark) as the benchmark for this project, to measure and compare the performance of the three different DBMSs. It has been one of the industry standard benchmarks for many years, with a focus on evaluating the performance of key-value stores and cloud serving stores [Cc23]. This benchmark evaluates the DBMSs under a series of *workloads*, simulating different use cases or scenarios for the databases at test [Bus19].

We opted for a Go implementation of the benchmark [Pin23]. We will talk more about it on section 3 of this chapter.

2 Workloads

A workload describes the type of load the DBMS will receive, and it has a significant impact on the final result observed [Coo10]. Each of them have multiple parameters, like the read proportion or the insertion proportion, and in the following section the values used are described in depth.

Each workload has two extra parameters: the Record Count and the Operations Count. The Record Count indicates how many rows are to be generated and inserted in a Load operations, whereas the Operations Count dictates the number of Read, Update, Scan, Insert and Read-Modify-Write operations are to be executed in total. Both parameters can vary per workload, but in this context they were always the same within each run of the benchmark.

It is also possible to control the amount of columns used generated for each key, but from our preliminary tests this had little impact on the results, and therefore this parameter was left with the default value (10 columns).

Workload	**Proportion**				
	Read	**Update**	**Scan**	**Insert**	**Read Modify Write**
A: Update heavy workload	0.5	0.5	0	0	0
B: Read mostly workload	0.95	0.05	0	0	0
C: Read only	1	0	0	0	0
D: Read latest workload	0.95	0	0	0.05	0
E: Short ranges	0	0	0.95	0.05	0
F: Read-modify-write	0.5	0	0	0	0.5

Table 5.1: Core Workloads Description

2.1 Core Workloads

In the YCSB benchmark, 6 core workloads are proposed, measuring different use cases [Bus20]. On Table 5.1 one can find a thorough description of the workloads.

Before executing the workloads, it is important to load the data accordingly, and ensure that the database is in the intended state (since some workloads perform insertions, the execution order has a big impact on the tests). Thus, the execution order implemented was:

1. Delete the data in the database (if any);
2. Load the database, using workload A's parameter file
3. Run workload A;
4. Run workload B;
5. Run workload C;
6. Run workload F;
7. Run workload D (This workload has side-effects on the database);
8. Delete the data in the database;
9. (Re-)Load the database, using workload E's parameter file;
10. Run workload E (This workload has side-effects on the database).

The time is measured for every Load operation as well as Run operation. Besides, in order to reduce the variability of the test measured times, each step test was executed 5 times and their values were averaged. The results are presented in Section 5.

3 go-ycsb

`go-ycsb` is the wrapper implementation of the YCSB benchmark that was used in this work. It fully supports the `benchmakrd` operators as well as all Core Workloads. Besides, it already supports multiple DBMS out of the box, facilitating the tests [Pin23].

The tool is open source, and the only change made to it was a modification in how it outputs the time. The original format of the total run time wasn't consistent, and would insert the units between the numbers (e.g. "1h23m45s"). This was changed to always show the time in seconds.

Find the complete function in the Appendix A.

4 Execution of Benchmarks

Using the designed benchmarking process and the go-ycsb tool, we executed benchmarks on the three systems at test: Redis, `etcd` and PostgreSQL.

To do so, shell scripts automating the execution of all benchmarks were developed, as the one shown below:

```
#!/bin/env bash

set -eu

for rep in {1..5}; do
    echo "Iteration ${rep}"
    for dbms in 'etcd' 'pg' 'redis'; do
        for records in 1000 2000 4000 8000 16000 32000 64000 128000; do
            operations=${records}
            echo -ne `date +"%Y/%m/%d %H:%M:%S"` "./run_benchmark ${dbms} ${
                operations} ${records}\n"

            ./run_benchmark.sh ${dbms} ${operations} ${records}
        done
    done
done
```

Of course there are other auxiliary scripts (such as the `run_benchmark.sh` one) and they are all provided alongside this report.

5 Analysis of Results

Following the execution of benchmarks, we analyzed the results obtained for the multiple workloads tested and we present our findings in this section.

Note that all plots are in log scale on the Y-axis, and the X-axis is doubling every time (1000, 2000, 4000...).

