Academic year : 2024-2025

INFO-H-415

Project Report :

Advanced Database
In-memory databases with Redis and Memcached

Group Members :

Léoplod Guyot

Iftissen Aya

Antonio Baldari

Hilal Rachik

Teacher :

Esteban zemiani

Boris Coquelet

Table des matières

1 Introduction

In the era of data-driven decision-making, the efficiency of database management systems (DBMS) is critical for application performance. This project explores the use of in-memory databases, which store data in system RAM for faster processing compared to traditional disk storage. In-memory databases are particularly useful for real-time applications like financial trading platforms and web caching.

The focus is on two leading tools in this domain : **Redis** and **Memcached**, known for their speed, scalability, and reliability. Redis supports complex data structures, while Memcached is designed for simple, efficient caching.

The objectives of this project are to :

— Understand the benefits of in-memory databases over disk-based systems.

— Implement an application using Redis and Memcached to compare their performance.

— Conduct a benchmark using YCSB and a custom benchmark to assess scalability and efficiency with datasets ranging from 1K to 1M records.

The dataset used is derived from the NASA Kennedy Space Center HTTP server logs, providing a challenging workload for testing. This project aims to assess the strengths and weaknesses of Redis and Memcached, offering insights into their application in real-world scenarios.

2 Project Background

2.1 In-Memory Databases

In-memory databases (IMDBs) store data in main memory (RAM) instead of disk drives, enabling faster response times. By avoiding disk queries, they excel in rapid data processing and real-time analytics, making them vital in industries like telecommunications, banking, gaming, and travel.

Also known as main memory databases (MMDB), real-time databases (RTDB), or in-memory database systems (IMDS), IMDBs face a key limitation : RAM's volatility, leading to data loss during crashes or power outages. This issue is mitigated by non-volatile memory (NVRAM), which retains data after power loss. Emerging NVRAM and flash memory technologies offer persistence, though flash memory faces durability issues due to limited write cycles.

2.2 Comparison : In-Memory vs Traditional Databases

— **Speed :** In-memory databases are faster due to direct access to RAM, while traditional databases rely on disk access.

— **Volatility :** IMDBs lose data in the event of power failure unless supported by NVRAM or persistence solutions. Traditional databases store data on disks, making them less volatile.

— **Data Redundancy :** Traditional databases often create redundant data copies for components, whereas IMDBs manage data efficiently through direct pointers.

— **Applications :** IMDBs are ideal for real-time analysis, whereas traditional databases excel in long-term data storage and reliability.

Aspect	**In-Memory Databases (IMDBs)**	**Traditional Databases**
Speed	Faster due to direct access to RAM	Relies on disk access, which is slower
Volatility	Lose data in power failure unless supported by NVRAM or persistence solutions	Store data on disks, making them less volatile
Data Redundancy	Manage data efficiently through direct pointers	Often create redundant data copies for components
Applications	Ideal for real-time analysis	Excel in long-term data storage and reliability

TABLE 1 – Comparison between In-Memory Databases and Traditional Databases

2.3 Memcached

Memcached is a lightweight, distributed, in-memory key-value store optimized for high-speed data access. It operates by caching frequently accessed data in RAM, reducing backend database load and improving application response times, particularly under heavy traffic.

As a key-value store, Memcached pairs unique keys with corresponding data values. These keys allow quick lookups(The process of retrieving specific data from a data structure or database), enabling efficient data retrieval without complex query processing. The simplicity of the key-value model makes Memcached particularly well-suited for scenarios requiring rapid access to precomputed or frequently requested data, such as session information or API responses.

Applications : Memcached is ideal for caching database query results, session data, and frequently accessed web resources. Its key-value design makes it effective for use cases like storing user authentication tokens, product catalog information, or configuration settings, significantly enhancing web application performance.

2.4 Redis

Redis is an open-source, in-memory key-value data store that supports a wide variety of data structures, including hashes, strings, lists, sets, and sorted sets. Redis provides ultra-fast performance, with operations executed in less than a millisecond.

Applications : Redis is well-suited for use cases like caching, session management, real-time analytics, and pub/sub messaging.

2.5 Comparison Table : PostgreSQL, Redis, and Memcached

Feature	PostgreSQL	Redis	Memcached
Architecture	Relational DBMS	In-memory key-value store	In-memory key-value store
Data Storage	Disk-based	RAM with optional persistence	RAM only
Data Structures	Tables, rows, columns	Strings, hashes, sets, lists	Strings (key-value only)
Speed	Slower (disk I/O)	Fast (sub-millisecond)	Fast (microseconds)
Scalability	Vertical and horizontal	High (supports clustering)	High (distributed nodes)
Persistence	High	Optional	None
Applications	Complex queries, analytics	Caching, session storage	Caching, quick lookups
Language Support	SQL	APIs for various languages	APIs for various languages
Transaction Support	ACID Transactions	No transaction support	No transaction support
Use Cases	Data warehousing, OLTP	Caching, real-time analytics	Caching, session storage

TABLE 2 – Comparison of PostgreSQL, Redis, and Memcached

3 Methods

3.1 Automated benchmark : YCSB

3.1.1 Experimental Setup

All benchmarks were performed on the same Lenovo laptop with the following characteristics :

— Model : Thinkpad 20TACTO1WW

— System type : 64 bit

— OS : Ubuntu 22.04.5 LTS

— RAM : 32 GB

— Processor : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 MHz, 4 Core(s), 8 Logical Processor (s)

3.1.2 Overview

Yahoo! Cloud Serving Benchmark (YCSB) developed by Yahoo! Research is a well known database management systems (DBMS) benchmarking tool distributed as an open-source software

with simple database operations on synthetically generated data.

We will use YCSB as it allows benchmarking of both NoSQL and traditional SQL DBMS (jdbc driver) and supports all 3 DBMS required : Redis, Memcached and PostgreSQL.

