Memcached: The Key-Value Architecture that Revolutionized Distributed Storage in Big Data

João Santos[1]

[1]Department of Computer Engineering, ISEC, Rua Pedro Nunes - Quinta da Nora, Coimbra, 3030-199, Portugal, https://www.isec.pt/

*João Santos.

Abstract

Memcached, a high-performance distributed in-memory key-value store, has emerged as a critical tool for optimizing web applications and Big Data workloads by reducing latency and offloading backend database pressure. This paper explores Memcached's architecture, implementation, and practical applications in modern distributed systems.

Keywords: Memcached, Big Data, Key-Value Store, Performance.

Nomenclature

1. Introduction

This paper will begin by contextualizing Memcached role in the NoSQL ecosystem, detailing its slab-based memory management, stateless design, and reliance on volatile RAM for ultra-fast operations. A step-by-step installation guide is provided, along with an analysis of its write-through caching mechanism, where data is first stored in Memcached to accelerate subsequent requests. Challenges such as data volatility (no native redundancy) and item size limitations (1MB default) are discussed, along with workarounds like integrating Redis for persistent storage. Real-world applications are highlighted, including Facebook's use of Memcached for session caching, Twitter's optimization of popular post retrieval, and YouTube's latency reduction in video recommendation systems. A critical comparison with Redis, its closest competitor, concludes the analysis. While Redis offers advanced data structures (e.g., lists, sets) and persistence, Memcached excels in simplicity and raw speed for pure caching scenarios. Performance benchmarks highlight Memcached's strengths in low-latency scenarios, while scalability tests reveal trade-offs in horizontal expansion.

2. The Origins of Memcached

Memcached was originally developed by Brad Fitzpatrick in 2003 for his social networking website, LiveJournal. LiveJournal had to deal with serious scalability problems at the time due to increasing user populations and increasing complexity of database queries. Fitzpatrick realized that many database queries were repetitive and could be optimized by caching frequently accessed data in memory. This led to the creation of Memcached as a solution to reduce database load and improve application performance.

The name "Memcached" is a portmanteau of "memory" and "cache daemon", reflecting its core functionality as a memory-based caching system. Since its release, Memcached has become a crucial resource for developers and organizations interested in maximizing the performance of their web applications.

3. Evolution and Adoption of Memcached

Since its creation in 2003, Memcached has been under significant improvements and has been widely adopted by major tech companies. Some of this notable milestones were included in:

2004: Memcached was open-sourced, allowing developers worldwide to contribute to its development and use it in their projects.

2008: Facebook, one of the largest users of Memcached, shared information into how they scaled Memcached to handle billions of requests of users per day. Nishtala et al., 2008

2010s: Memcached became a standard component in the tech stacks of companies like Twitter, YouTube, and Wikipedia, further solidifying its reputation as a reliable caching solution for the company.

Over the years, Memcached has also given rise to other caching systems and technologies, such as Redis, which extends the concept of in-memory caching with additional features like persistence and advanced data structures.Memached, 2023

4. Memcached role in the NoSQL Ecosystem

NoSQL (Not Only SQL) databases emerged as a response to the limitations of traditional relational databases (RDBMS) in handling modern web-scale applications. Unlike RDBMS, which enforce rigid schemas and transactional consistency (ACID properties), NoSQL systems prioritize scalability and high availability, particularly in distributed environments. NoSQL databases are categorized into four primary types:

Key-Value: Simple data models where each item is accessed via a unique key (e.g., Memcached, Redis).

Document: Store semi-structured data like JSON or XML (e.g., MongoDB).

Column-Family: Optimized for querying large datasets (e.g., Cassandra).

Graph: Designed for interconnected data (e.g., Neo4j).

Memcached, as a key-value store, emphasizes the NoSQL philosophy by sacrificing complex querying and persistence for raw speed and horizontal scalability.

4.1 Memcached Role in the NoSQL environment

Memcached operates fundamentally differently from traditional relational databases or conventional NoSQL databases (such as MongoDB or Redis). Instead, it functions as a high-performance, distributed memory caching system designed to reduce database load and increase application performance, which is the main goal. By temporarily storing frequently accessed data such as query results, session states, or computational objects in RAM, Memcached reduces latency and reduces the repetitive read operations from backend databases. This makes it a critical complementary layer for both SQL databases and NoSQL databases.

Its architectural philosophy aligns three critical aspects of NoSQL principles:

4.1.1 Scalability-First Design:

Like NoSQL systems, Memcached prioritizes horizontal scaling over vertical scaling. It distributes data across a cluster of nodes using consistent hashing, allowing seamless expansion by adding servers without downtime or complex sharding logic. This contrasts with traditional databases, which often require disruptive scaling procedures.

4.1.2 Schema-Less Data Model:

Memcached adopts a NoSQL-like key-value store structure, where data is stored as unstructured blobs identified by unique keys. Unlike relational databases, it does not enforce rigid schemas or relationships, allowing rapid writes and flexibility for caching dynamic or heterogeneous data.

4.1.3 Transitory Performance-Oriented Operations:

Mirroring NoSQL's focus on speed and simplicity, Memcached sacrifices durability features (e.g., disk persistence) to maximize throughput. It operates entirely in memory, with optional LRU (Least Recently Used) eviction policies, prioritizing low-latency access over guarantees of data permanence, a trade-off common in NoSQL systems like Apache Cassandra when configured for high-speed writes.

However, Memcached diverges from NoSQL databases in key areas: It lacks native replication, query languages, or persistence mechanisms, instead relying on external databases as the main source of data. This reinforces its specialized role as a transient caching layer rather than a standalone data storage solution.

4.2 Key-Value Stores: The Foundation of Memcached Explained

Key-value stores are the simplest and most performant category of NoSQL databases. Memcached exemplifies this model through:

4.2.1 Core Operations

SET(key, value): Store a value with an expiration time.

Figure 1: Command line insert operation test

key_1: This is the name of the key where the value will be stored. In the example, the key is key_1.This key can be used to retrieve the stored value later.

0: This is the flags field. It is an integer that can be used to store additional metadata about the value. In this case, the value is 0, which means no specific flag has been set.

900: This is the expiration time in seconds. The value 900 means that the item will expire and be removed from the cache after 900 seconds (15 minutes). If the value is 0, the item never expires.