5.1 Total results

From Figure 5.1 it is possible to see that the total time is increasing linearly with the Record Count.[1] Besides, Redis seems to have a smaller total benchmark time than etcd and PostgreSQL for all Record Counts. The second fastest DBMS tested was PostgreSQL and lastly it comes etcd.

And Figure 5.2 corroborates with the linearly growing time, since the throughput is basically constant across all Record Counts for the three Database Systems.

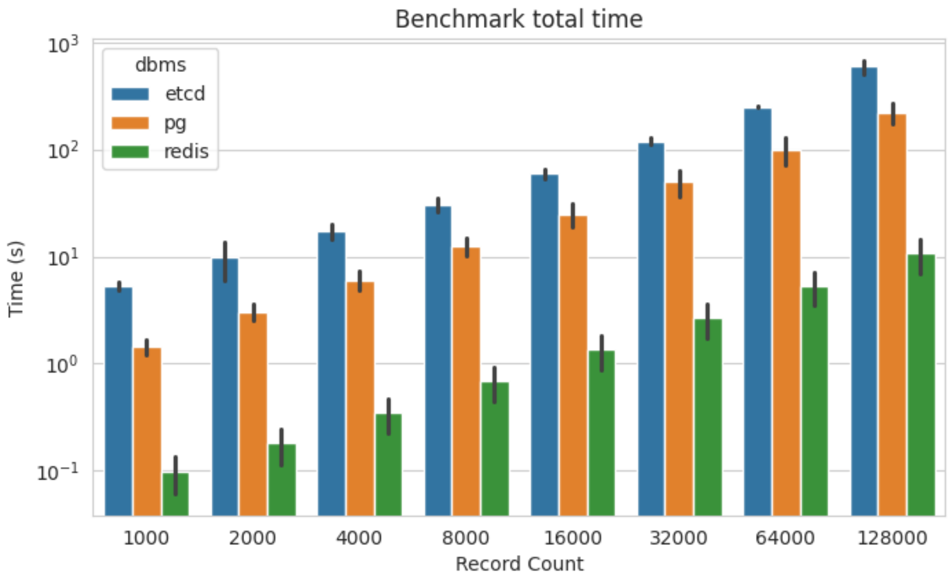

Figure 5.1: Benchmark total time for all Workloads

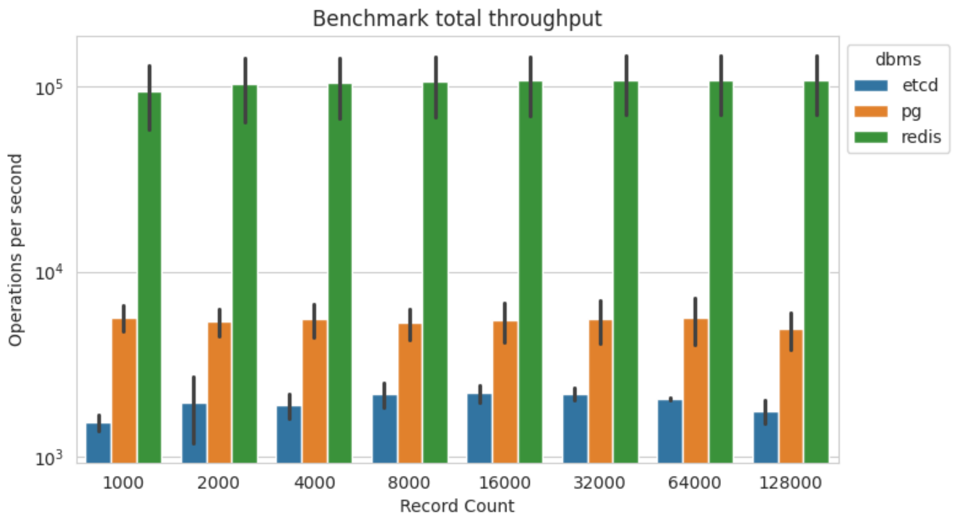

Figure 5.2: Benchmark total throughput for all Workloads

[1]Remember that the Record Count is the same as the Operations Count in our benchmark.