Core Workloads

YCSB includes a set of 6 core workloads that define a basic benchmark for DBMS, we will use the following 3 :

As the 3 other workloads are either redundant (C to B) or too specific (D and E).

— **Workload A : Update heavy**

This workload has a 50/50 reads and writes operation distribution. Updates in this workload do not presume you read the original record first. The assumption is all update writes contain fields for a record that already exists.

— **Workload B : Read mostly**

This workload has a 95/5 reads/write operation distribution. As with Workload A, these writes do not presume you read the original record before writing to it.

— **Workload F : Read-modify-write**

In this workload, the client will read a record, modify it, and write back the changes. This workload forces a read of the record from the underlying datastore prior to writing an updated set of fields for that record. This effectively forces all datastores to read the underlying record prior to accepting a write for it. A random delta for the write is used (counter).

3.1.3 Benchmark Schemes

2 YCSB performance benchmark schemes were implemented :

— Throughput vs Latency

— Recordcount vs Latency

Throughput vs Latency

Objective : Evaluate the trade-off between latency and throughput, measuring latency as throughput increase to asses DBMS' ability to perform under load.

Scheme : For every DBMS

(default recordcount = 100k, default operationcount = 100k)

— Empty database

— Load phase with workload a

    — Run phases with every (workload, throughput) combinations

Recordcount vs Latency

Objective : Evaluate the impact of dataset size on the average latency of operations, assessing DBMS's ability to manage larger datasets.

Scheme : For every (DBMS, workload, recordcount) combinations

— Empty database

— Load phase with iteration specific workload and recordcount

— Run phase with iteration specific workload and recordcount

3.2 Case study : HTTP-NASA

3.2.1 Experimental Setup

Benchmarks were performed on a laptop with the following characteristics :

— Model : MacBook air M1 2020

— System type : 64 bit

— RAM : 8 GB

3.2.2 Application and dataset

To use the two different in-memory technologies in a real life scenario, we conceptually defined an application that would use this kind of database. This application would be an analytical dashboard that present the current load on website. This would be use internally by an company that has a popular website with lot of users. This dashboard would have access to the HTTP requests logs of a website, it would then use these logs to compute useful metrics and draw plots. The goal of this dashboard would be to inform the admins of the current load on the website to monitor it in real time. It would also be interesting to know which are the peak hours of activity.

To simulate the type of data that we would use in this application we utilized a dataset derived from the NASA Kennedy Space Center HTTP server logs, publicly accessible through the [Internet Traffic Archive](https://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html). The logs cover two months of HTTP requests from July and August 1995. The dataset comprises approximately 3.5 million HTTP requests. Each log entry includes essential fields such as the host (hostname or IP address of the client), timestamp (date and time of the request), request line (HTTP method, resource path, and protocol), HTTP reply code (status code like 200 or 404), and bytes (size of the response in bytes).

3.2.3 Benchmark procedure

Note that we choose not to include postgreSQL in our tests since it was already showed with the YCSB benchmark that postgreSQL is too slow for this application (cf. Results section). To prepare the raw NASA HTTP logs for benchmarking, minimal preprocessing was performed. The raw log files were downloaded, decompressed, and each log entry was parsed to extract the relevant fields mentioned earlier. The parsed data was then converted into JSON format.

The structured dataset was loaded into the databases as part of a Java benchmarking handmade program. The Java program read the JSON file, constructed the dataset in memory, and inserted the data into the databases in a 'key value' basis with the id of the request as the key and the unstructured fields as value.

The benchmarking tool was developed in Java, utilizing the Yahoo Cloud Serving Benchmark (YCSB) framework. It was designed to evaluate the performance by performing a series of database operations;

INSERT, READ, UPDATE, DELETE, and a Custom Query on datasets of varying sizes (1000, 10 000 and 100 000 records). For each measure a 3 technical replicates were made.

The INSERT operation measured the time taken to store a fixed number of records into the database. The tool iterated over the dataset, inserting records using their unique keys. Timing was measured using ‘System.currentTimeMillis()‘ before and after the operation, and the total time taken was recorded.

The READ operation assessed the time required to retrieve all loaded records. For each key generated during the INSERT operation, the tool read the corresponding record from the database, measuring the total time taken in the same manner as the INSERT operation.

The UPDATE operation evaluated the performance of modifying existing records. For each record, a specific field (‘bytes‘) was updated by incrementing its value by 1,000. The tool read the existing record to obtain the current ‘bytes‘ value, incremented it, and updated the record in the database. Timing was recorded as before.

The Custom Query operation simulated a more complex data retrieval scenario. Since Redis and Memcached are key-value stores without native support for complex queries, the custom query was implemented at the application level. The objective was to find all records where ‘http_reply_code‘ is ‘200‘ and ‘bytes‘ is greater than ‘5,000‘. The tool read each record, applied the filtering criteria, and collected the keys of matching records. The total time taken and the number of matching records were recorded.

The DELETE operation measured the efficiency of removing records from the database. The tool deleted each record using its unique key, again recording the total time taken.

The results of each operation were recorded for analysis. The benchmarking tool followed a sequence for each database and dataset size : initialization (configuring database-specific properties and initializing the database connection), data loading, performing benchmark operations while recording timings, result recording, and cleanup (closing the database connection and releasing resources).

4 Results

4.1 Automated benchmark : YCSB

4.1.1 Overall load runtimes

The first results obtained with YCSB is the load time to insert records in the databases (Fig. 1). As expected, PostgreSQL is noticeably slower compared to the two in-memory alternatives, given its disk-based architecture (two last number of counts are not shown as they took too much time).

In contrast, Redis and Memcached demonstrate significantly faster runtimes as a result of their in-memory nature. Interestingly, the run-time between these two is quite similar, with Memcached showing a slight edge in performance, though not by a large margin. This close performance between Redis and Memcached suggests that both are well optimized for tasks requiring rapid data insertion, but Memcached might have a marginal advantage in this specific use case.