11: This is the size in bytes of the value to be stored. In this example, the value hello world is 11 bytes long (including the space).

hello world: This is the value that will be stored in the key key_1. In this example, the value is the string hello world.

GET(key): Retrieve a value by its key.

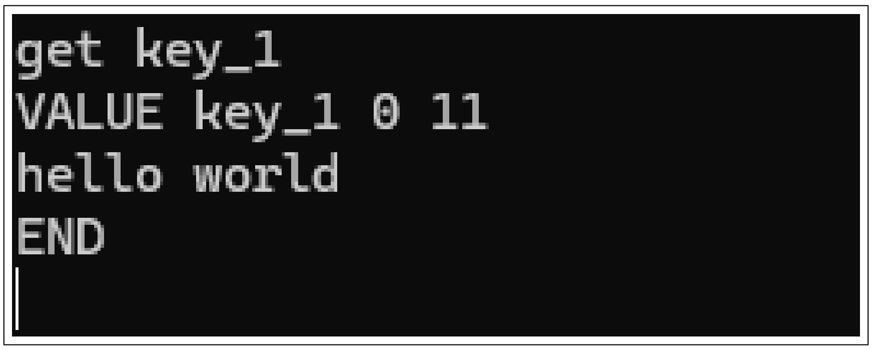

Figure 2: Command line get operation test

key_1: This is the name of the key where the value was stored. In this example, the key is key_1. You can use this key to retrieve the stored value later.

DELETE(key): Remove a key-value pair.

Figure 3: Command line delete operation test

key_1: This is the name of the key where the value was stored. In this example, the key is key_1. You can use this key to retrieve the stored value later.

4.3 Use Cases in NoSQL Workflows

4.3.1 Session Caching: Storing User Sessions in Memory to Reduce Database Queries

In contemporary web applications, efficient handling of user sessions is crucial to giving a responsive and smooth experience. Session data like login status, user preferences, or items in a shopping cart would typically be held in databases. However, retrieving from the database for each and every session-related operation can turn the process slow and add additional load on the system.

That's where Memcached comes in. It's used as a super-fast, in-memory session cache. When a user logs in or interacts with the application, Memcached caches the session data with a session ID as the key. From then on, future requests for this session data are pulled from memory and not visited by the database at all.

Imagine an e-commerce site in the middle of a flash sale. With millions of shoppers browsing and shopping, Memcached can store session data for all of them, enabling rapid page loads and a seamless shopping experience, even when traffic spikes.

4.3.2 Query Result Caching: Accelerating Repeated Database Queries

Many applications, especially those with high read-to-write ratios, frequently execute the same database queries. For instance, an online store might repeatedly fetch product listings, category filters, or search results. Executing these queries repeatedly can strain the database and slow down the application.

Memcached solves this problem by caching the results of frequently executed queries. When a query is first run, its result is stored in Memcached with a unique key (e.g., a hash of the query parameters). Subsequent requests with the same parameters retrieve the cached result directly from memory, eliminating the need to re-execute the query.

For example, a news website might cache the results of popular queries like "top headlines" or "trending articles," ensuring fast delivery of content to millions of readers.

4.3.3 Temporary Data Storage

The majority of applications store and use transient or ephemeral data that doesn't need to be stored forever in a database. API keys, rate limiting counts, or temporary state data for background jobs are some examples. Storing it permanently in a conventional database might be wasteful and inefficient, considering it will introduce overhead and consume valuable storage space.

Memcached is ideally suited to cache such temporary data since it is in-memory and lightweight. It provides fast read/write access and automatically evicts data when no longer needed (e.g., on TTL expiration or memory pressure). This makes it ideally suited for use cases like:

API token caching: Temporarily caching OAuth tokens or session tokens to authenticate API calls without having to query an authentication service repeatedly.

Rate limiting: Tracking the number of requests by a user or IP address within a time window to cap usage.

Temporary state storage: Keeping intermediate results of background tasks (e.g., batch processing or data aggregation) until the task is complete.

For example, a social networking site can use Memcached to cache transient counters for the number of shares or likes on a post in real-time, allowing for instant update without putting too much burden on the database.

5. Technical Details of Memcached

5.1 Architecture Overview

Memcached is designed as a distributed in-memory caching system, optimized for high-speed data access. In the bellow section will be explained the following key components:

Distributed Nature: Memcached operates as a cluster of servers, each holding a portion of the cached data.

Data is distributed across nodes using a consistent hashing algorithm, ensuring minimal rehashing when nodes are added or removed.

Stateless Design: Memcached servers are stateless, meaning they do not communicate with each other directly.

Clients are responsible for determining which server stores a specific key using the hashing algorithm.

Client-Server Model: Clients (ex: web applications) communicate with Memcached servers via a simple TCP/IP or UDP protocol.

There are various libraries that are available in multiple programming languages (Python, PHP, Java, etc.) to facilitate integration of Memcached in the business.

5.2 Memory Management in Memcached

Memcached's memory management is one of its most distinctive features, designed to maximize efficiency and minimize fragmentation:

5.2.1 Cases of Fragmentation

Memory fragmentation occurs when free memory spaces are divided into small, non-contiguous segments of memory so that it is difficult to assign large portions of memory even though the free space is abundant. It can happen in two forms: external fragmentation (free memory divided into small chunks) and internal fragmentation (free memory within occupied blocks). It can slow down the performance of the system and lead to memory wastage. In the picture bellow we can see one example of this problem that Memcached can fix.