5.2 Load steps

Looking at the load step of the benchmark (Figures 5.3 and 5.4) we don't see much variation for Redis or PostgreSQL. There is a very small decline in the load throughput for PostgreSQL as the Record Count increases, whereas for `etcd` we see unstable amount of operations per second for Record Counts of less than 4000, for both suites, 'A' and 'E'.

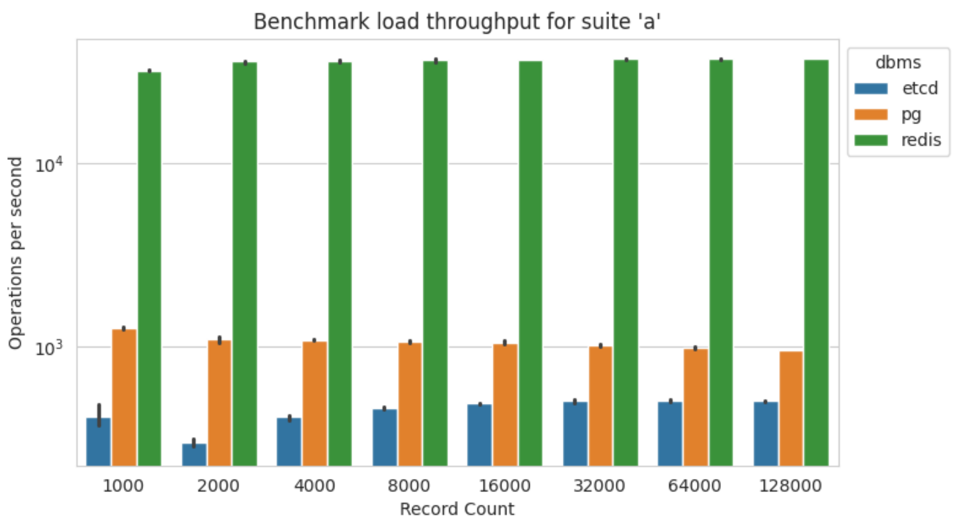

Figure 5.3: Benchmark load throughput for Workload A

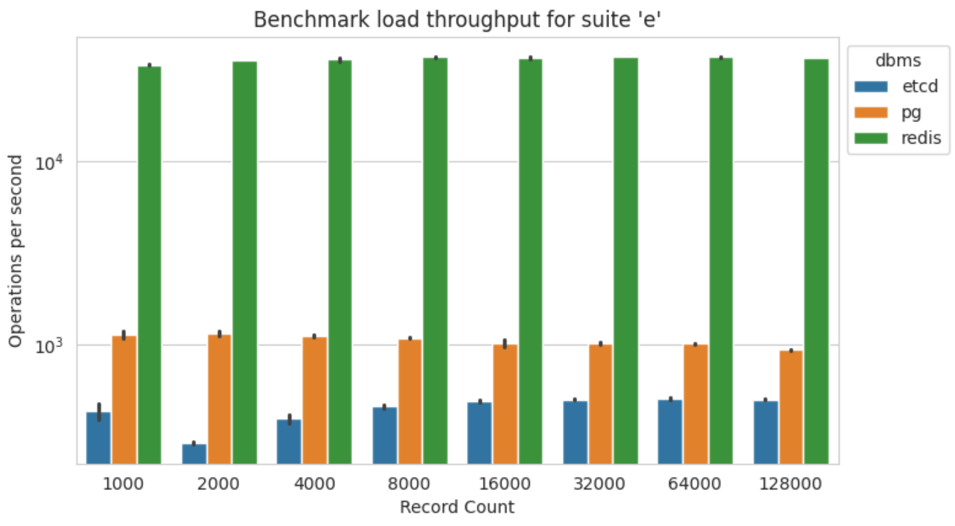

Figure 5.4: Benchmark load throughput for Workload E

5.3 Run steps

Generally speaking, the throughput for the Run steps would stabilize after 4000 Records and they were quite consistent across the workloads. Redis had the highest throughput, followed by PostgreSQL and then by `etcd` once again.

An interesting behavior can be seen on Workload E, where the performance of `etcd` degraded for higher

Record Counts. PostgreSQL also showed a strange behavior between 2000 and 16000 records, but with high variance across the test results. Thus we are attributing this to the low amount of runs executed (5), and we believe that if we were to run for a bigger number of iterations the result would be more stable.