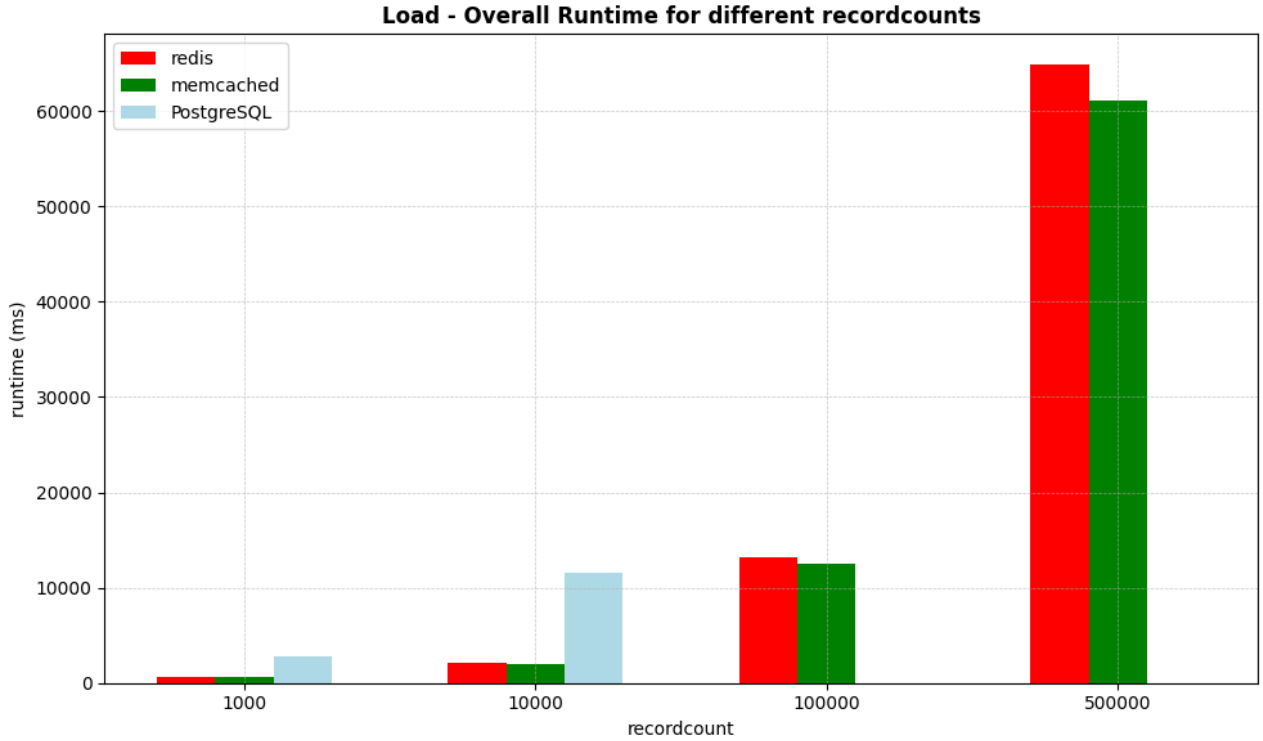

FIGURE 1 – Overall runtime performance (in milliseconds) for loading data with varying record counts across three systems : Redis, Memcached, and PostgreSQL. The x-axis represents the number of records, ranging from 1000 to 500 000, while the y-axis indicates the runtime in milliseconds. Note that the data for postgres with 100 000 and 500 000 counts is not present because it was too time consuming.

A second important metric that impacts the loading of the data into the database is the average latency as the number of inserted counts increases (Fig. 2). We observe that the latency decreases as the number of counts grows. This behavior is logical because, as more counts are inserted, batching or caching mechanisms in the systems become more effective, leading to improved efficiency per operation.

PostgreSQL, however, stands out with a noticeably larger average latency compared to the two in-memory solutions, reflecting its disk-based design.

As with runtime performance, Redis and Memcached show quite similar average latency values, showcasing their efficiency as in-memory solutions. Although Redis might exhibit slightly larger latency on average, the difference is minimal and unlikely to be significant in most practical scenarios.