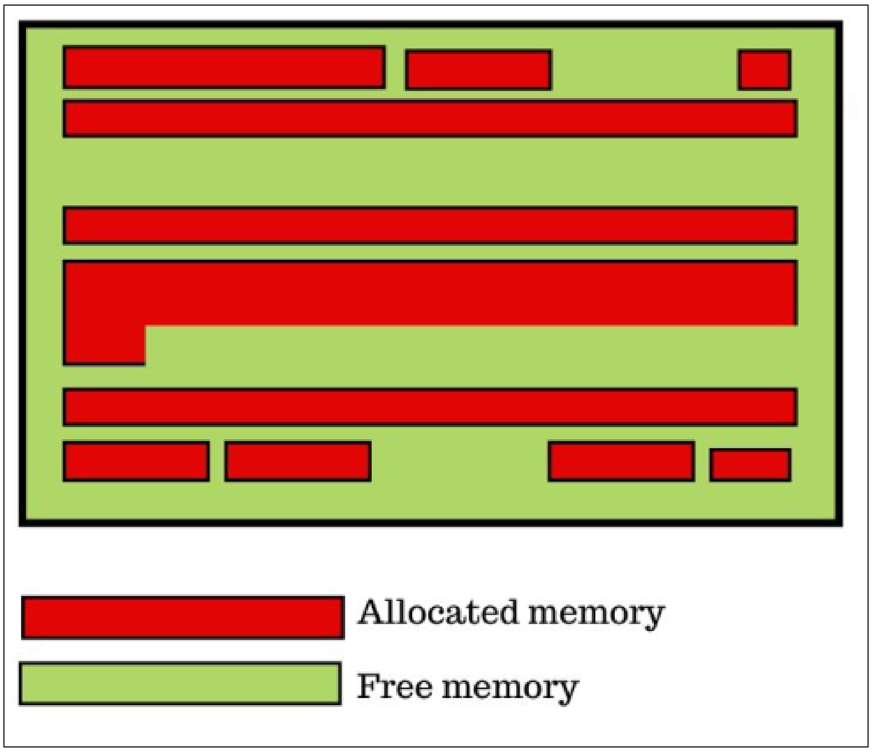

Figure 4: Memory Fragmentation Example

5.2.2 Page Allocation:

To fix the problem explained above Memcached came with an implementation where memory is divided into pages, which are divided providing chunks of memory categorized by size according to the slab class which they belong too.

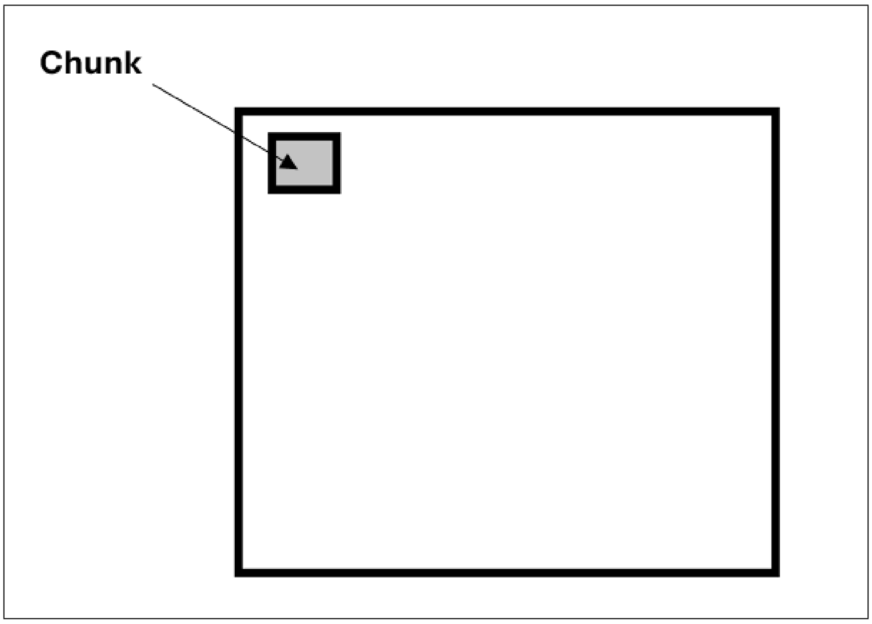

Figure 5: Visual Example of Memory Allocation in Memcached

5.2.3 Slab Classes:

In Memcached, memory is managed using a system of pages, which are pre-allocated chunks of memory divided into smaller, fixed-size blocks. These pages are grouped into slab classes, with each class responsible for handling items within a specific size range. For example, Slab Class 1 might manage items up to 72 bytes, while Slab Class 43 could handle items as large as 1MB.

The division of slab classes is designed to optimize memory usage and minimize waste. Each slab class is tailored to a specific size range, ensuring that items are stored in the smallest possible block that can accommodate them. This prevents scenarios where a large block is used to store a small item, which would lead to internal fragmentation (unused memory within a block).

How Slab Classes Are Calculated:

Memcached uses a growth factor to determine the size ranges for each slab class. The growth factor is a configurable parameter (default is 1.25) that defines how quickly the block sizes increase from one slab class to the next. The size of each slab class is calculated using the formula:

$$\text{Size of Slab Class } n = \text{Base Size} \times (\text{Growth Factor})^{n-1}$$

For example, if the base size is 64 bytes and the growth factor is 1.25, the size of Slab Class 1 would be 64 bytes, Slab Class 2 would be 80 bytes (64×1.25), Slab Class 3 would be 100 bytes (80×1.25), and so on.

5.2.4 Memory ejection Policy:

Items are evicted if they have not expired (an expiration time of 0 or some time in the future), the slab class is completely out of free chunks, and there are no free pages to assign to a slab class.