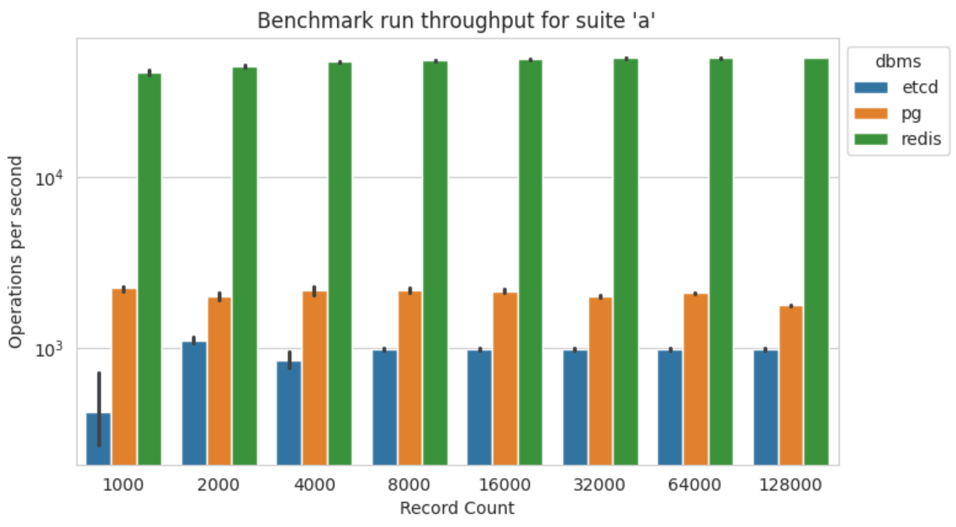

Figure 5.5: Benchmark run throughput for Workload A: Update heavy

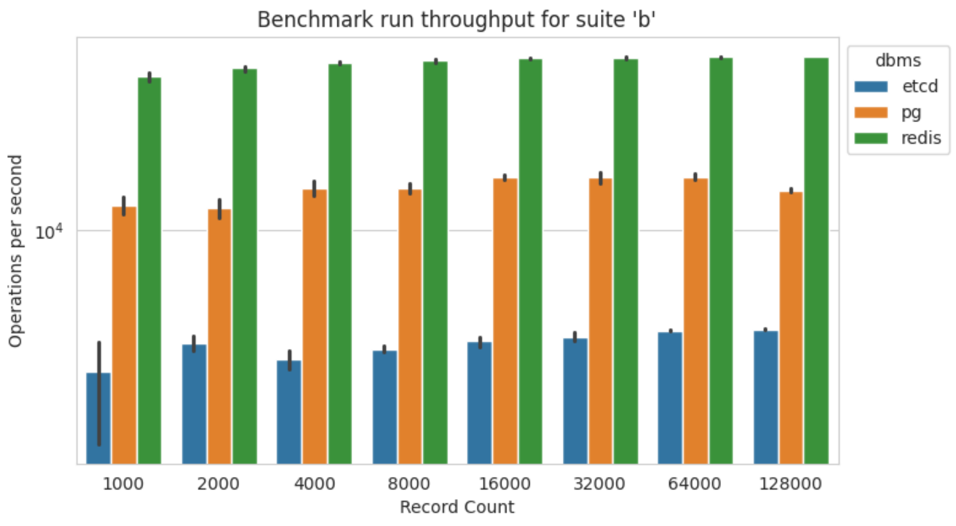

Figure 5.6: Benchmark run throughput for Workload B: Read mostly

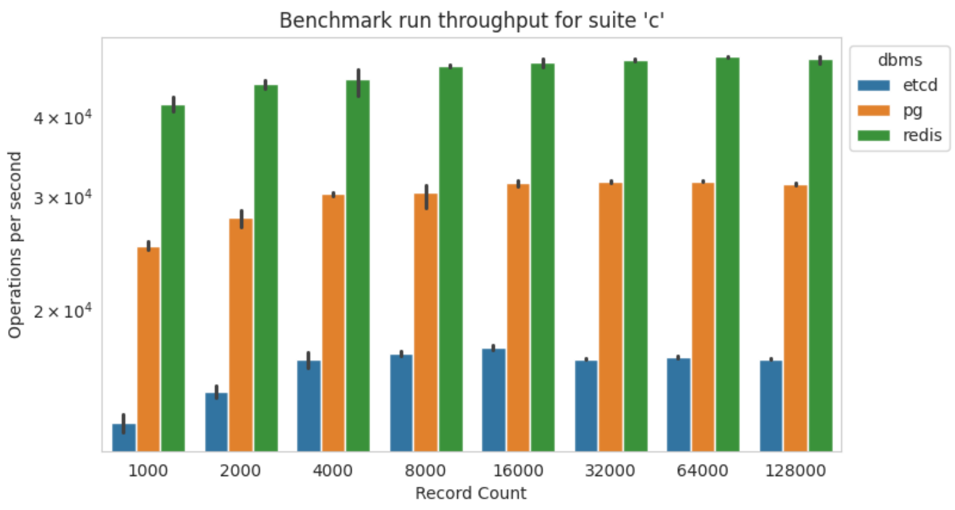

Figure 5.7: Benchmark run throughput for Workload C: Read only

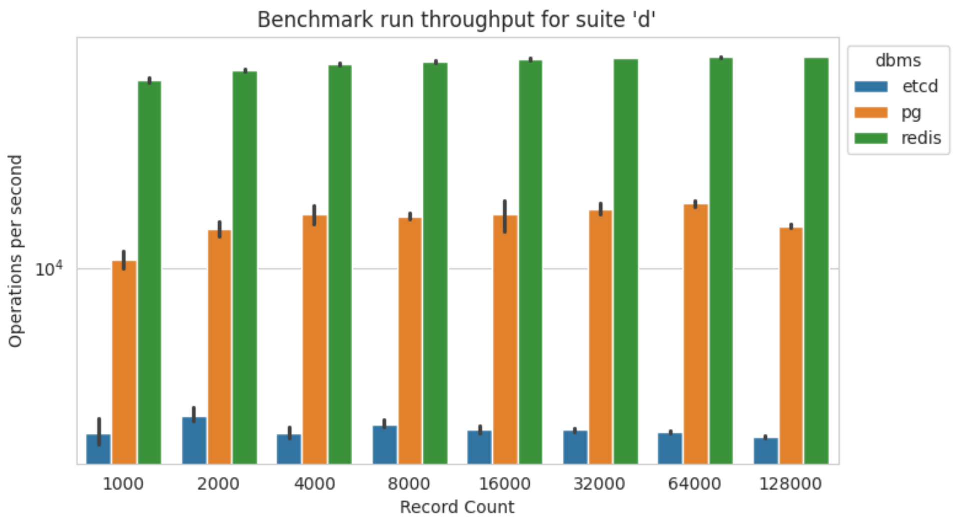

Figure 5.8: Benchmark run throughput for Workload D: Read latest

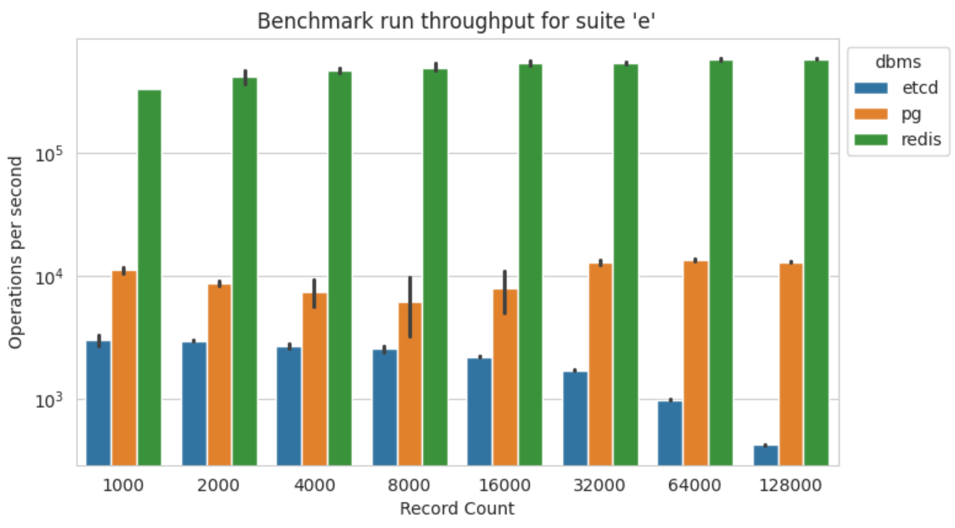

Figure 5.9: Benchmark run throughput for Workload E: Short ranges

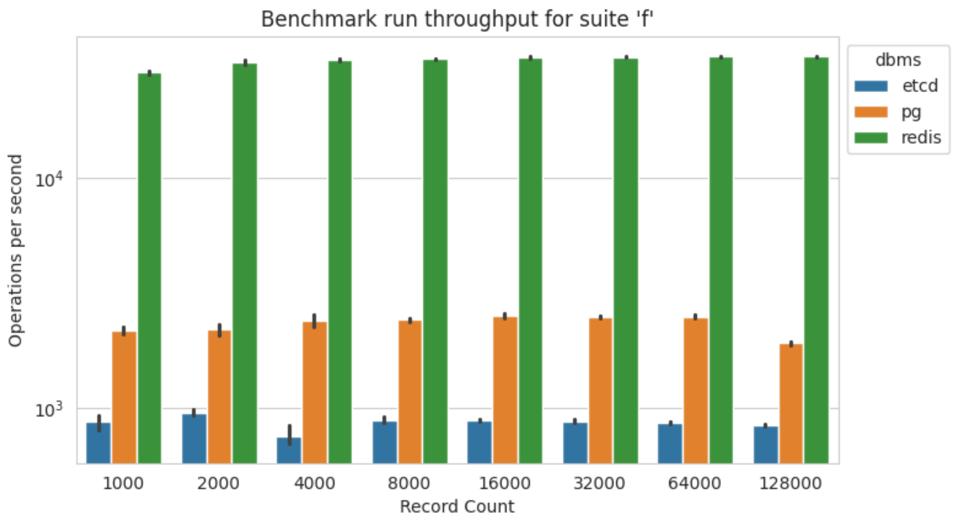

Figure 5.10: Benchmark run throughput for Workload F: Read-modify-write

6 Reproducing the results

In this section we describe the hardware utilized for running the benchmark and which Docker images were used.

6.1 Hardware utilized

The benchmark was conducted on a computer with the following hardware specifications:

- CPU: Intel Core i7-4710HQ CPU @ 2.50GHz × 4