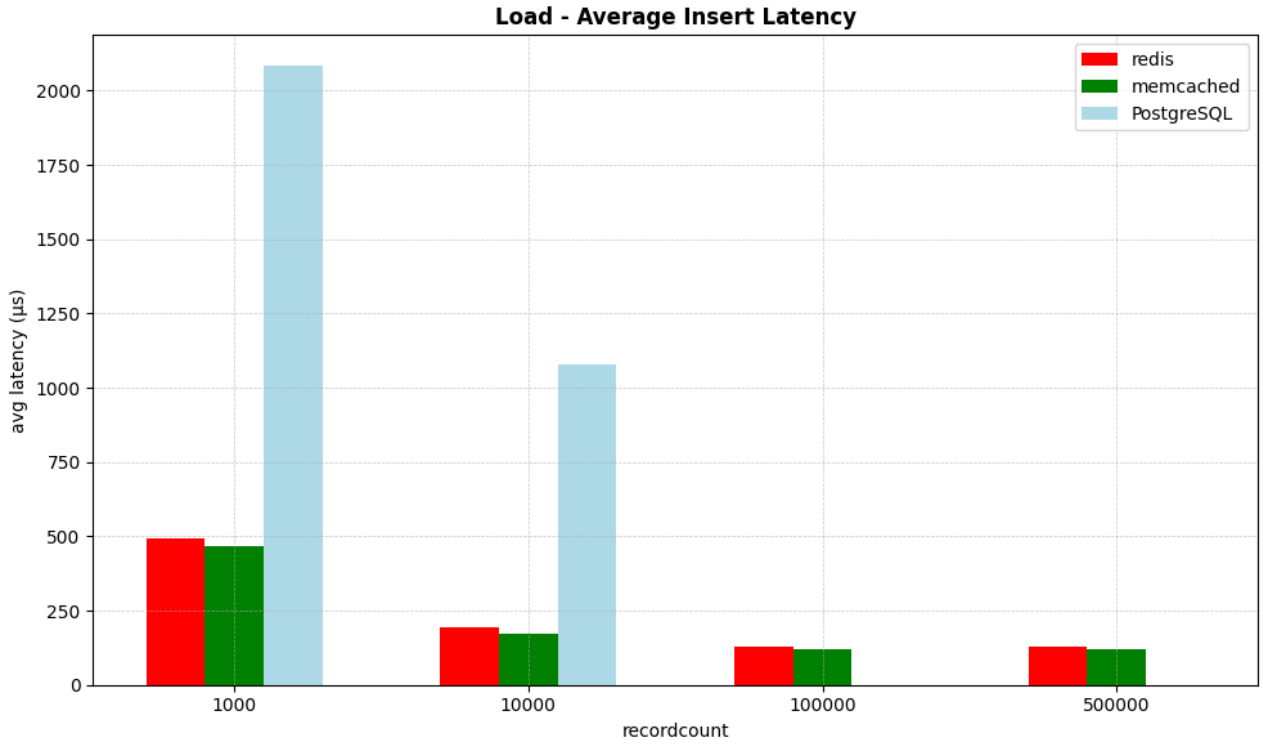

FIGURE 2 – Average insert latency (in microsecond) for loading data with varying record counts across three systems : Redis, Memcached, and PostgreSQL. The x-axis represents the number of records, ranging from 1000 to 500 000, while the y-axis indicates the runtime in microseconds. Note that the data for postgres with 100 000 and 500 000 counts is not present because it was too time consuming.

4.1.2 Throughput vs Latency

— Redis achieves lowest latency

Redis consistently achieves the lowest latency for all recordcounts and workloads. This is explained by its in-memory architecture (vs PostgreSQL) and its optimized data structures like hash and sets (vs Memcached basic key-value).

— PostgreSQL vs Memcached on read operations

At smaller dataset sizes (1k, 10k), PostgreSQL slightly outperforms Memcached for reads despite being a disk-based relational database. This can be explained by PostgreSQL having several optimizations that allow it to perform well for small datasets, like efficient indexing and caching mechanisms (in-memory cache, for small record counts, much of the dataset can reside in this memory cache).

As the dataset grows (100k, 100k), Memcached logically outpaces PostgreSQL due to its key-value

store architecture allowing constant time look ups where PostgreSQL manage indexing and relationships. And due to its in-memory architecture Memcached avoid disk acces overheads which increases as recordcount increase (cache evictions, tree look up complexity, ...).