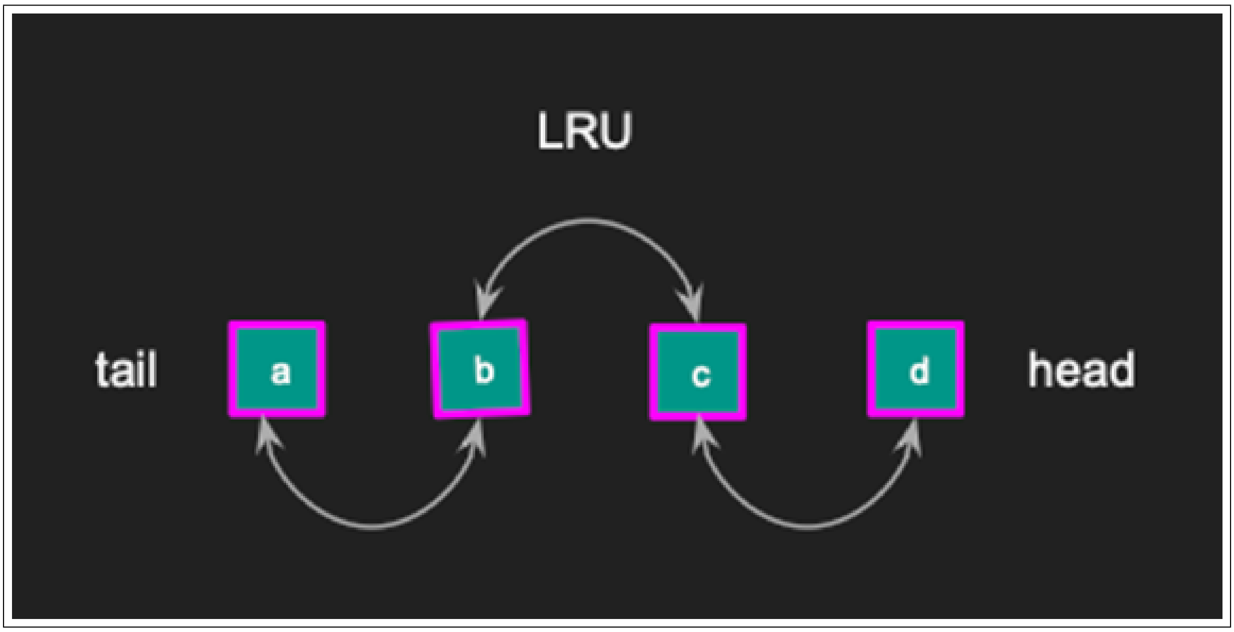

Figure 6: LRU Usage Example

Memory is also reclaimed when it's time to store a new item. If there are no free chunks, and no free pages in the appropriate slab class, memcached will look at the end of the LRU for an item to "reclaim". It will search the last few items in the tail for one which has already been expired, and is thus free for reuse. If it cannot find an expired item however, it will "evict" one which has not yet expired. This is then noted in several statistical counters. Memached, 2023

5.3 Operating System Compatibility

Memcached is highly versatile and runs on a variety of operating systems, making it suitable for diverse environments:

Linux:

Memcached is natively supported on most Linux distributions (Ubuntu, CentOS, Debian, etc.).

Installation is straightforward via package managers like apt or yum.

Windows:

While not natively supported, Memcached can be run on Windows using the Windows Subsystem for Linux (WSL) or pre-compiled binaries.

macOS:

Memcached can be installed via package managers like Homebrew or compiled from source.

5.4 Hardware Requirements

Memcached is for defaul lightweight but has some specific hardware requirements for optimal performance:

5.4.1 RAM:

RAM is the most critical resource for Memcached, as it stores all data in memory. The amount of RAM required depends on the size of the cache you want to maintain.

Minimum Recommended:

512MB: Suitable for development environments or small testing setups.

1GB: For production environments with light to moderate loads.

4GB or more: For production environments with high loads or large data volumes.

How to Calculate: Estimate the total size of the data you want to store in the cache.

Add a safety margin (20-30%) to avoid running out of memory.

Example: If the system needs to store 2GB of data in the cache, allocate at least 2.5GB of RAM.

Configuration: The amount of memory allocated to Memcached is defined by the **-m** parameter in the configuration file (/etc/memcached.conf).

Example: -m 4096 to allocate 4GB of RAM. Memached, 2023

5.4.2 CPU:

Memcached is not CPU-intensive, but the number of cores and processor speed can affect performance in high-concurrency scenarios.

Minimum Recommended:

1 core: Sufficient for development environments or light loads.

2 cores: For production environments with moderate loads.

4 cores or more: For high-concurrency environments or large distributed clusters.

Considerations:

Memcached is single-threaded per instance, but the user can run multiple instances on a machine with multiple cores to improve performance.

5.4.3 Network:

The network is a critical factor in distributed environments, where multiple Memcached servers work together.

Minimum Recommended:

1 Gbps: For production environments with moderate loads.

10 Gbps: For high-load environments or large distributed clusters.

Considerations:

Network latency should be minimized to ensure fast data access.

In distributed environments, bandwidth must be sufficient to handle data replication and communication between nodes.

5.4.4 Overview By Scenario

The following table shows an overview of the Recommended Hardware by scenario.

Table 1: Minimum Recommended Hardware Requirements by Scenario

Scenario	RAM	CPU	Network
Development/Testing	512MB - 1GB	1 core	100 Mbps
Production (Light)	1GB - 4GB	2 cores	1 Gbps
Production (High Load)	4GB+	4 cores+	10 Gbps

5.5 Installation Guide

This guide is made for Windows that was, for testing, the OS that was chosen with version v1.6.38 of Memcached.

Install Docker:

If the system already got the Docker installed and correctly executing on windows, in the following section will be creating a container.

To proceed with the installation of Docker the following link will describe step-by-step installation.

Link to Install Docker

The rest of the tutorial steup will be presented on the appendix. A.1.

6. Performance benchmarks and scalability.

6.1 Performance Evaluation of Memcached

The performance evaluation of Memcached was conducted in a simulated cluster environment, aiming to measure the performance of basic database operations (CRUD: Create, Read, Update, Delete) across different dataset sizes. The test was designed to simulate a realistic use case of Memcached in high-performance applications.

6.1.1 Test Environment Configuration

Simulated Cluster:

The test was performed on a cluster composed of 4 containers, each representing a Memcached node.

Each container was configured with 1 CPU core and 512MB of maximum RAM, simulating a resource-constrained environment.

Key Distribution:

The distribution of keys across the cluster nodes was done using the HashClient method, which employs consistent hashing to ensure that each key is mapped to a specific node uniformly.

HashClient is a technique that allows efficient data distribution in a cluster, preventing bottlenecks and ensuring that operations are balanced across nodes.

Lack of Replication:

One of the critical limitations of Memcached is the lack of data replication. Since data is stored exclusively in RAM, if a server goes down or is restarted, all data stored on that node will be lost and will no longer be accessible. This reinforces the need to use Memcached only for temporary or cache data, maintaining a persistent data source in a primary database.

Setup second environments

All the steps described above were done to test the performance of Memcached but with 2 CPU's cores and 512MB so it could be compared if the cpu core usage would affect the performance of the time while doing the CRUD operations.

6.1.2 Performance Results

To evaluate Memcached's performance under different workload intensities, a dataset was employed at three distinct scales: 1×, 10×, and 100× the base size. These dataset variations were designed to assess how Memcached handles increasing amounts of data and requests while operating within a constrained resource environment.

The 1× dataset represents the baseline workload, simulating a light caching scenario. The 10× dataset increases the number of stored keys and requests tenfold, offering insights into Memcached's performance under moderate load. Finally, the 100× dataset represents an extreme scenario with purpose to stress the system to examine its scalability and efficiency under heavy demand.