- Memory: 16GB DDR3 1600 MHz (HMT41GS6AFR8A-PB)
- Storage: Samsung SSD 850 PRO 128GB

- Operating System: Linux Mint 20.2 Cinnamon

6.2 Docker

All the databases were running in isolated Docker containers, and the Docker commands used can be found below. Also, note that since the test for Postgres requires a database “test”, this database is created right after running the container.

```
docker run --name POSTGRES -e POSTGRES_USER=root \
    -e POSTGRES_HOST_AUTH_METHOD=trust -p 5432:5432 postgres

docker run --name REDIS -p 6379:6379 -p 8001:8001 redis/redis-stack

docker run --name ETCD -e ALLOW_NONE_AUTHENTICATION=yes -p 2379:2379 bitnami/etcd

docker exec POSTGRES psql -c 'DROP DATABASE IF EXISTS test;'
docker exec POSTGRES psql -c 'CREATE DATABASE test;'
```

6 Application Implementation

1 Description of the Chosen Application

Our chosen application is the *Weather & Quality of Life Explorer*, a sample tool designed to help users to explore cities around the world and make informed decisions about potential relocations or visits. The application leverages data from the Teleport API, which provides detailed information about weather conditions and quality of life scores for various cities.

Moreover, the main aim of the application is to help users/developers who need to fetch almost the same data repetitively in a short time. So that rather than trying to retrieve data from API in each call, Redis improves execution time by storing those data in cache after the first fetching from API.

2 Integration of Redis in the Application

In the implementation of the *Weather & Quality of Life Explorer*, Redis is seamlessly integrated as a high-performance, in-memory data store. The primary purpose of incorporating Redis into the application is to act as a caching layer for frequently accessed city information, optimizing data retrieval and enhancing overall system efficiency.