Workload a

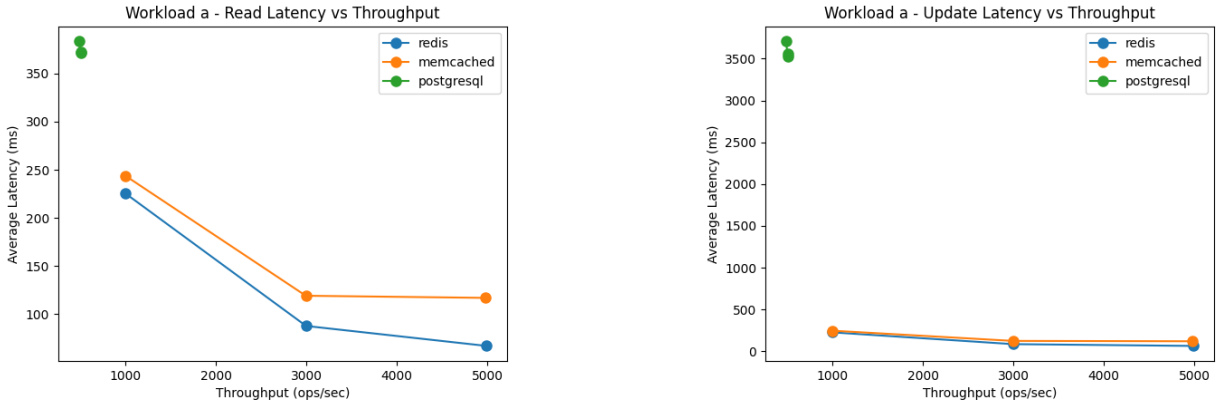

FIGURE 3 – Workload a : Throughput vs Latency

Workload b

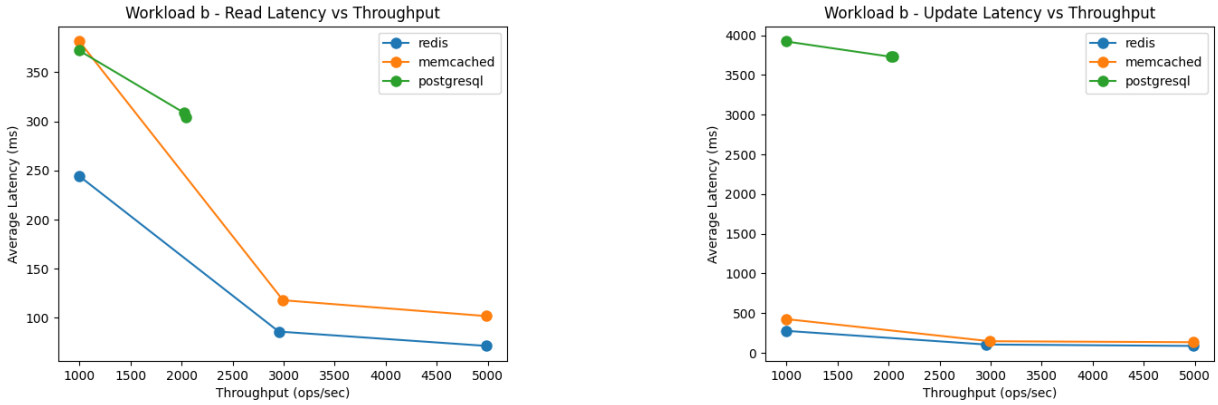

FIGURE 4 – Workload b : Throughput vs Latency

Workload f

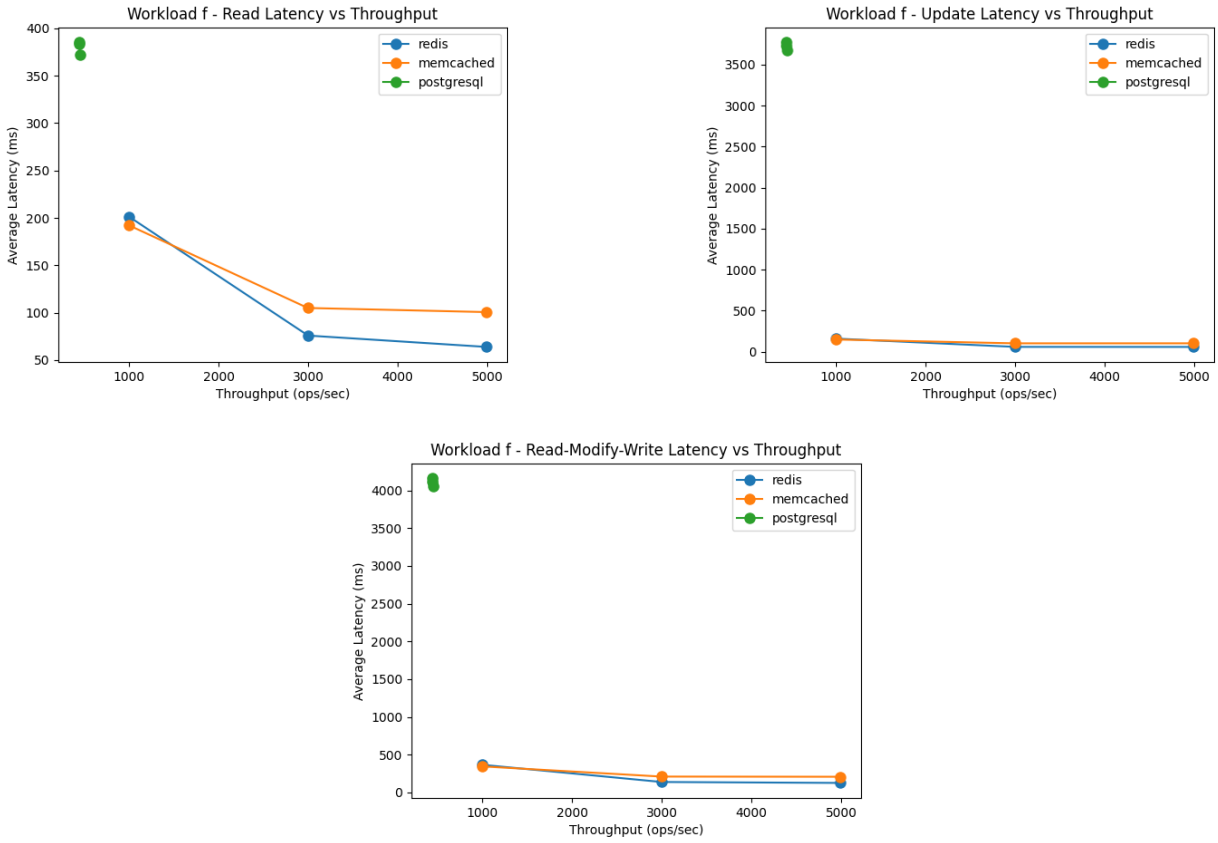

FIGURE 5 – Workload f : Throughput vs Latency

4.1.3 Recordcount vs Latency

— PostgreSQL throughput bottleneck

PostgreSQL fails to reach the target throughput in most cases, plateauing at 500 ops/sec, with slightly better performances on read-heavy workload B reaching 1k-2k throughput. This is explained by disk acces overhead, which is even more noticeable for large "operationcounts".

We also notice a huge drop of performances for Update and RMF operations compared to acceptable performances for Read operation, this is explained by modifying operations requiring index maintenance (MVCC) and additional disk acces, where reads can benefit from PostgreSQL in memory caching mechanism and optimized indexing.

— Redis and Memcached performances

Both Redis and Memcached consistently achieve the target throughput levels with low latencies. Redis slightly outperforms Memcached but shares a similar scaling behavior, showing improved latencies as throughput increases. This is due to combination of in-memory architecture (avoiding disk acces) and key-value architecture (fats look ups).

The improvement in latency as throughput increases might result from optimizations in the workload

distribution or efficient thread handling. However, this trend should reverse if the throughput targets were pushed further, leading to resource saturation.