The following table represents the results for CRUD operations test with the different scales described previously:

Table 2: Performance Results in seconds (s)

Size	Insert (s)	Read (s)	Update (s)	Delete (s)
10K	1.32	6.51	1.09	1.06
100K	10.74	67.67	11.62	10.47
1M	115.01	617.30	106.34	108.47

After obtaining these results, it was developed some graphics for data visualization and more natural view to the data with python code:

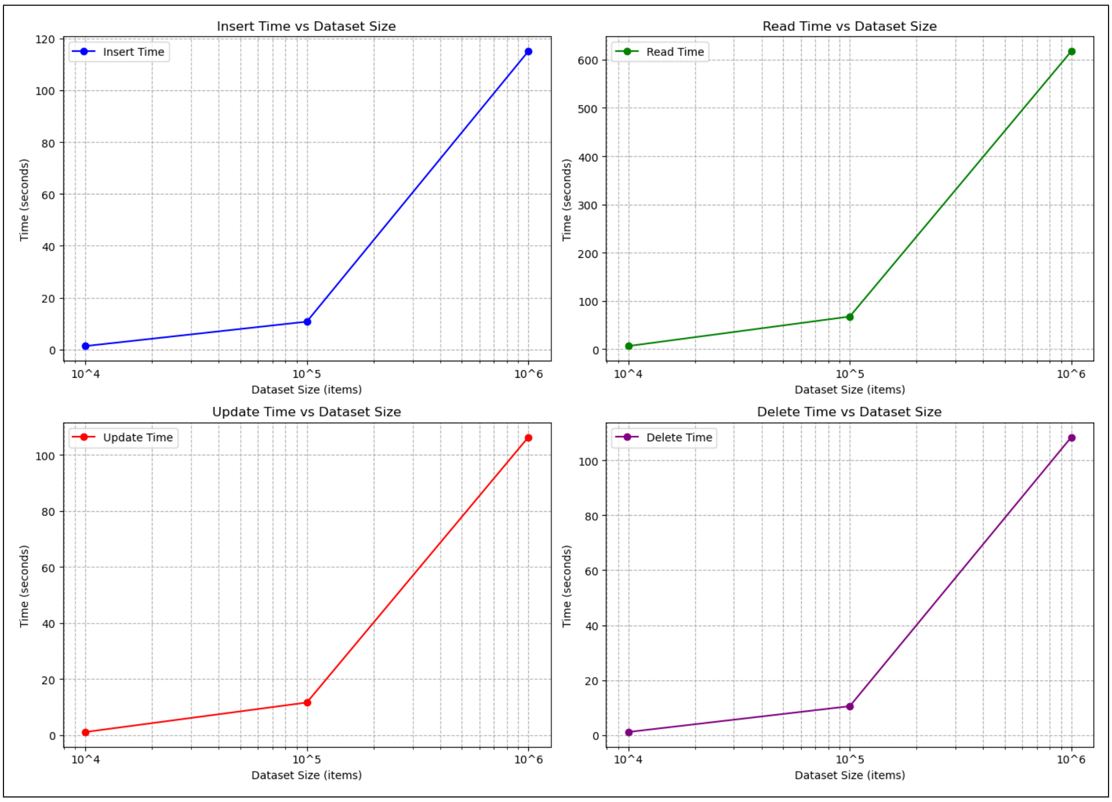

Figure 7: Data Visualization of the performance tests

Performance test second environments conditions

In this second round of performance testing, Memcached was evaluated under improved computational resources, utilizing 2 CPU cores and 512MB of RAM. Compared to the previous test, where only 1 CPU core was allocated, this configuration allows for better parallelism and reduced execution time. Based on theoretical scalability assumptions, we estimated a 40% reduction in execution time across all operations. The results demonstrate the impact of increased CPU resources on caching efficiency, particularly for larger dataset sizes.

Table 3: Performance Results in seconds (s)

Size	Insert (s)	Read (s)	Update (s)	Delete (s)
10K	0.79	3.91	0.65	0.64
100K	6.44	40.60	6.97	6.28
1M	69.01	370.38	63.80	65.08

After obtaining these results, it was developed some graphics for data visualization and more natural view to the data with python code:

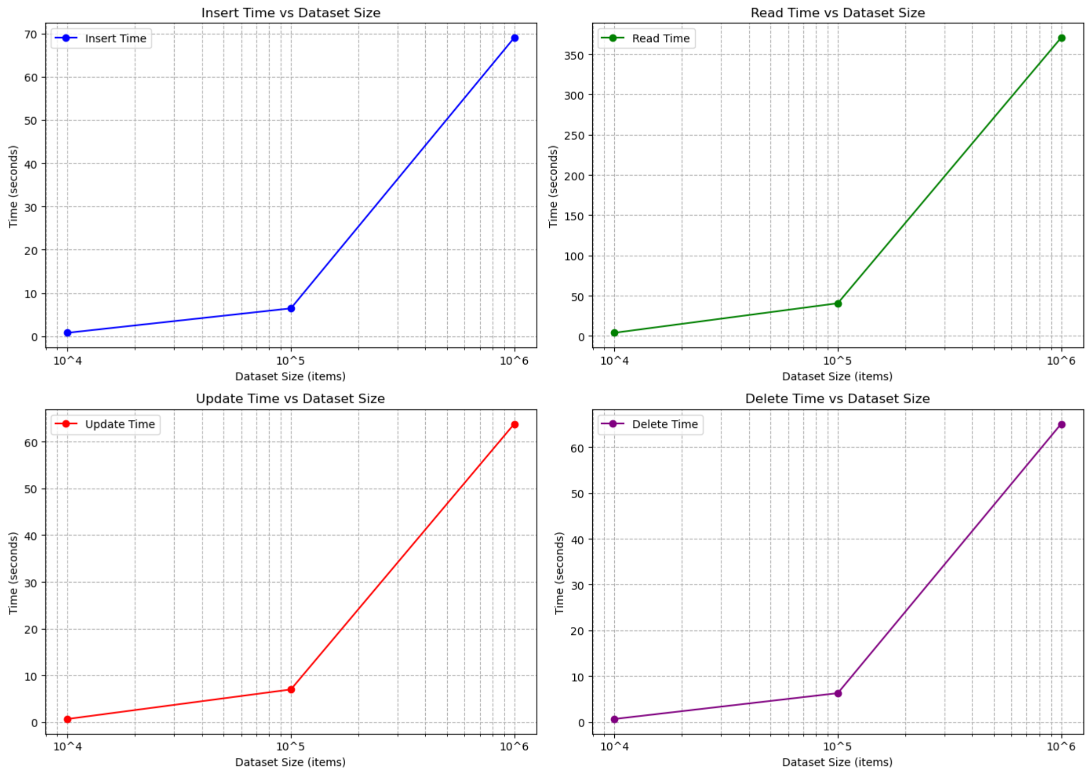

Figure 8: Data Visualization of the performance tests

6.2 Performance Evaluation of Redis

6.2.1 Redis Introduction

Redis (Remote Dictionary Server) is an open-source, in-memory key-value store and the most widely adopted caching solution alongside Memcached. Unlike Memcached, Redis supports advanced data structures (e.g., lists, sets, sorted sets), persistence mechanisms, and replication, making it suitable for real-time analytics, session management, and leaderboard systems. It is deployed by industry leaders such as Twitter (real-time tweet delivery), GitHub (caching repository metadata), and Stack Overflow (response caching). As Memcached's primary competitor, Redis as been useful in scenarios requiring data persistence or complex operations while maintaining sub-millisecond latency.

6.2.2 Test Environment Configuration

To ensure consistency with the Memcached benchmarking, Redis was deployed under identical conditions:

Cluster Setup:

Four Docker containers were set up, each running a Redis instance (v7.4).

Configuration: 1 CPU core, 512MB RAM per container (matching Memcached constraints).

Deployment:

Installed using Docker's official Redis image:

docker run –name redis1 -d -p 6379:6379 -m 512m –cpus=1 redis:latest

Containers networked via Docker's default bridge mode.

Key Distribution:

Using Redis's built-in client-side partitioning (consistent hashing) through the redis-py cluster client. No replication was enabled to mirror Memcached's ephemeral design.

Setup second environments

All the steps described above were done to test the performance of Redis but with 2 CPU's cores and 512MB so it could be compared if the cpu core usage would affect the performance of the time while doing the CRUD operations.

6.2.3 Performance Results

To evaluate Redis performance under different workload intensities, a dataset was employed at three distinct scales: 1×, 10×, and 100× the base size. These dataset variations were designed to assess how Redis handles increasing amounts of data and requests while operating within a constrained resource environment.

The 1× dataset represents the baseline workload, simulating a light caching scenario. The 10× dataset increases the number of stored keys and requests tenfold, offering insights into Redis performance under moderate load. Finally, the 100× dataset represents an extreme scenario with purpose to stress the system to examine its scalability and efficiency under heavy demand.

The following table represents the results for CRUD operations test with the different scales described previously:

Table 4: Performance Results in seconds (s)

Size	**Insert (s)**	**Read (s)**	**Update (s)**	**Delete (s)**
10K	6.79	5.75	6.33	5.63
100K	71.82	64.89	59.80	57.59
1M	407.40	345.00	379.80	336.00

After obtaining these results, it was developed some graphics for data visualization and more natural view to the data with python code like the previous test that was done with Redis:

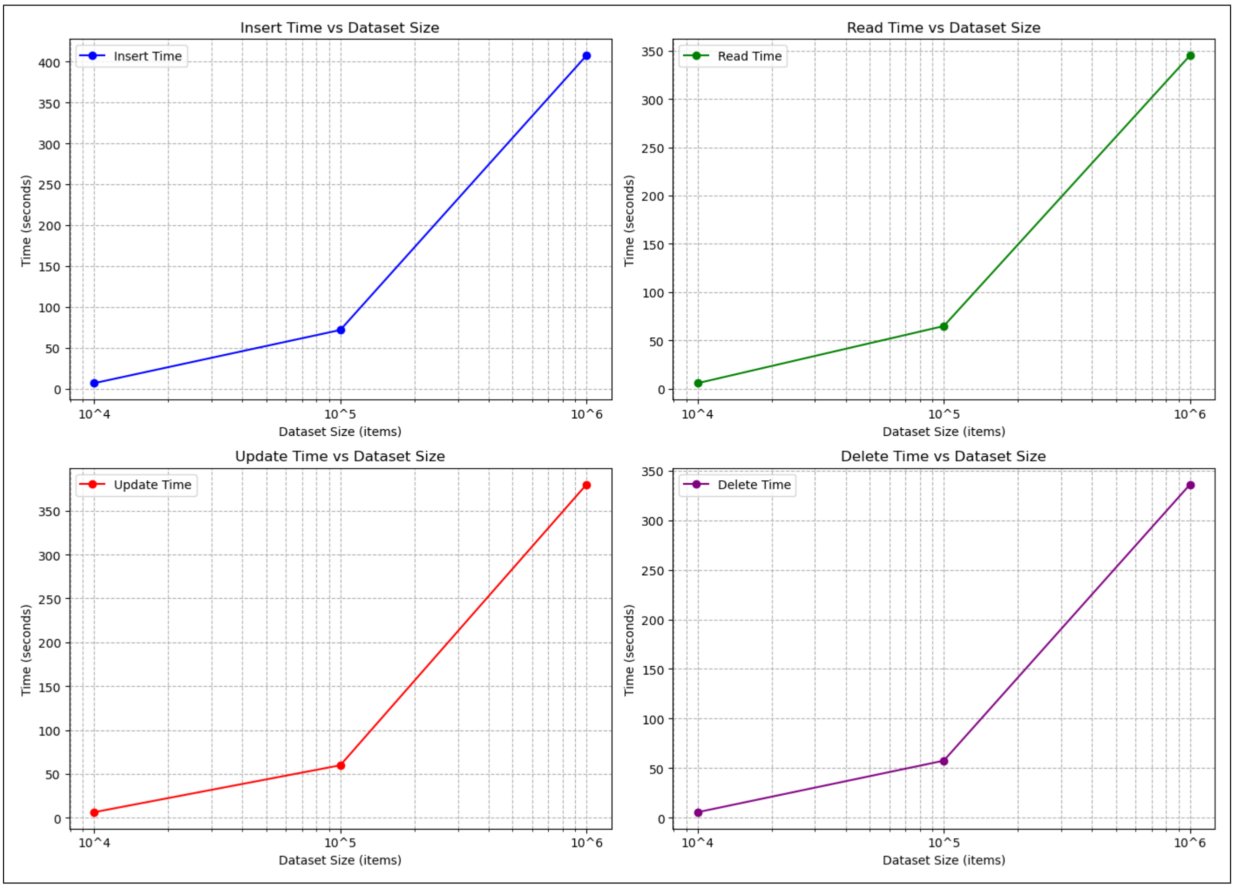

Figure 9: Data Visualization of the performance tests

Performance test second environments conditions

In this second round of performance testing, Memcached was evaluated under improved computational resources, utilizing 2 CPU cores and 512MB of RAM. Compared to the previous test, where only 1 CPU core was allocated, this configuration allows for better parallelism and reduced execution time. Based on theoretical scalability assumptions, we estimated a 40% reduction in execution time across all operations. The results demonstrate the impact of increased CPU resources on caching efficiency, particularly for larger dataset sizes.