2.1 How Redis Enhances Performance

Faster Response Times:

- When a user searches for a city, the application first checks Redis for cached information.
- If the data is found in Redis, the application retrieves it almost instantly, ensuring a seamless and swift user experience.
- This is particularly crucial for popular cities that users inquire about frequently.

Reduced API Calls:

- Redis eliminates the need for redundant API calls by storing previously fetched city data.

- When a user revisits a city, the application retrieves the information from Redis, avoiding time-consuming API calls.
- This not only speeds up the user experience but also reduces the load on the Teleport API, contributing to better overall system efficiency.

Improved Responsiveness:

- Clearing the cache is a straightforward process with the option to 'Clear Cache' in the menu.
- This ensures that users always get the latest information when needed.
- The time taken to clear the cache is optimized, making the application responsive and reliable.

3 Challenges Faced during Implementation

While implementing the *Weather & Quality of Life Explorer* with Redis integration, some challenges were encountered and addressed:

1. **Cache Invalidation Strategy:**
    - Designing an effective cache invalidation strategy to ensure that users receive the latest and most accurate information when needed.
    - Implemented a mechanism to update the cache based on the freshness of the data from the Teleport API.
2. **Optimizing Cache Clearance:**
    - Ensuring that the process of clearing the cache is optimized for responsiveness.
    - Implemented efficient algorithms to clear the cache while minimizing impact on application performance.
3. **Handling Dynamic City Data:**
    - Managing dynamic city data from the Teleport API and ensuring that the cache reflects the latest changes.
    - Implemented strategies to dynamically update the cache based on real-time updates from the API.

4 Integration of etcd in the Application

As shown in result section 5 of the Methodology Chapter, Redis is the best choice for us to integrate a fast cache system. It's why we decided to rely solely on Redis to focus on observing its performance in real-life.

But etcd could provide several improvements on data reliability and consistency.

Distributed Consistency: etcd through the Raft consensus algorithm ensures consistency across a distributed system. This could be beneficial in scenarios where multiple instances of the *Weather & Quality of Life Explorer* are running, providing a more robust and fault-tolerant solution.

Advanced Transaction Support: The transactional capabilities of etcd can be advantageous for complex scenarios where atomic operations on multiple keys are required. This aligns with use cases that involve intricate data interactions and updates. It can be seen as a great improvement for our application in the future.

Dynamic Configuration Management: etcd is commonly used for dynamic configuration management in distributed systems. This could be leveraged for managing configuration parameters of the *Weather & Quality of Life Explorer*, offering flexibility and adaptability.

4.1 Potential Integration Strategies

As the goal of this project is to focus on benchmark rather than application development, we suggest a possible path to integrate etcd.

Parallel Data Storage: One approach could involve using both Redis and etcd in parallel. For example, etcd could be used for managing configuration and ensuring distributed consistency, while Redis continues to serve as the caching layer for frequently accessed city information.

Migration Considerations: For a transition from Redis to etcd we will plan data migration to avoid data inconsistencies and ensure a smooth user experience during the transition.

4.2 Challenges and Considerations

Network Latency: Integrating etcd introduces dependencies on network communication. Proper handling of potential network latency issues would be essential to maintain the responsiveness of the application.

Concurrency Control: Ensuring proper concurrency control in scenarios where multiple instances of the application are accessing etcd concurrently is crucial to avoid data conflicts and inconsistencies.

Authentication and Security: Setting up authentication and secure communication with etcd would be paramount to protect sensitive information stored in the distributed key-value store.

7 Conclusion

Our project set out to simulate a design decision by comparing traditional relational databases with key-value store databases, with a focus on Redis and etcd. The practical application chosen for evaluation, the *Weather & Quality of Life Explorer*, served as a robust use case for assessing the performance, scalability, and reliability of these databases.

1 Summary of Findings

In the realm of key-value stores, Redis emerged as a standout performer, demonstrating exceptional performance, low-latency data access, and efficient caching capabilities. Its seamless integration into the application led to faster response times, reduced API calls, and an overall improvement in system responsiveness. The benchmarks consistently underscored Redis's superior throughput compared to etcd and PostgreSQL.

While etcd (not implemented in the current version of the application) would offer potential benefits such as distributed consistency, advanced transaction support, and dynamic configuration management. The decision to prioritize Redis was driven by its exceptional performance, especially in the context of our cache-centric application.