Workload a

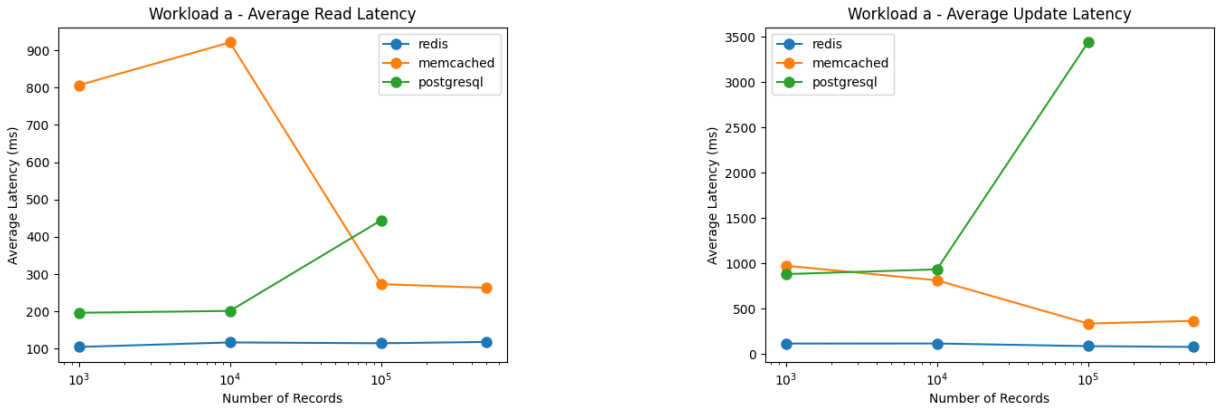

FIGURE 6 – Workload a : Recordcount vs Latency

Workload b

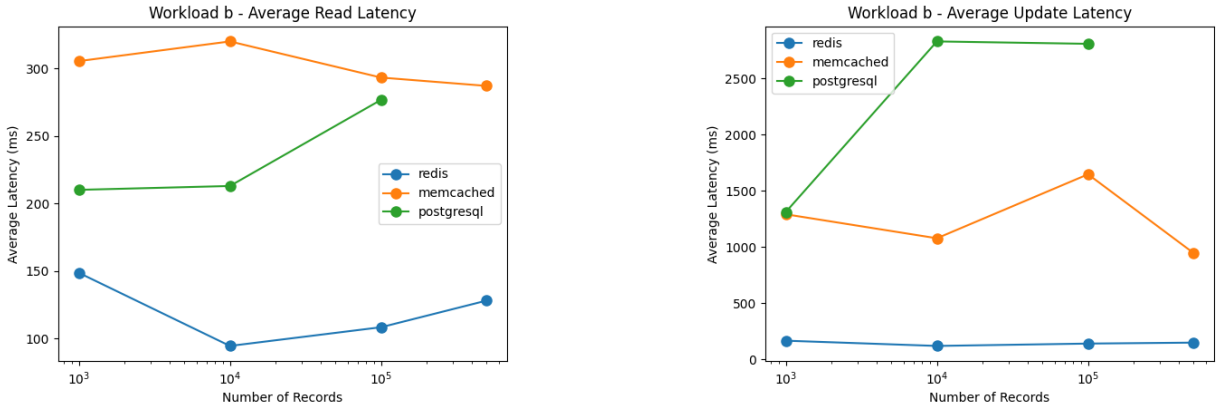

FIGURE 7 – Workload b : Recordcount vs Latency

Workload f

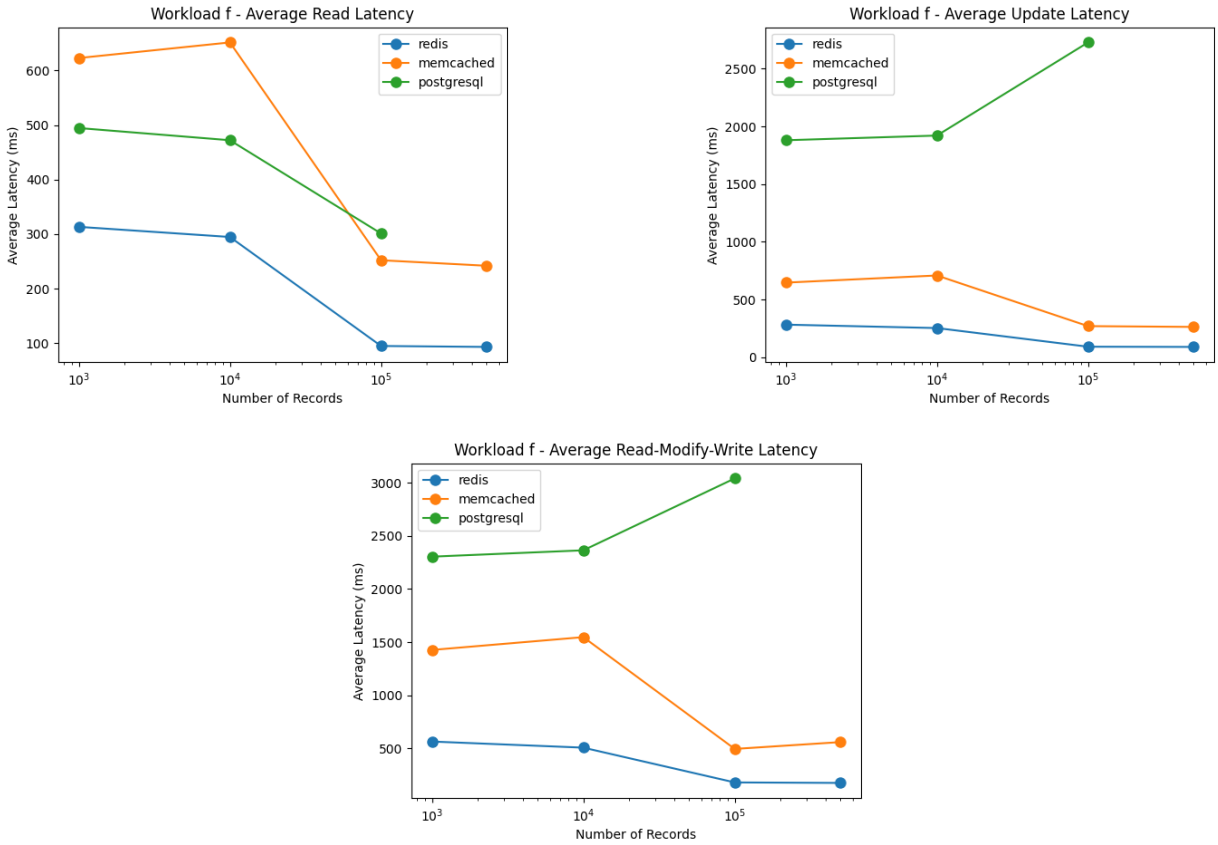

FIGURE 8 – Workload f : Recordcount vs Latency

4.2 Case study : HTTP-NASA

In this section we will see how redis and memcached compare in a real case scenario. The results obtained with this handmade benchmark seem to coincide with the results obtained with YCSB.