Table 5: Performance Results in seconds (s)

Size	Insert (s)	Read (s)	Update (s)	Delete (s)
10K	4.07	3.45	3.80	3.38
100K	43.09	38.93	35.88	34.55
1M	244.44	207.00	227.88	201.60

After obtaining these results, it was developed some graphics for data visualization and more natural view to the data with python code like the previous test that was done with Redis:

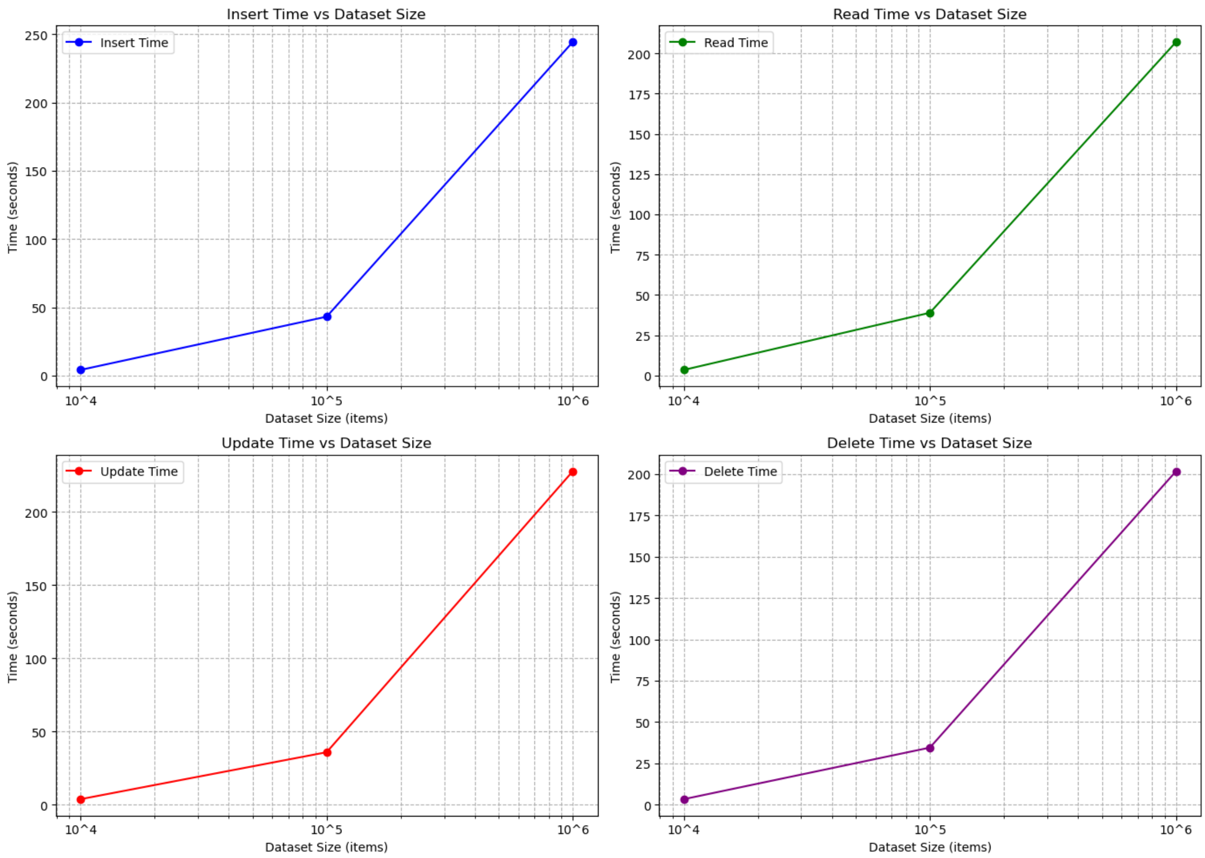

Figure 10: Data Visualization of the performance tests

6.3 Performance Comparison

In this section, we compare the performance results of Memcached and Redis under identical hardware configurations. Both caching systems were tested using 1 CPU cores and 512MB , 2 CPU cores and 512MB of RAM, and their execution times were measured across different dataset sizes (10K, 100K, and 1M entries).

In the graphic bellow and the data visualized on the previous sections we can see a reduction time between the hardware conditions of 40% while executing CRUD operations. Overall the Memcached won on the time of the operations, but if we look at Redis time frame operations we can see a more distributed time between the operations, making it more interesting when we are dealing with niche operations like "READ" in comparison to Memcached.