The Yahoo! Cloud Serving Benchmark (YCSB) served as a robust evaluation tool, providing valuable insights into the behavior of each database system under various workloads. The results consistently showcased Redis's efficiency, supporting our decision to utilize it for our caching mechanism.

All in all, it is important to note how important the choice of the appropriate DBMS is for an application, and that there are multiple options available that should be considered.

2 Recommendations for Future Work

For future enhancements, the integration of etcd could be considered in specific areas of the application requiring distributed consistency and advanced transactional capabilities. Careful planning and consideration of challenges such as network latency, concurrency control, and security measures are essential for the success of such an integration.

Additionally, to further optimize the application and similar systems, future work could explore specific strategies for enhancing Redis integration, addressing any identified limitations in the caching mechanism.

In conclusion, our project provides valuable insights into the strengths and weaknesses of key-value store databases, assisting developers in making informed decisions for their applications. The performance benchmarking results and integration considerations lay the foundation for ongoing optimization and expansion of the *Weather & Quality of Life Explorer* and similar applications, contributing to the evolving landscape of database management systems.

Bibliography

[Bus19] Busbey, Sean (2019). *Yahoo! Cloud Serving Benchmark (YCSB).* `https://github.com/brianfrankcooper/YCSB/wiki`.

[Bus20] — (2020). *Core Workloads.* `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`.

[Coo10] Cooper, Brian (2010). *Implementing New Workloads.* `https://github.com/brianfrankcooper/YCSB/wiki/Implementing-New-Workloads`.

[Cc23] Cooper, Brian F. and contributors (2023). *YCSB - Yahoo! Cloud Serving Benchmark.* URL: `https://github.com/brianfrankcooper/YCSB`.

[OO14] Ongaro, Diego and John Ousterhout (May 20, 2014). "In Search of an Understandable Consensus Algorithm (Extended Version)". In.

[Pin23] PingCAP (2023). *go-ycsb.* `https://github.com/pingcap/go-ycsb`.

[Red23] Redis (2023). *Redis Documentation.* `https://redis.io/documentation`.

[Spe12] Spezia, Didier (2012). *Redis is single-threaded, then how does it do concurrent I/O?* URL: `https://stackoverflow.com/questions/10489298/redis-is-single-threaded-then-how-does-it-do-concurrent-i-o`.

[Zul17] Zulauf, Carl (2017). *why Redis is single threaded(event driven).* URL: `https://stackoverflow.com/questions/45364256/why-redis-is-single-threadedevent-driven/45374864#45374864`.

GitHub Repository Link

You can find the source code on GitHub: https://github.com/ranaislek/advancedDB-keyValue.

8 Appendices

1 Appendix A - runClientCommandFunc

Updated runClientCommandFunc to show time in seconds

```
func runClientCommandFunc(
    cmd *cobra.Command,
    args []string,
    doTransactions bool,
    command string) {
        dbName := args[0]

        initialGlobal(dbName, func() {
                doTransFlag := "true"
                if !doTransactions {
                        doTransFlag = "false"
                }
                globalProps.Set(prop.DoTransactions, doTransFlag)
                globalProps.Set(prop.Command, command)

                if cmd.Flags().Changed("threads") {
                        // We set the threadArg via command line.
                        globalProps.Set(prop.ThreadCount, strconv.Itoa(threadsArg))
                }

                if cmd.Flags().Changed("target") {
                        globalProps.Set(prop.Target, strconv.Itoa(targetArg))
                }
```

```
            if cmd.Flags().Changed("interval") {
                globalProps.Set(prop.LogInterval,
                    strconv.Itoa(reportInterval))
            }
        })

        fmt.Println("*****************␣properties␣*****************")
        for key, value := range globalProps.Map() {
            fmt.Printf("\"%s\"=\"%s\"\n", key, value)
        }
        fmt.Println("**********************************************")

        c := client.NewClient(globalProps, globalWorkload, globalDB)
        start := time.Now()
        c.Run(globalContext)
        fmt.Println("**********************************************")
        fmt.Printf("Run␣finished,␣takes␣%.3fs\n", time.Now().Sub(start).Seconds())
        measurement.Output()
}
```