First it is important to note that the results obtained does not seem to vary a lot between the technical replicates (3 replicates), which means that we can trust our results. When we look at the results for the insert operation (Fig. 9, we can see that as we increase the number of counts memcached becomes better in term of runtime than redis. This result concord with the YCSB results that showed slightly better performance for memcached on the loading of the data in the database. When looking at the figure 10 we see that for the custom query, the delete, read and update operations have a similar pattern in term of runtime. Indeed, as we increase the number of operations made (counts) the results show that redis has a better runtime. This results also concord with the results from YCSB that showed a better performance for redis when we use operations that interact with already present data (update, read, delete).

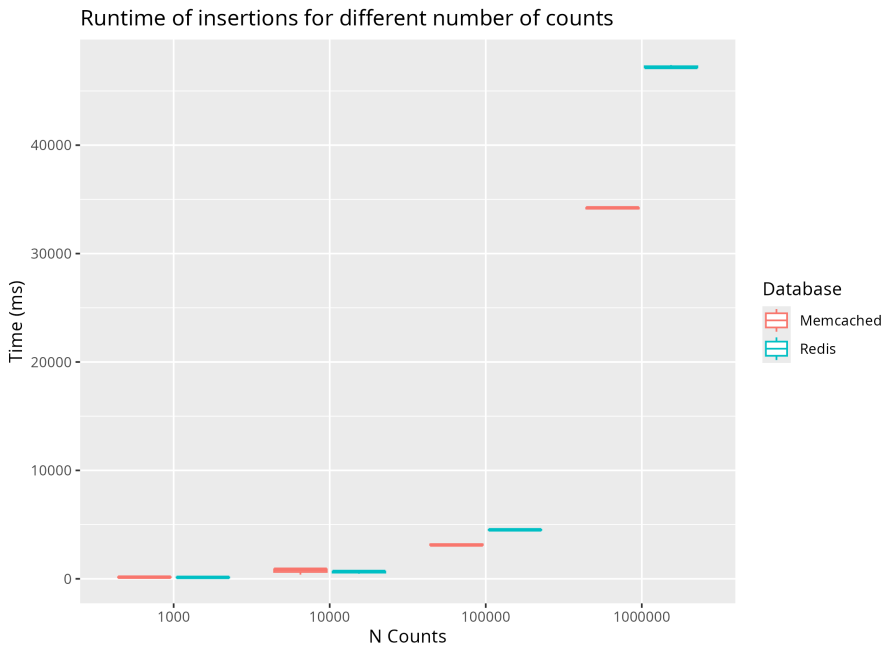

FIGURE 9 – Runtime performance (in milliseconds) of insertions across two systems : Memcached(red), and Redis(blue), as a function of the number of counts (N). The x-axis represents the number of counts, while the y-axis indicates the runtime in milliseconds.

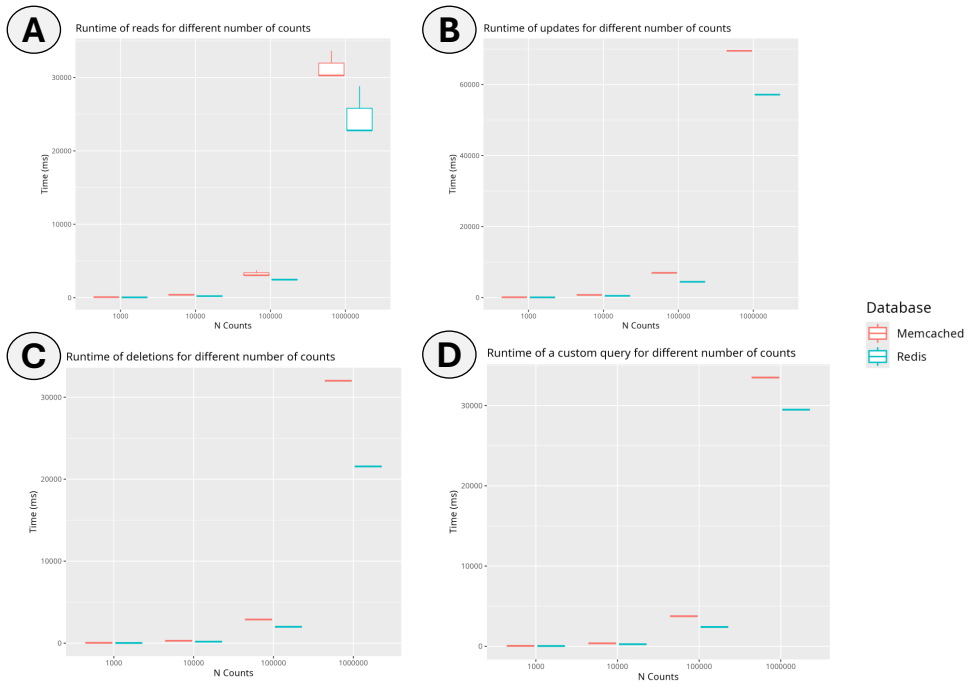

FIGURE 10 – Runtime performance (in milliseconds) of various operations—deletions, reads, updates and a custom query across two systems : Memcached(red), and Redis(blue), as a function of the number of counts (N). Each subplot corresponds to a specific operation. Subplot A illustrates the runtime of read queries, Subplot B focuses on updates, Subplot C shows the runtime of deletion, and Subplot D highlights a custom query (details in methods). The x-axis in all subplots represents the number of counts (N), while the y-axis displays the runtime in milliseconds.