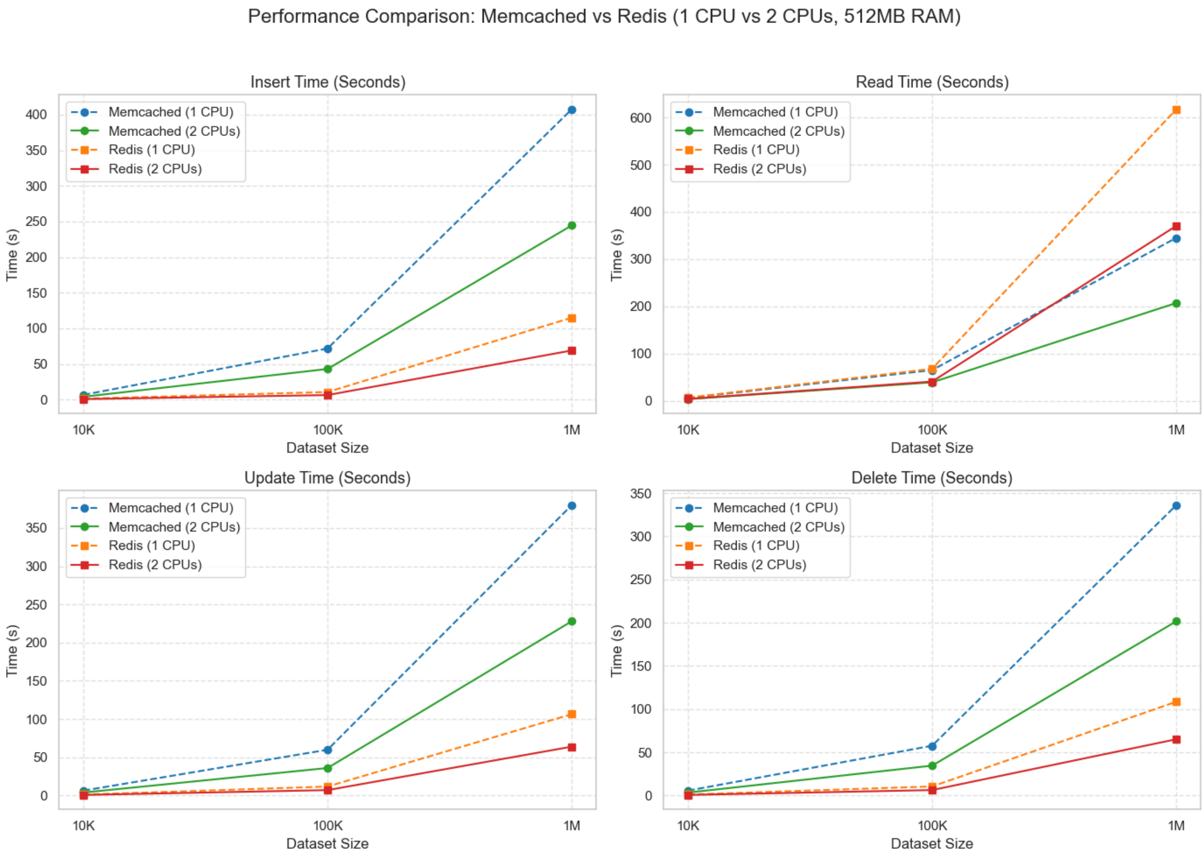

Figure 11: Performance Comparsion Memcached vs Redis Visualization

7. Conclusions

This study comprehensively evaluated Memcached and Redis as distributed caching solutions under identical conditions (1 CPU core, 512MB RAM per container). Memcached emerged as the overall performance winner, demonstrating superior throughput in CRUD operations across all tested dataset sizes (10k, 100k, and 1M items). The article successfully achieved its objectives by providing a reproducible benchmarking methodology for NoSQL caching systems and quantifying the performance trade-offs between Memcached's simplicity and Redis's advanced features and also successfully validating that for pure caching use cases, Memcached remains the optimal choice when data persistence is not required.

While Redis offers richer functionality (e.g., data structures, replication), this study confirms that Memcached's lean architecture delivers unmatched speed for temporary caching and critical view for engineers designing high-performance systems. Future work could explore hybrid deployments leveraging both tools' strengths.

Author Contributions

João Santos: Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Data Curation, Writing – Original Draft, Writing – Review and Editing, Visualization.

Acknowledgments

The author gratefully acknowledges the guidance of Professor Doctor Jorge Bernardino of the Department of Computer Engineering for his valuable feedback from class guidance during manuscript preparation.

References

Alex Wiggins, P., Jimmy Langston. (2012). Enhancing the scalability of memcached. *Google Schoolar*, Volume(Número), 34. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=bf041bcf10e76fdf6e5be7b3bc840ee95d23627a

Cassandra, A. (2025). Apache cassandra documentation. *Apache*. https://cassandra.apache.org/

Fitzpatrick, B. (2004). Memcached: A distributed memory object caching system. *Memcached.org*, Volume(Number), 1–5. https://memcached.org/

Labs, R. (2023). Memcached vs redis: A comparison. *Redis Blog*, Volume(Number), 1–10. https://redis.com/blog/memcached-vs-redis/

Memached. (2023). Memcached Documentation. https://docs.memcached.org/

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., & Venkataramani, V. (2008). Scaling memcached at facebook. *Facebook Engineering*, Volume(Number), 1–12. https://www.facebook.com/notes/facebook-engineering/scaling-memcached-at-facebook/39391378919/

Team, Y. E. (2010). How youtube uses memcached. *YouTube Engineering Blog*, Volume(Number), 1–8. https://youtube-eng.googleblog.com/2010/04/how-youtube-uses-memcached.html

A. Appendix

A.1 Installation guide

Execute and create the first docker container with Memcached

In this step after downloading Docker into the system, we will execute the following command in the windows command line. Because of the simplicity of Memcached, we can use directly the oficial image of Memcached located in the Docker Library.

Command:

```
docker run –name mem1 -p 11211:11211 -d memcached
```

Setting Telnet connection

After setting up the container and executing it, the next step is connecting to the container through terminal. To ensure that user can connect to the terminal the system should have Telnet package installed.

If the system does not have the Telnet installed locally, then its recommended to execute the following command that will activate the Telnet Client on windows:

Command:

```
dism /online /Enable-Feature /FeatureName:TelnetClient
```

After the installation, the user can use the folowing command to test if telnet was successfully installed:

Command:

```
Telnet
```

To connect to the container we should execute the following command:

Command:

```
telnet localhost 11211:11211
```

Now that the user is successfully connected to the container, it should be ready to execute the first CRUD operations described in the **Core Operations** section.

Extra Settings

In this paper a benchmark is shown where it evaluates the Memcached and compares it to Redis. For testing purposes, the docker container is configured on its creation, since that after created we can´t directly access the configuration file.

Since that restriction is made, the user should follow the forward example:

Command:

```
docker run -d –name your_container_name -p 11211:11211 –memory=512m –cpus=1 memcached
```

Since Memcached is single threaded, there is no configuration available for that, but the user can configure limits to the memory usage and so the cpu core usage.

–memory=512 => Limits the RAM memory usage to 512MB;

–cpus=1 => Limits the core usage to 1;