5 Discussion

The results obtained in our benchmarks reveal several important insights into the comparative performance of Redis and Memcached, as well as their suitability for different use cases. The results section highlighted predictable patterns, the discussion here synthesizes these findings to inform practical decision-making in the context of our analytical dashboard application. First, it is important to note that our benchmarks lead to reliable results, indeed there were coherent between the two benchmark and the technical variability of the NASA benchmark was quite low.

By combining insights from both benchmarks, several conclusions emerge :

RAM vs. Disk Memory : As anticipated, PostgreSQL's performance is significantly lower compared to Redis and Memcached. This was highlighted by the latency results coming from the YCSB benchmark. This result underscores the fundamental advantage of in-memory databases, where operations are inherently faster due to their reliance on RAM rather than disk memory. Note that for smaller dataset postgreSQL can achieve similar results than memcached trough the use of caching and indexing.

Redis vs. Memcached Trade-offs : The comparison between Redis and Memcached seems to highlight a trade-off in their design philosophies. Memcached demonstrates superior performance in inserting new data, making it particularly efficient for handling high-throughput HTTP request insertion. On the other hand, Redis excels in operations involving already stored data, benefiting from a higher-cost encoding process that optimizes subsequent interactions with the encoded data. This distinction reflects intentional design choices tailored to different types of workloads.

While Memcached offers raw speed in certain scenarios, Redis provides a broader set of functionalities, which could make it more suitable for specific tasks. For example, Redis supports advanced data structures, such as HSET, that are unavailable in Memcached. Utilizing these structures could enhance the performance of specific queries by enabling more efficient filtering or grouping operations. For instance, filtering HTTP request entries by specific tags like request size could be optimized using HSET instead of traditional value encoding. However, adopting these advanced structures could introduces a trade-off : it may improve query performance but could also reduce insertion speeds due to a additional encoding cost.

Redis also offers more configuration options compared to Memcached. Fine-tuning these settings could potentially improve Redis's performance for specific tasks.

In the context of an analytical database dashboard, the need for an in-memory database is evident. The ability to process data in real-time is critical for such use cases, and both Redis and Memcached offer significant performance advantages over traditional disk-based databases like PostgreSQL. However, the choice between Redis and Memcached requires careful consideration of specific operational priorities.

For our use case, where high-traffic websites may generate substantial numbers of HTTP requests per second, prioritizing faster insertion rates seems prudent. Memcached's superior performance in this area aligns well with the need to handle bursts of incoming data efficiently. Moreover, the refresh rate of the

dashboard can be adjusted to accommodate the computation time required for analyzing stored data (e.g., computing request counts or trends over time). By choosing a longer refresh interval, we can mitigate the impact of slower operational speeds while still delivering timely insights.

We also compared our results to an already published benchmark (Kabakus and al. 2017). In this benchmark study they compared redis and memcached. Their results coincide with our results, indeed they show that redis is slower to write new entries (for 1 000 000 elements, redis = 14 638ms and memcached = 2813ms). Like us they also show that redis is faster to read elements (for a key within 1 000 000 elements, redis = 8ms and memcached = 30ms). These results, similar to our own, give us confidence in our benchmarks, particularly our benchmark using http data.

6 Conclusion

The primary objective of this study was to evaluate and compare the performance of Redis and Memcached. To achieve this, we conducted two benchmarks aimed at highlighting the strengths and weaknesses of these in-memory databases under different conditions.

The first benchmark utilized the automated YCSB tool, it provided useful insights. We observed that both Redis and Memcached were significantly faster than PostgreSQL. Additionally, Redis demonstrated faster read operations, while Memcached excelled at inserting new data.

In the second benchmark, we used a dataset containing HTTP requests to simulate a database environment suitable for an analytical dashboard designed to analyze website traffic. This benchmark reinforced the findings from YCSB : Memcached proved to be faster at inserting new entries, while Redis excelled in interacting with already-stored data. These results align with previous studies (Kabakus and al. 2017), lending credibility to our observations.

Based on these benchmarks, we recommend Memcached for the context of an analytical dashboard, particularly due to its superior performance in insertion operations. Efficient insertion is critical for handling high-traffic websites where large volumes of data must be written to the database quickly.

However, further testing is necessary to validate these results and explore other dimensions of performance. For instance, benchmarks focusing on memory efficiency would be valuable, as the limited nature of RAM makes storage efficiency a critical factor. Additionally, testing these technologies in a fully developed application environment with queries tailored to specific application needs would provide deeper insights into their real-world performance and suitability.

In conclusion, while our study provides a solid basis for choosing between Redis and Memcached, additional research is essential to fully understand their capabilities and trade-offs a specific application.

7 References

Bibliographie

— **Memcached Documentation**

https://memcached.org/

https://aws.amazon.com/memcached/

These official documentations on AWS provides an overview of Memcached, its use cases, and features, particularly as a high-performance, scalable caching solution.

— **Redis Documentation**

https://redis.io/

The official Redis website offers comprehensive information about this in-memory data structure management system, as well as its features and usage.

— **YCSB Documentation**

https://github.com/brianfrankcooper/YCSB/

— **Oracle - Java SE Documentation**

https://docs.oracle.com/javase/8/docs/

This official site provides comprehensive documentation on Java SE (Standard Edition), including libraries, APIs, and tools necessary to develop and run Java applications.

— KABAKUS, Abdullah Talha et Resul KARA (oct. 2017). « A performance evaluation of in-memory databases ». In : Journal of King Saud University - Computer and Information Sciences 29.4, p. 520-525. ISSN : 1319-1578. DOI : 10.1016/j.jksuci.2016.06.007. URL : https://www.sciencedirect.com/science/article/pii/S13 (visité le 16/10/2024).